CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Leveraging network state for software-defined data centre network management
Authors	Sherwin, Jonathan
Publication date	2023
Original Citation	Sherwin, J. 2023. Leveraging network state for software-defined data centre network management. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	© 2023, Jonathan Sherwin. - https://creativecommons.org/licenses/by-nc-nd/4.0/
Download date	2026-11-17 00:51:59
Item downloaded from	https://hdl.handle.net/10468/14965



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Ollscoil na hÉireann, Corcaigh

National University of Ireland, Cork



Leveraging Network State for Software-Defined Data Centre Network Management

Thesis presented by

Jonathan Sherwin, MSc

ORCiD: 0000-0002-8754-4718

for the degree of

Doctor of Philosophy

University College Cork

School of Computer Science and Information Technology

Head of School: Prof. Utz Roedig

Supervisor: Prof. Cormac J. Sreenan

January 2023

Table of Contents

List of Figures v
List of Tables vii
Acknowledgements ix
Abstract x
List of Abbreviations xii
1 Introduction 1
1.1 Research Challenges 2
1.2 Contributions 4
1.3 Publications and Dissemination 7
1.4 Thesis Summary 8
2 Background Research and Literature Review 10
2.1 Introduction 10
2.2 Software-Defined Networking, and Network Softwarisation 10
2.2.1 SDN Architecture 11
2.2.2 SDN Controllers 12
2.2.3 SDN Switches 13
2.2.4 The OpenFlow Protocol 15
2.3 Data Centre Networks 21
2.3.1 DCN Scale 21
2.3.2 DCN Forwarding Devices 21
2.3.3 DCN Topologies 22
2.3.4 DCN Usage Patterns 26
2.4 The Role of SDN in DCNs 27
2.4.1 Network Configuration / Monitoring 27
2.4.2 Performance / Utilisation 29
2.4.3 Security 32
2.5 Challenges of SDN in DCNs Selected as the Focus of this Work 33
2.5.1 Challenge 1: Recording and Reproducing the Past State of a Software-Defined DCN 33
2.5.2 Challenge 2: Model Querying to Provide Deeper Insight into the Past State and Behaviour of a Software-Defined DCN 35
2.5.3 Challenge 3: Addressing Control-Plane Performance Issues in a Software-Defined DCN 37
2.6 Conclusion 39

3 Reproduction of the Topology and State of a Software-Defined DCN from a Control-Plane Message History 41
3.1 Introduction 41
3.2 Motivation 41
3.3 Research Challenges 43
3.4 Requirements and Architecture of the Proposed Solution 45
3.4.1 DCN Control-Plane Message Capture and Storage 46
3.4.2 Reproduction of DCN Topology and State 47
3.5 System Implementation 48
3.5.1 Logging Subsystem 49
3.5.2 Network State Recreation Subsystem 54
3.6 Evaluation 57
3.6.1 Test Environment 57
3.6.2 Functional Verification Testing 61
3.6.3 Message Log Storage Requirements 63
3.6.4 Packet Capture Tool Stress Testing 64
3.7 Conclusions 66
4 Using Snapshots to Speed up the Reproduction Time of an Emulated DCN 68
4.1 Introduction 68
4.2 Motivation 68
4.3 Research Challenges 69
4.4 Snapshot Generation and Utilisation 70
4.5 Snapshot Components Implementation 74
4.6 Evaluation 78
4.6.1 Demonstrating the Benefit of Snapshots 78
4.6.2 Storage requirements of snapshots 81
4.7 Conclusions 84
5 An Ontological Approach to Creating a Temporal and Topological Model of a DCN 85
5.1 Introduction 85
5.2 Motivation 85
5.3 Research Challenges 87
5.4 Ontologies 88
5.4.1 Background 88
5.4.2 Existing Ontologies 91
5.4.3 New Ontologies 93

5.5 Design and Implementation 95
5.5.1 Dealing with Uncertainty 95
5.5.2 Inferring DCN Elements from OpenFlow Control-Plane Messages 97
5.5.3 Representing Change Over Time in a DCN History 115
5.5.4 Implementation 117
5.6 Evaluation 121
5.6.1 Expressing DCN History Queries for Verification 122
5.6.2 Discussion of Evaluation, Queries and Results 129
5.7 Conclusion 130
6 Resolution of Sophisticated Operator Queries on a DCN 132
6.1 Introduction 132
6.2 Analysis of sample queries 133
6.2.1 Query 1 - For what hosts were flow-rules installed in switch tables, but for which no packets were observed in the DCN? 133
6.2.2 Query 2 - What (if any) hosts migrated from one part of the network to another, and what were the details (when, from where, to where)? 134
6.2.3 Query 3 - What VPLS tunnels were configured on the network, and what were the details? 135
6.2.4 Query 4 - What path would a specified packet follow through the network at a particular time? 136
6.3 Research Challenges 136
6.4 A Methodology for DCN Ontological Concept and Query Design 137
6.4.1 Identifying and Adding New Concepts to the Ontologies 139
6.4.2 Inferring Instances of New Concepts 140
6.4.3 Validating Instances of New Concepts 148
6.4.4 Encoding Queries 148
6.5 Implementation Approach 150
6.6 Methods for Evaluation 150
6.6.1 Example of Results for Query 2 151
6.6.2 Example of Results for Query 3 153
6.6.3 Example of Results for Query 4 154
6.6.4 Discussion of Results 155
6.7 Conclusion 156
7 Reducing the Latency of SDN Controller and Hardware Switch Interactions 158
7.1 Introduction 158

7.2 Motivation 158
7.3 Research Challenges 160
7.4 Design 160
7.5 Implementation 162
7.6 Evaluation 165
7.6.1 Problem Verification 165
7.6.2 Experimental Testbed 171
7.6.3 Experimental Testing and Results 172
7.7 Conclusions 176
8 Conclusions and Future Work 177
8.1 Summary 177
8.2 Key Contributions 178
8.3 Future Work 180
8.3.1 Developing a DSL for Querying and Inferencing 180
8.3.2 Extending the Ontologies to Support Other Sources of Control-Plane or Network Management Data 181
8.3.3 Answering Queries about DCN History for PISA Devices 182
8.3.4 Applying Machine-Learning to Control-Plane Message Logs 183
References 185

List of Figures

Figure 2.1: SDN Components 12
Figure 2.2: An initial OpenFlow exchange 19
Figure 2.3: Installing a flow-rule in response to a PacketIn message 20
Figure 2.4: Removing a flow-rule that has been idle beyond a threshold 20
Figure 2.5: Multi-tier DCN topology 23
Figure 2.6: Spine-leaf DCN topology 24
Figure 2.7: Fat-tree topology (k=6) 25
Figure 2.8: SDN – Key Concerns of Relevance to Data Centre Managers 27
Figure 3.1: Architecture Diagram 45
Figure 3.2: Implementation Diagram 49
Figure 3.3: DCN Topology Diagram 58
Figure 3.4: Testbed Diagram 59
Figure 3.5: Distribution of OpenFlow Packet Lengths 65
Figure 4.1: Augmenting a snapshot to compile DCN state for a selected timepoint 73
Figure 4.2: Architecture Diagram 74
Figure 4.3: Implementation Diagram 75
Figure 4.4: CPU time required to recreate a network as it was at a specific time 79
Figure 4.5: OpenFlow messages captured per second by capture filter 81
Figure 5.1: Layers of Ontology Languages 90
Figure 5.2: Hierarchy of Ontologies 94
Figure 5.3: OpenFlow Message Ontology (Partial) 94
Figure 5.4: Inferring Controller Instances from Switch-Controller Sessions (illustration purposes only) 97
Figure 5.5: Relevant PacketIn and LLDP Packet Fields, with Example Values 104
Figure 5.6: Relevant PacketIn and ARP Packet Fields, with Example Values 110
Figure 5.7: DCN History Ontology (Partial) 118
Figure 5.8: Workflow Diagram 118
Figure 5.9: Logical Topology for Original Multi-Tier DCN 123
Figure 5.10: Composite Inferred Topology for Multi-Tier DCN 123
Figure 5.11: Composite Inferred Topology for Spine-Leaf DCN 124
Figure 5.12: Logical Topology for Original Spine-Leaf DCN 124
Figure 5.13: Inferred Switch Instance Lifespans for Multi-Tier DCN Topology 126

Figure 5.14: Inferred Switch Instance Lifespans for Spine-Leaf DCN Topology 127
Figure 5.15: Host Lifespan with Lifespan of Flow-Rules Matching Host as Source of Packets (Multi-Tree Topology) 128
Figure 5.16: Host Lifespan with Lifespan of Flow-Rules Matching Host as Source of Packets (Spine-Leaf Topology) 129
Figure 6.1: Query and Concept Creation Flowchart 138
Figure 6.2: Migrated Hosts 153
Figure 6.3: Inferred Path Charted for Specified Packet 155
Figure 7.1: Temporary tunnelling of new flow 161
Figure 7.2: Steps taken by a controller to handle a new flow 163
Figure 7.3: Virtual testbed for timing flow-rule updates 167
Figure 7.4: Flow-rule Installation Delay 169
Figure 7.5: Physical Testbed 171
Figure 7.6: Rolling mean TCP connection setup delay at specific flow arrival rates 174

List of Tables

Table 2.1: OpenFlow 1.0 12-tuple 16
Table 2.2: OpenFlow message categories 18
Table 2.3: OpenFlow message types 18
Table 3.1: Key Information in OpenFlow Messages 54
Table 3.2: Network Recreation Functional Verification 64
Table 4.1: Snapshot Generator actions for messages processed from capture log 76
Table 5.1: Sample Queries 86
Table 5.2: Meanings of relevant logic symbols 98
Table 5.3: Logic variables used to represent instances of concepts and data properties 98
Table 5.4: Inference Dependencies 115
Table 5.5: Lifespan of Inferred Switches for Multi-Tier DCN Topology 126
Table 5.6: Lifespan of Inferred Switches for Spine-Leaf DCN Topology 127
Table 6.1: List of Sample Queries 132
Table 6.2: Meanings of relevant logic symbols 141
Table 6.3: Logic variables used to represent instances of concepts and data properties 141
Table 6.4: Header Field Values for Specified Packet 154
Table 7.1: Flow-rule Installation Statistics 168

This is to certify that the work I am submitting is my own and has not been submitted for another degree, either at University College Cork or elsewhere. All external references and sources are clearly acknowledged and identified within the contents. I have read and understood the regulations of University College Cork concerning plagiarism and intellectual property.



Jonathan Sherwin, January 2023

Acknowledgements

It would not have been possible to perform the research, development, testing, writing, and reporting behind the production of this volume without the support of significant group of people. Prof. Cormac Sreenan, my supervisor, never failed to provide expert, practical advice and encouragement backed by enormous patience. Thanks to Prof. Ken Brown, my thesis advisor, for sage wisdom at research review days and points between.

My deepest appreciation is due to my wife, Eirin, the uncredited coordinator whose love, care and support carried me through the hardest parts of the journey, and who made it clear that there was no turning back... Our children, Michael, Róisín and Adam were with us all the way, entertaining, boosting morale and providing company for those crucial coffee breaks. My parents, Mary and Frank, have seen me through three degrees without missing a beat. I could not have done it without the solid foundation you gave me.

There is a long list of friends, colleagues and fellow researchers who I will thank in person, but some key individuals deserve a mention here: a succession of heads of department of Computer Science in MTU (Jim O'Dwyer, Tim Horgan, Prof. Donna O'Shea, Dr. Seán Mc Sweeney) who helped me find time and funding for the research; MTU registrars (Dr. Barry O'Connor, Dr. Áine Ní Shé) who provided much of the funding; Dr. Fatima Gunning of Tyndall National Institute and Dr. Paul Gunning of BT, who gave me access to equipment; Maciej Kuzniar of Google, for sharing his proxy code and his advice on hardware SDN switches; Megan Katsumi of University of Toronto, for pointers on developing ontologies.

A last mention goes to Bella, whose unbiased appraisals of my updates at online meetings with Cormac were a valuable reminder to keep everything in perspective. There's a time to work, and a time to rest. The work is mostly done, for now. Time to plan a rest, and to re-engage with family and friends…

Abstract

This work addresses key aspects in the management of a software-defined multi-tenant Data Centre Network (DCN). Software-defined networking (SDN) offers significant potential benefit for many aspects of DCN management – e.g. configuration, monitoring, performance and utilisation, and security – and many researchers have worked, successfully, over the last number of years to show how improvements to these areas could be achieved. The insight that motivated this work is that the focus by researchers and tool vendors has been on managing the current state or desired future state of the DCN. Meanwhile, issues relating to reviewing, analysing and evaluating the past state of a software-defined DCN have been neglected. SDN has been essential for DCNs to meet the requirements of a multi-tenant cloud environment - through automated, dynamic reconfiguration of network devices and services to accommodate tenants arriving, leaving, and modifying their resource requirements via a self-service interface. Consequently, a DCN operator cannot be expected to know the current state of the network, let alone past states, without tools to query the network and provide that information. DCN operators are well served by tools to manage and query the current state of an SDN, but not previous network states and events.

The research question addressed in this work is how to record the topological and configuration history of a DCN, present a useful view of that historical record to the DCN operator, and provide the means for the operator to query the historical DCN state. A related question is how to employ SDN itself to improve the performance of a software-defined DCN. Solving these issues is challenging because of the scale of a DCN (potentially scaling to hundreds of switches connecting thousands of devices), the constantly changing configuration of a multi-tenant software-defined DCN, and the duration of months or years over which the history must be recorded, analysed and available for querying.

The selected approach takes advantage of the decoupled nature of the control- and data-planes in SDN, to eavesdrop on communications between an SDN controller and switches. Emulated and conceptual models of a DCN have been created from

key information extracted from the captured communications. The conceptual model relies on a combination of new and existing ontologies, with an extensible set of concepts and relationships that are the target of complex operator-focussed queries.

The contributions of the work are as follows:

1. Accurate reproduction of an emulated copy of a DCN (topology and state) for any time in its history solely from a passively captured log of control-plane messages.

2. The use of snapshots to speed up the reproduction time of an emulated DCN with topology and state for a particular point in time.

3. Use of an ontological approach to creating a temporal and topological model of a DCN, populated solely from a log of control-plane messages, and created for the purpose of answering operator queries.

4. A methodology to develop additional sophisticated queries against the ontological model regarding the historical state of and events on a DCN.

5. A method to improve the veracity of control-plane messages by reducing the latency of interactions between SDN controllers and hardware switches in a DCN.

List of Abbreviations

APIC	Cisco Application Policy Infrastructure Controller
BCC	Berkeley Packet Filter Compiler Collection
BDDP	Broadcast Domain Discovery Protocol
BGP	Border Gateway Protocol
BGP-LS	Border Gateway Protocol - Link State
BPF	Berkeley Packet Filter
CAM	Content Addressable Memory
DCN	Data Centre Network
DPI	Deep Packet Inspection
DPID	Datapath Identifier
DSL	Domain-Specific Language
ECMP	Equal Cost MultiPath
ETE	Energy-aware Traffic Engineering
GENEVE	Generic Network Virtualisation Encapsulation
IPFIX	Internet Protocol Flow Information Export
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
LDP	Label Discovery Protocol
LLDP	Link Layer Discovery Protocol
MEBN	Multi-Entity Bayesian Network
MPLS	Multi-Protocol Label Switching

NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualisation
NML	Network Markup Language
NOS	Network Operating System
NSX	VMware Network Virtualization and Security Platform
NTP	Network Time Protocol
NV	Network Virtualisation
NVGRE	Network Virtualisation using Generic Routing Encapsulation
ONIE	Open Network Install Environment
ONL	Open Network Linux
ONOS	Open Network Operating System
OVSDB	Open vSwitch Database Protocol
OWL	Web Ontology Language
OWL-DL	Web Ontology Language Description Logic
PCE-P	Path Computation Element Communication Protocol
PISA	Protocol Independent Switch Architecture
PTP	Precision Time Protocol
QoS	Quality of Service
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representation State Transfer
SDN	Software-Defined Networking
SDNDL	Software-Defined Networking Description Language

SFC	Service Function Chaining
SHACL	Shapes Constraint Language
SPARQL	SPARQL Protocol and Resource Description Format Query Language
SPB	Shortest Path Bridging
SPIN	SPARQL Inferencing Notation
SQL	Structured Query Language
SQWRL	Semantic Query-Enhanced Web Rule Language
STP	Spanning Tree Protocol
SWRL	Semantic Web Rule Language
TCAM	Ternary Content Addressable Memory
TLS	Transport Layer Security
ToS	Type of Service
TRILL	Transparent Interconnect of Lots of Links
TTL	Terse Resource Description Framework Triple Language
UML	Universal Markup Language
URI	Universal Resource Identifier
VLAN	Virtual Local Area Network
VPLS	Virtual Private Local Area Network Service
VXLAN	Virtual Extensible Local Area Network
XML	Extensible Markup Language
YAML	Yet Another Markup Language

1 Introduction

There has been robust growth in the number of data centres globally, and in the size of the largest data centres over the last couple of decades. Annual global data centre traffic (including traffic into, out of, and within data centres) rose from 1.8 ZetaBytes (ZB) in 2011 to 6.8 ZB in 2016, with a continuing expected compound annual growth rate of 25% [1, 2] suggesting a figure of over 25 ZB for 2022. The number of hyperscale data centres doubled from 2015 to around 600 in 2021, with plans to continue building these facilities at close to a linear growth rate [3, 4]. Many of the large and hyperscale data centres are multi-tenant, where the data centre is owned and operated by one entity, but resources are leased and used by 3rd party customers. Some of the 3rd parties use their resources to operate virtual data centres for their own group of customers.

The ascent of data centres was fuelled by high-bandwidth and low-latency Internet connectivity, inexpensive commodity hardware, and technologies such as virtualisation and software-defined networking (SDN). These technologies have been core to the provision of cloud services from data centre facilities. Cloud services present a self-service front-end to users that must be backed by software for timely and dynamic provisioning of resources and application services in response to user actions. Cloud services have been enthusiastically adopted by businesses and consumers, for applications ranging from backup and recovery, to collaboration, to media streaming. Indeed, the work-from-home mode of operation forced on many organisations and their employees by the COVID-19 pandemic accelerated the investment in and use of cloud services and facilities.

Data centre resources can be divided into compute, storage, and network. The flexibility required for cloud computing for compute and storage resources is provided by virtualisation. SDN unlocked the same level of flexibility for network resources, by automating network configuration tasks that had previously been mostly manual and disjointed. Network configuration must take account of multiple paths within the data centre network (DCN), congestion, traffic prioritisation, and

the possibility of link or device failure. In a multi-tenant DCN, the configuration is constantly changing dynamically as tenants arrive, leave, and change their set of leased resources or the network policy they desire for their part of the network.

SDN programmatically controls the resources and functions of a network to make the network more dynamically configurable to match the requirements of users and applications. Its advantages over static configuration have seen SDN adopted in many different types of networks, for example enterprise / campus, wi-fi, mobile, sensor, and wide area networks, as well as in data centre networks. SDN is often implemented using the open, standard OpenFlow control-plane protocol, although there are other implementation options. From its genesis, SDN was adopted enthusiastically in DCNs, for several reasons:

1) Devices within a DCN are under the administrative control of a single entity (the DCN operator), thus lending the environment to the centralised control that is a common feature of SDN implementations.

2) SDN implementations scale well to handle many devices. DCNs can contain hundreds of forwarding devices (switches / routers) connecting thousands of servers. Managing static configurations at that scale, even with the help of automation tools, is error-prone, cumbersome, and too slow.

3) SDN can be used to implement and update fine-grained, complex policies. Tenant churn in multi-tenant DCNs results in constantly changing network policies, and traditional static configuration could not accommodate resulting dynamic changes required to provide basic reachability between the resources of one tenant while enforcing isolation between tenants, let alone more complex requirements such as load-balancing and congestion control.

1.1 Research Challenges

SDN has proven indispensable for data centres, however the benefits that it brings are not without cost. Of necessity, the DCN operator cedes control of the device-level network configuration to the SDN controller, which automatically adjusts the

configuration to meet the policy set by the operator (and tenants). Assuming a well-defined policy, and a reliable SDN controller, the delegation of control is not a major concern in other network environments, for example enterprise networks or WANs. In those environments, configurations remain stable and static for long durations, so changes can be flagged and investigated as exceptions when necessary.

In a multi-tenant DCN, configuration changes constantly, as outlined earlier, making it difficult for a DCN operator to definitively identify what the configuration is currently, or indeed what it was at any time in the past. This opaqueness makes it problematic to troubleshoot problems, or to respond to tenant queries about traffic patterns in the network. Vendors have added support for SDN to their network management tools (e.g. [5]), and cloud operators using SDN have developed their own monitoring tools [6], however these focus on on-line monitoring. They do not gather enough information for effective post-hoc analysis.

Moreover, SDN implementations are not equal. Hardware designers and software engineers make choices based on their interpretations of the requirements, or are limited by the resources they have available. The Open Networking Foundation has facilitated interoperability testing and conformance certification [7] for switch implementations of the OpenFlow standard, providing a certain level of assurance for potential purchasers of OpenFlow switches. However, significant differences still exist between such switches, notably in the performance of the OpenFlow implementations, restricting DCN operator's equipment purchasing choices or risking this one component of the data centre infrastructure being a bottleneck that reduces the efficiency and therefore profitability of the entire operation.

The major research challenges addressed by this work are:

- How to assist a DCN operator observe and query the historical configuration of their constantly-changing network.

- How to improve control plane performance in a DCN for the purpose of ensuring low-latency update of network state to match the DCN operator's specification.

1.2 Contributions

Thesis Statement: *DCN control-plane messages can be efficiently captured and stored, and used to build accurate models of the network from which they were logged, to support sophisticated network management query facilities for DCN operators.*

The contributions of this thesis are as follows:

1) *Accurate reproduction of an emulated copy of a DCN (topology and state) for any time in its recorded history, solely from a log of control-plane messages*. The purpose of reproducing the network is to enable DCN operators to test and experiment with the DCN as it was at any point in the past. While tools were previously available for reconfiguring a network with a previous state based on captured control-plane messages, the approach of extracting topology information from those messages to create an emulated copy of the DCN – with state, and for any time in its history - is novel.

2) *The use of snapshots to speed up the reproduction time of an emulated DCN with topology and state for a particular point in time*. Knowledge of the topology and state of the network is learned incrementally based on the information in control-plane messages. The quantity of control-plane messages exchanged per unit of time in a multi-tenant DCN is high compared to other environments (e.g. campus, WAN), and the message log could cover periods of months or years. Rather than having to 'replay' the message log from the start each time the DCN operator wishes to reproduce the network for a different point in time, an approach was taken to process the full message log once, creating regular snapshots of topology and state. Then, the DCN can be reproduced quickly (and in bounded time) for any

point in time covered by the log by starting with the closest prior snapshot and augmenting that with information from messages captured in the log between the snapshot time and the desired time point for reproduction. Previous approaches were non-bounded, always starting from the beginning of the message log, albeit with a 'fast-forward' function, and assumed a small message log captured over a relatively short duration of a few hours.

3) *An ontological approach to creating a temporal and topological model of a DCN*. Reproducing a DCN for a specific point in the network's history is useful for analysing the state and behaviour of that network at that point in time. DCN operator queries of a broader temporal nature require a different approach – queries such as 'What switches were active in the network within a specific interval?', or 'For what hosts were flow-rules installed in switch flow-tables, but no packets forwarded based on those rules?'. Answering these kinds of queries necessitates a model of the network that includes the dimension of time. The use of an ontological approach to creating a temporal model of a DCN is novel, and provides the benefit of making consistency and correctness more achievable. The approach allows the model to be created dynamically, solely based on a set of ontologies and a log of control-plane messages exchanged between the switches and controllers in the DCN. The ontologies define concepts including the elements of a DCN and the relationships between them, and are populated with instances of those elements, and properties of the instances, inferred from the control-plane messages.

4) *Resolution of sophisticated operator queries regarding the historical state of and events on a DCN*. Sophisticated, high-level, temporal DCN operator queries can be applied against the populated network model to provide meaningful and useful answers. New, complex queries can be developed by

adding more concepts (including abstract concepts defined in terms of existing concepts) and relationships to the ontologies, and coding the queries to make use of the existing and new concepts. The novelty provided here is the ability to answer queries about the DCN's historical state and events. The state of the DCN is defined as the switch configuration (flow-table contents) and topology (the switches, SDN controllers, active hosts, inter-switch links and host-switch links). The events that can be identified via queries are switch configuration and topology changes. The model does not contain information about specific data-plane packets, inactive devices, or any configuration change or activity that has not been signalled within the logged control-plane data.

5) *Reducing the latency of interactions between SDN controllers and hardware switches in a DCN*. Hardware SDN switches have been shown to be slow at processing control-plane requests from controllers. At busy times, requests to modify flow-tables can queue up at a switch, and it falls behind on responding to the controller – and this is clearly shown in the timing of the relevant control-plane messages captured in a message log. If the message log is subsequently used to analyse and query the historical state of the network, the widening intervals between controller requests and switch responses reduces the certainty of when events occurred (such as when a modified flow-rule took effect). Since software switches do not suffer from the same flow-modification delay as hardware switches, and as the first and last switches on a communication path between two virtual machines in a data centre network are software switches, an approach was devised to temporarily tunnel new flows between software switches while waiting for the appropriate rules to be installed in the flow tables of the hardware switches. The rules to tunnel a new flow are installed quickly by the software switches, so later queries about the flow can be answered more accurately, and there are the additional benefits that flow start time is

decreased, and that there is time for the non-tunnelled path to be verified before the remainder of the flow is switched to use that path rather than the tunnelled path. Note that a flow will experience jitter and some packets may arrive out of order when that flow is being migrated to a non-tunnelled path. Tunnelling is commonly used for static overlay or virtual networks, but this use of temporary tunnelling, which can be done selectively for specific types of flows and/or at busy times, is novel.

1.3 Publications and Dissemination

Much of the work presented in this thesis has been subjected to peer review through submission to and presentation at multiple international conferences. The publications are listed below, grouped under the chapter each publication supports. All papers listed were co-authored by Jonathan Sherwin (the author of this thesis) and Cormac Sreenan (the thesis supervisor).

Chapter 2 - Background Research and Literature Review:

- " *Software-Defined Networking for Data Centre Network Management: A Survey*" - Published on arXiv.org June 2021, arXiv:2106.10014.

Chapter 3 - Reproduction of the Topology and State of a Software-Defined Data-Centre Network from a Control-Plane Message History; Chapter 4 - Using Snapshots to Speed up the Reproduction Time of an Emulated DCN with Topology and State for a Particular Point in Time:

- "*LogSnap: Creating Snapshots of OpenFlow Data Centre Networks for Offline Querying*" - Peer-reviewed full paper presented at the 10th International Conference on the Network of the Future (NoF) 2019, published on IEEE Xplore.

Chapter 5 - An Ontological Approach to Creating a Temporal and Topological Model of a DCN; Chapter 6 - Resolution of Sophisticated Operator Queries Regarding the Historical State of, and Events on, a DCN:

- "*Inferring and Querying the Past State of a Software-Defined Data Center Network*" - Peer-reviewed full paper submitted to the 8th International Conference on Software Defined Systems (SDS) 2021, published on IEEE Xplore.

Chapter 7 - Reducing the Latency of Interactions Between SDN Controllers and Hardware Switches in a DCN:

- "*Improved Data Centre Network Management Through Software-Defined Networking*" – Peer-reviewed short paper presented at the Collaborative European Research Conference (CERC) 2016, published in the CERC 2016 proceedings ISSN: 2220 - 4164
- "*Intelligent Management of OpenFlow Rule Installation in Data Center Networks*" – Peer reviewed abstract, and poster presented at the Conference on emerging Network EXperiments and Technologies (CoNEXT) 2016, unpublished.
- "*Reducing the Latency of OpenFlow Rule Changes in Data Centre Networks*" – Peer-reviewed short paper presented at the 21st Conference on Innovation in Clouds, Internet and Networks (ICIN) 2018, published on IEEE Xplore.

1.4 Thesis Summary

The remainder of the thesis is organised as follows:

Chapter 2 presents background information, related work, and the state of the art of managing DCNs with SDN.

Chapter 3 describes the problem of analysing the historical configuration state of a DCN, and the approach that has been taken for this work to reconstruct network state from a record of control-plane messages. The reconstructed network state is used to produce an emulated copy of the original DCN for any time in the recorded history.

Chapter 4 details an optimisation to the approach described in the previous chapter, that uses snapshots of network state to allow an emulated copy of the network to be reproduced in bounded time.

Chapter 5 presents an ontological approach to creating an extensible model of a DCN with temporal and topological properties. Elements of the model are inferred from a log of control-plane messages captured from a DCN.

Chapter 6 describes how the ontological model from Chapter 5 can be used as the target for sophisticated operator queries, revealing behaviours and attributes of the network that are challenging to extract but that provide crucial insights to an operator.

Chapter 7 addresses a performance issue with switch implementations of the OpenFlow SDN protocol, presenting an approach that leverages the control-plane performance of software switches to mask the relatively poor performance of their hardware equivalents.

Finally, Chapter 8 summarises the achievements and key findings, and catalogues opportunities for future work.

2 Background Research and Literature Review

2.1 Introduction

Chapter 2 provides the context for the contributions of this thesis. A background to Software-Defined Networking (SDN), the OpenFlow protocol, and Data Centre Networks (DCNs) is presented. This is followed by a categorised review of relevant research work under the heading of the Role of SDN in DCNs. Three challenges are then described that were selected as the focus of Chapters 3 to 7.

2.2 Software-Defined Networking, and Network Softwarisation

The term 'Software-Defined Networking' was coined in 2009 [8], and defined as the process of remotely controlling networking hardware with software. The approach was an advance on the active networking and control-/data-plane separation initiatives of the late 1990's and early 2000's. A major milestone was the OpenFlow control-plane protocol, introduced in 2008 [9], and standardised subsequently by the Open Networking Foundation [10]. OpenFlow was developed in conjunction with network switch manufacturers, with the result that the protocol was implemented on commodity switches - making SDN available to researchers and practitioners alike.

The OpenFlow protocol has been through multiple minor and major revisions since its inception. The term 'Software-Defined Networking' has broadened beyond OpenFlow to include other protocols and means of controlling network devices remotely via software. Research work continues on the use of NETCONF and BGP for SDN control-plane communication, for example [11, 12]. Other alternatives for software-defined control of network devices include the OVSDB and OpFlex protocols.

The terms Network Virtualisation (NV) and Network Functions Virtualisation (NFV) are related to SDN, all three fitting under the banner of 'Network Softwarisation'. Generally, Softwarised Networks are those where functions that were traditionally

done in hardware (including functions other than packet-forwarding) are instead done in software.

NV abstracts the functionality of a physical network to create one or more virtual networks. The purpose could be to provide a simplified view of the underlying resources, or for controlled sharing of access to those resources between untrusted clients. SDN can be used to implement network virtualisation. Equally, SDN could be implemented over a virtualised network.

NFV is the implementation of network appliance (for example, firewall, intrusion detection system, NAT gateway) functionality in a virtualised form. The virtual appliance has the benefit of being migratable and easier to upgrade or replace than a physical device. SDN is used to ensure that relevant packets are directed to a virtual appliance wherever its topological location.

For a more complete history of SDN, see [13]. For a broad survey of SDN research, the reader is referred to [14]. A more recent survey [15] catalogues the 'flexibility' provided by research outputs on softwarised networking.

2.2.1 SDN Architecture

SDN separates control of a network from the forwarding function. Forwarding is performed by devices, normally categorised as switches, but which forward packets based on more diverse criteria than the layer 2 or 3 capabilities of traditional network switches [16]. SDN forwarding devices do not make decisions other than basic forwarding decisions independently but take instruction from a (usually) centralised controller.

The logic and algorithms are encoded at the controller, which configures the forwarding devices according to the policy defined by the network operator [17]. Packet forwarding is defined as a data-plane activity, while communication between the controller and forwarding devices takes place at the control-plane. Communication at the control-plane is achieved via a control-protocol, for example OpenFlow.

A common controller architecture is to provide an interface for applications (or 'apps') so that logic can be compartmentalised into individual controller applications – a routing app, a firewall app, a NAT app, for example. Apps can subscribe to be notified about network events by the controller, and/or use external sources of data, to contribute to changes to forwarding device configuration. The controller is responsible for combining the output of the apps into a coherent policy that it then translates into configuration instructions for the individual forwarding devices. In this architecture, the interface between the controller and the apps is termed the northbound interface, and the communications between controller and forwarding devices take place through the southbound interface. Some SDN proponents refer to the controller and SDN apps as a Network Operating System (NOS). Figure 2.1 illustrates the major components of SDN and the interfaces between them.

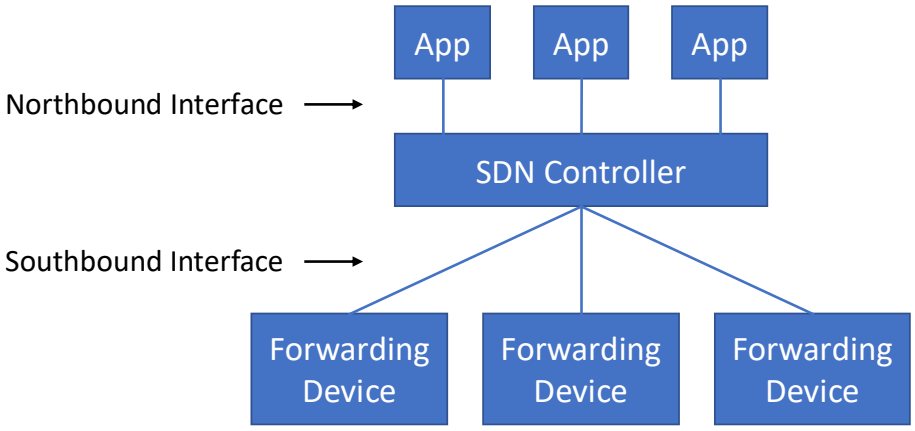


Figure 2.1: SDN Components

2.2.2 SDN Controllers

Since the advent of SDN, a variety of controllers have been introduced. Prominent open-source controllers include ONOS (Open Network Operating System), OpenDayLight, Ryu, POX and Floodlight [18]. Proprietary controllers include Cisco APIC and VMware NSX. Some vendors repackage open-source controllers (e.g. Ericsson and IBM – OpenDaylight; Huawei and Ciena – ONOS) to offer as a vendor-tested and supported option to their equipment and service customers.

Several SDN controllers have a distributed architecture, which is particularly suited to WAN (Wide Area Network) environments where high latency or unreliable links between controller and forwarding devices would cause issues. The majority of SDN controllers are centralised, however they may support clustering for scalability and reliability. In a clustered deployment, multiple instances of the controller are in the same physical location but running in different VMs (Virtual Machines) or on independent physical hardware for fault-tolerance. Centralised controllers are used in campus, wireless, IoT (Internet of Things), and data centre networks. In the latter environment, a clustered deployment is commonly employed. See [19] for a broad survey of SDN controllers.

2.2.3 SDN Switches

The forwarding devices used for SDN are commonly referred to as switches. In common with traditional switches, they often have high port-densities and are capable of fast forwarding rates due to the use of CAM (Content Addressable Memory) and TCAM (Ternary Content Addressable Memory) to store forwarding information [20].

CAM has been used for many years in switches to hold MAC address tables. Because the search function for CAM is implemented in hardware, it is much faster than an equivalent software-based search of a table held in RAM. CAM supports binary (or exact) matching: the item being searched for is either present or not present. This is perfect for traditional layer 2 forwarding, where a row is returned from the MAC address table if one exists for a MAC address taken from the destination MAC address field of an Ethernet frame.

TCAM allows a more sophisticated partial (or wildcard) match, making it suitable for layer 3 forwarding where the longest match with an address in a routing table is usually sought, rather than an exact match. A higher-end router stores its forwarding database (some or all of its routing table) in TCAM to achieve fast layer 3 forwarding. In SDN, wildcard matching may be applied to many fields of a packet. Wildcard matching can be done in RAM, but is much faster using TCAM.

TCAM and CAM are more expensive than RAM, limiting their use to applications where fast lookup is essential, and meaning that SDN switches with greater TCAM resources are more expensive than those with less. Furthermore, TCAM contents must be ordered, which can impede the update process. For a more detailed explanation, see [21].

Traditional switches provide support for control-plane protocols including, but not limited to, Spanning-Tree Protocol (STP), Shortest Path Bridging (SPB), Transparent Interconnection of Lots of Links (TRILL), Label Discovery Protocol (LDP). Some of the software that runs the protocols is processor and/or memory intensive. and is normally implemented in a proprietary operating system. The CPU and memory resources and the cost of development of the operating system are reflected in the price of the device.

SDN switches may support as little as one control-plane protocol, for communicating with the controller, and run a lightweight open-source operating system, making them much cheaper to produce and to buy. Individually, these switches do not match the capabilities of the legacy switches, but when paired with an SDN controller they can provide all of the required functionality.

SDN switches with unfamiliar brand names that are manufactured and sold at low prices are known as 'whitebox' switches. Because of their cost advantage, they are commonly used in data-centre networks where large numbers of forwarding devices are needed.

Traditional switches can still be used for SDN, either via proprietary mechanisms, or through support added to the operating system by the vendor for SDN-specific control-plane protocols. When used in this manner, it should be noted that many of the features, CPU and memory resources of the switch, that have contributed to the cost of the device, may be unused.

The SDN switches described so far are hardware switches. The rise in the use of server virtualisation, particularly in data centres, led to the development of software switches. Software switches logically extend the physical network into the

virtual environment, allowing VMs on one server to communicate with each other, and with other hosts (physical or virtual) across the physical network. Software switches have been created for SDN [22], most notably Open vSwitch, supporting the same control-plane protocols as hardware switches, and appearing to an SDN controller as little different to a hardware switch.

2.2.4 The OpenFlow Protocol

Several standardised control-plane protocols have been used for SDN – NETCONF (Network Configuration Protocol), BGP (Border Gateway Protocol), PCE-P (Path Computation Element Communication Protocol), and OpenFlow, for example [23]. OpenFlow was developed specifically for SDN, was adopted quickly by equipment vendors for inclusion in their devices and is widely used and supported.

OpenFlow 1.0 was released in 2009. There have been major and minor releases since then under the oversight of the Open Network Foundation (ONF), which manages the standard. The latest publicly released version is 1.5.1, although version 1.6 is available to ONF members. The version most widely supported by hardware vendors is 1.3.1 [24].

2.2.4.1 OpenFlow Operation

The OpenFlow specification defines switch and controller roles and responsibilities, and the protocol is used for communication between the switch and controller. A switch initiates a communication session with a controller, either over a TCP or UDP channel. TCP/UDP port 6653 is reserved for an OpenFlow controller, which listens on that port for incoming connections from switches. An OpenFlow switch is identified by a 64-bit value called a Datapath Identifier (DPID). Switches are sometimes referred to as 'Datapaths'.

An OpenFlow switch maintains one or more flow-tables. A flow-table is initially empty, and is then populated with flow-rules sent to the switch by the controller. A flow-rule contains match-criteria and an action-list. The match-criteria is a set of tuples of the form (*field-name*, *value*), and a packet that arrives in on a switchport is matched against flow-rules in turn to find one for which all of the values specified in

the match-criteria match the values in the corresponding fields of the packet. In OpenFlow version 1.0, twelve packet header fields were specified against which a flow-rule could match [25], listed in Table 2.1. By version 1.5.1, the list had expanded to forty five fields, along with mechanisms allowing flexible matching against other parts of a packet.

Table 2.1: OpenFlow 1.0 12-tuple

Input port	Src MAC	Dst MAC	Ether Type	VLAN ID	VLAN Priority	Src IP	Dst IP	IP Proto	IP ToS	TCP/ UDP Src	TCP/ UDP Dst

If a matching flow-rule is found, the actions in the action-list are performed. An action could be to forward the packet through single egress port, to broadcast the packet through all ports, to forward it to the controller, to drop the packet, or to look for another rule to match the packet in another flow-table on the same switch. As a packet is matched against rules in various flow-tables, actions can be carried out immediately, or collected in an action-list for later execution. The behaviour to follow is specified in the rules.

A controller can set two types of timeout, measured in seconds, when adding a flow-rule: an idle timeout, or an absolute timeout. With an idle timeout, the flow-rule is removed the specified number of seconds after the last successful packet match against the rule, or that number of seconds after the rule was added if no packets matched. An absolute timeout causes the flow-rule to expire the specified number of seconds after the rule was added, regardless of whether packets matched or not. Setting both timeout values to zero when adding the rule is an indication that the flow-rule will not expire, and so should not be removed from a flow-table unless the controller explicitly requests that it be removed.

A controller can specify a priority value with each flow-rule. The priority value is used to control the order in which an ingress packet is matched against flow-rules. The lowest priority flow-rule is often a default rule, configured by the controller to

match all packets that did not match against any more specific rule, and with an action to drop the packet or forward it to the controller.

A default rule that drops all packets might be configured on a network were a proactive approach is used, that is where the controller configures flow-rules on all switches in advance to handle packets for all flows allowed to traverse the network.

A default rule to forward otherwise unmatched packets to the controller is appropriate if the network is operating under a reactive regime, where flow-specific flow-rules are configured only after the controller has had sight of the first packet in the flow and evaluated if administrative policy permits that flow to be carried within the network.

The operator's preference may be for proactive installation of flow-rules, for example to minimise the flow-setup time inherent in the reactive approach. However, that preference may not always be possible to meet because switches have finite memory capacity for flow-tables. For best performance, flow-rules will be held in CAM for exact-match flow-rules (e.g. layer 2 forwarding) or TCAM for wildcard matching (e.g. layer 3 forwarding). Some switches operating systems will allow flow-rules to overflow into RAM, albeit with a significant penalty to the speed of matching against those rules. When flow-table capacity is exhausted, switches will select flow-rules for removal if required to add new rules. In that case, reactive flow-rule installation may be a necessary fall-back mechanism. When requesting a new flow-rule to be added, a controller can instruct the switch to send a notification message if or when the flow-rule is removed for any reason.

2.2.4.2 OpenFlow Messages

OpenFlow messages are categorised according to Table 2.2, and the full list of message types defined for OpenFlow version 1.5.1 [26] in each category is shown in Table 2.3. For controller-to-switch messages that require replies, the reply message types are listed in brackets after the request message. Some message types have sub-types, for example a *FlowMod* message, which is used to modify flow-rules, may be further qualified as *Add*, *Modify*, *ModifyStrict*, *Delete* or *DeleteStrict*.

Table 2.2: OpenFlow message categories

Category	Description
Controller-to-Switch	Only sent from a controller to a switch, may require a reply
Asynchronous	Unsolicited messages from switch to controller
Symmetric	May be sent by either a controller or a switch

MultipartRequest messages are used by a controller to request information or statistics from a switch. For example, the controller may request statistics relating to configured flow-rules, including how many packets were matched against the rule since it was added, how many bytes those packets represented, and how long it has been since a packet matched the flow-rule. Similarly, port statistics can be requested to discover how many packets and bytes have been sent and received on a switchport, how many error packets were seen on that port and how many packets were dropped. *MultipartReply* messages can contain a lot of information, requiring them to be split across OpenFlow packets – hence the use of 'multipart' in the message name.

Table 2.3: OpenFlow message types

Controller-to-Switch	Asynchronous	Symmetric
FeaturesRequest (*FeaturesReply*)	*PacketIn*	*Hello*
GetConfigRequest (*GetConfigReply*)	*FlowRemoved*	*EchoRequest*
SetConfig	*PortStatus*	*EchoReply*
PacketOut	*TableStatus*	*ErrorMsg*
TableMod	*RequestForward*	*Experimenter*
FlowMod	*ControllerStatus*	
GroupMod		
PortMod		
MeterMod		
MultipartRequest (*MultipartReply*)		
BarrierRequest (*BarrierReply*)		
RoleRequest (*RoleReply*)		
BundleControl		
BundleAdd		

Figure 2.2 represents an initial exchange of messages between an OpenFlow switch and controller. Through *Hello* messages, the switch and controller agree the version of OpenFlow that they will use for their dialogue. It will be the highest version

supported by both parties. From the *FeaturesReply* message, the controller will learn the DPID of the switch, and how many tables the switch supports.

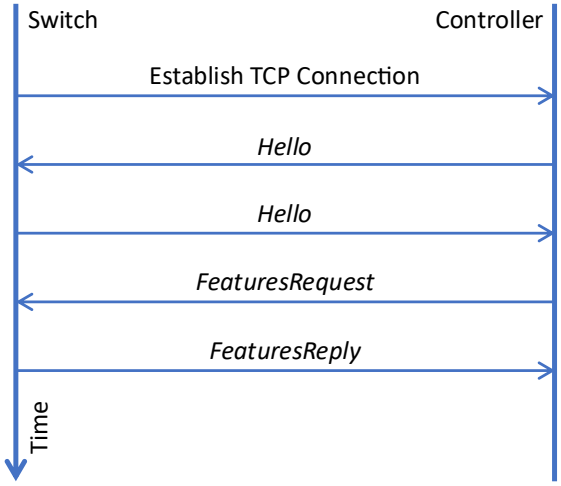


Figure 2.2: An initial OpenFlow exchange

Figure 2.3 represents an exchange of packets resulting in a flow-rule being added to a switch's flow-table. The switch sends a *PacketIn* message containing a packet that did not match against the match criteria for any rule other than the default rule, which forwards otherwise unmatched packets to the controller for processing. The controller assesses the packet against administrative policy, with the result (in this case) that the flow represented by the packet is to be allowed proceed through an appropriate switchport to get the packet closer to its destination. The controller sends a *FlowMod* message to the switch with an instruction to add a flow-rule with match-criteria based on the header fields and field values of the packet received in the *PacketIn* message, and an action to forward matched packets through the relevant port. The last step is for the controller to send the packet from the *PacketIn* message back to the switch in a *PacketOut* message so that packet is forwarded according to the newly added rule.

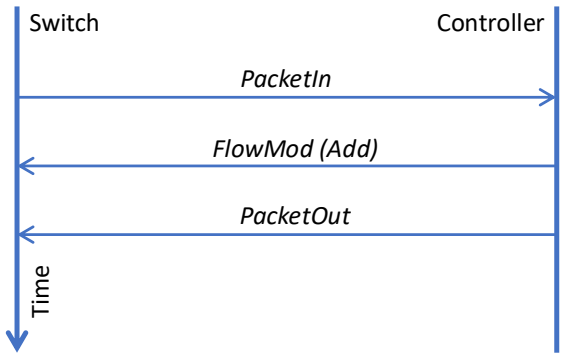


Figure 2.3: Installing a flow-rule in response to a PacketIn message

A third example of an exchange between a controller and a switch is illustrated in Figure 2.4. In this example, the controller queries the switch via a *MultipartRequest* message for statistics regarding a specific flow-rule. The *MultipartReply* from the switch includes information about how long it has been since a packet matched against the flow-rule. From this, the controller determines that the flow has most likely completed for which the flow-rule was added. It sends the switch a *FlowMod* message with an instruction to remove the flow-rule. The switch confirms completion of the instruction with a *FlowRemoved* message.

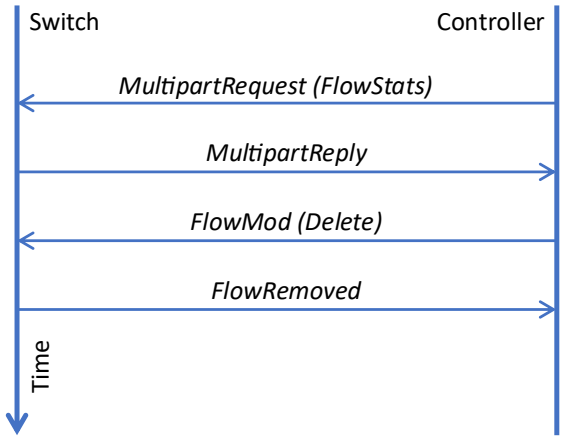


Figure 2.4: Removing a flow-rule that has been idle beyond a threshold

The information provided about OpenFlow here is an overview, provided as context for the following chapters. For the full detail of the protocol operation, readers are directed to the protocol specification [26], or to [21] for a more accessible textbook description.

2.3 Data Centre Networks

The purpose of a DCN is to interconnect a large number of rack-mounted servers [27]. In this respect it differs from other network types such as a campus network, a wide area network (WAN), a small-office / home-office (SOHO) network or a wireless network, among others. Specifically, these factors distinguish DCNs from other types of networks:

- The scale to which they can grow,
- The forwarding devices used within them,
- The topologies used to implement them,
- Their usage patterns.

Each factor is explored below.

2.3.1 DCN Scale

Like any network, DCNs come in different sizes. But when measured in terms of numbers of devices and numbers of connections, the biggest DCNs can be very large. For example, a hyperscale data centre may have tens of thousands of servers connected by thousands of switches [28, 29]. Scaling a network to such a level is achieved through uniformity and modularity: uniformity in the switches used in the network, and modularity in the topology.

2.3.2 DCN Forwarding Devices

To state that there is uniformity in the switches in a DCN is not to say that they are all the same. However, the variety is intentionally limited to make the network easier to manage and for the purchasing power of bulk buying of equipment. Hyperscale operators order devices directly from contract equipment manufacturers who will build switches to requested specifications in large

quantities at a competitive price. The manufacturers then offer similarly specified switches on the open market for further economies of scale.

So-called 'whitebox' switches use commodity parts, prioritise high-speed forwarding, and support the basic required features. They may come without an operating system, leaving the customer to install an operating system themselves. The Open Network Install Environment (ONIE) [30] was designed to facilitate this: manufacturers install ONIE on switches before they ship them, and ONIE helps the customer install the operating system of their choice. For compatible switch hardware, Open Network Linux (ONL) [31] is one operating system offering.

2.3.3 DCN Topologies

The topology of a DCN is fundamental to its operation. As a result, DCN operators tend to stick with a topology with which they have positive operational experience, or one from a vendor's reference design. At the same time, there has been active research into alternative DCN topologies, and some of the ideas have been adopted in production DCNs.

2.3.3.1 Multi-Tier Topology

A multi-tier topology is a classic DCN topology, included by Cisco Systems, for example, as part of its reference design for data centres [32]. The topology is shown in Figure 2.5. In this architecture, each server within a rack is connect to a top-of-rack (ToR) switch. Top-of-rack switches within a cluster are connected in a mesh to edge switches. Edge switches have redundant connections to aggregation switches, and aggregation switches are connected to each other. Aggregation switches are also connected to routers for traffic to exit the data centre to a campus network or the Internet, or to enter the DCN from those external sources. The network design is hierarchical, with switches nearer the top of the hierarchy handling more traffic. VLANs are implemented to contain broadcast traffic below the aggregation switches, which must then do inter-VLAN routing.

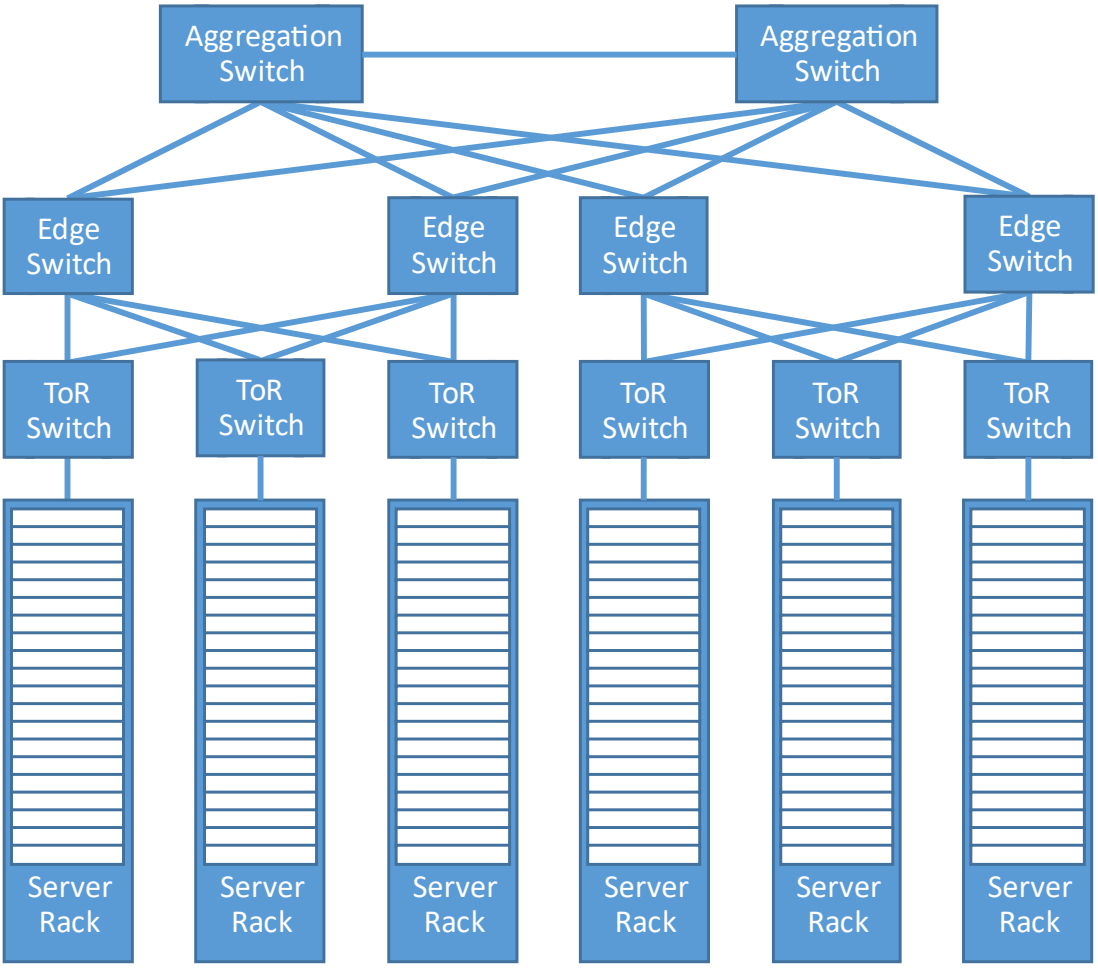


Figure 2.5: Multi-tier DCN topology

2.3.3.2 Spine-Leaf Topology

A spine-leaf design, Figure 2.6, was created for data centres based on the Clos multi-stage design used to build traditional circuit-switched voice networks [33]. The number of layers of switches is reduced to two. Each rack has a leaf switch, to which all servers in the rack connect. Each leaf switch is connected to every spine switch. The design limits the maximum path length between servers within the data centre to three switches. It maximises the number of redundant paths, which can be utilised for equal-cost load balancing. Spine switches are connected to routers for traffic to enter or leave the data centre.

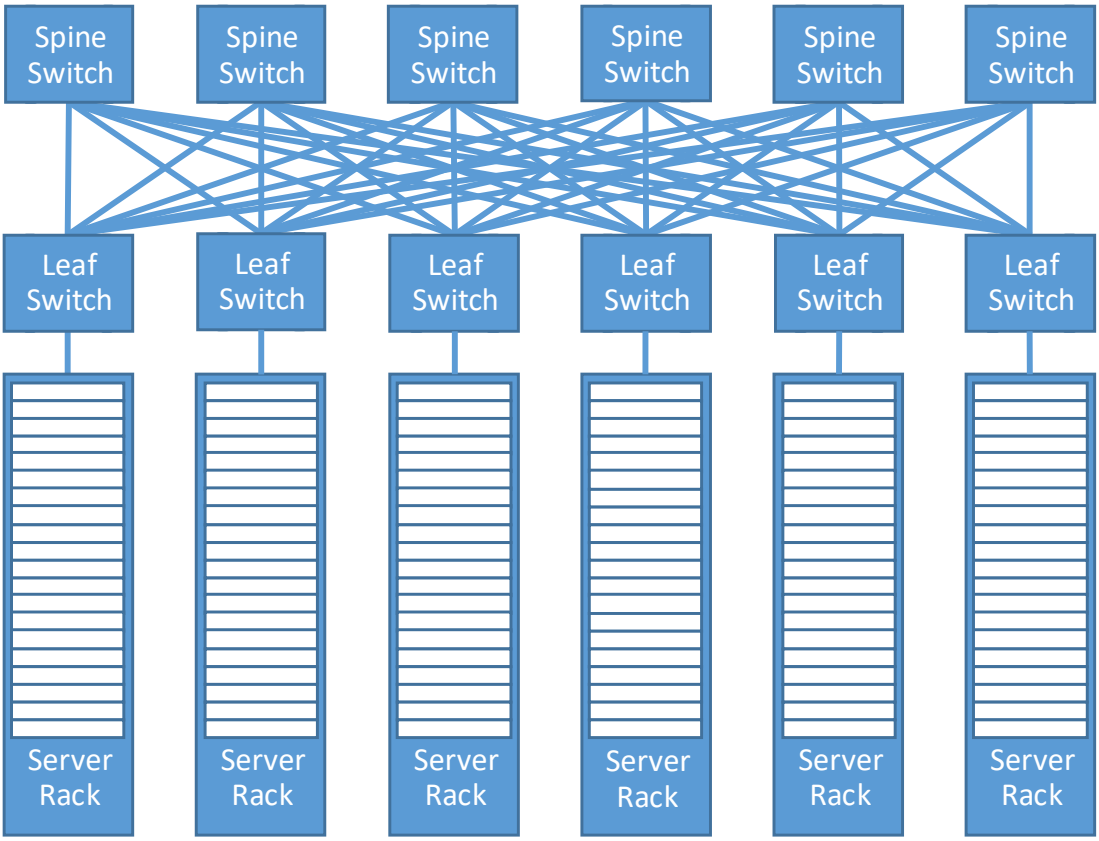


Figure 2.6: Spine-leaf DCN topology

2.3.3.3 Fat-Tree Topology

The fat-tree DCN topology in Figure 2.7 was introduced through the research community [34], has been adopted in production data centres, and enhanced for better scalability. The variable *k* is defined for an implementation of the topology.

There are *k* pods of aggregation and edge switches. Each pod contains *k*/2 aggregation switches, and *k*/2 edge switches, and each edge switch is connected to every aggregation switch within the pod, and also provides connectivity for *k*/2 servers. There are $(k/2)^2$ core switches, and each core switch is connected to one edge switch in each of the *k* pods. The topology shown in Figure 2.7 is for *k*=6. It can be seen that every switch in the topology has *k* ports in use. A notable characteristic of the fat-tree topology is that, assuming all switchports operate at the same speed, the aggregate bandwidth at each layer of switches is the same.

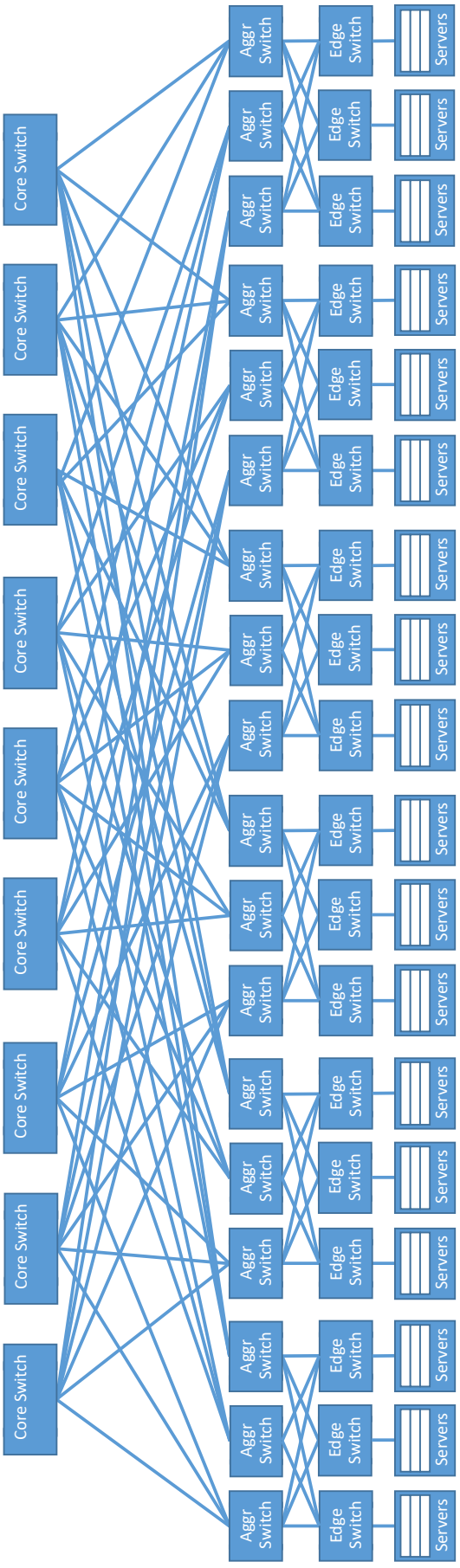


Figure 2.7: Fat-tree topology (k=6)

There are other DCN topologies, including BCube for modular shipping-container based data centres, and c-Through for a hybrid DCN containing electrical and optical switches. For survey of a variety of DCN topologies and architectures, see [27], or for a more recent comparison see [35]. However, the spine-leaf topology has proven durable: it forms the basis of Cisco Systems' current DCN reference design [36], and has been extended to cloud-scale for Facebook's DCN [37].

2.3.4 DCN Usage Patterns

Data centres operate twenty-four hours of the day, every day of the year. Particularly for multi-tenant data centres, there is no downtime during which the whole data centre is taken offline. Individual clusters or pods are taken offline as necessary to service devices or replace failed equipment, with workloads migrated to other areas of the data centre where there is spare capacity. Failures occur frequently due to the use of commodity hardware and the large volume of devices [38].

In a cloud data centre, new tenants can sign up for service, pay for resource reservations and put those resources to work immediately. This is all done through a self-service dashboard. In the background the tenant activities cause adjustments to the network policy to enable communication between the resources that a tenant has reserved. Tenants can change their resource reservations over time, and the cloud operator may offer an autoscaling feature that adjusts resource allocations up or down automatically in response to utilisation [39]. All of the changes must be translated into network device configurations.

The work described in this thesis is focussed on topics relating to the application of SDN in DCNs. As mentioned in Chapter 1, SDN was enthusiastically adopted in data centres, where the growth of the network was constrained by traditional network configuration practices. The next section describes the ways that the research community has devised for SDN to be employed in data centres.

2.4 The Role of SDN in DCNs

To review the application of SDN in DCNs, key concerns of relevance to data centre managers have been categorised, as shown in Figure 2.8. A survey [40] was conducted to identify the key concerns and catalogue the work produced by researchers in applying SDN to the aspects of DCN operation shown in the figure. Notable relevant contributions of their work are described below.

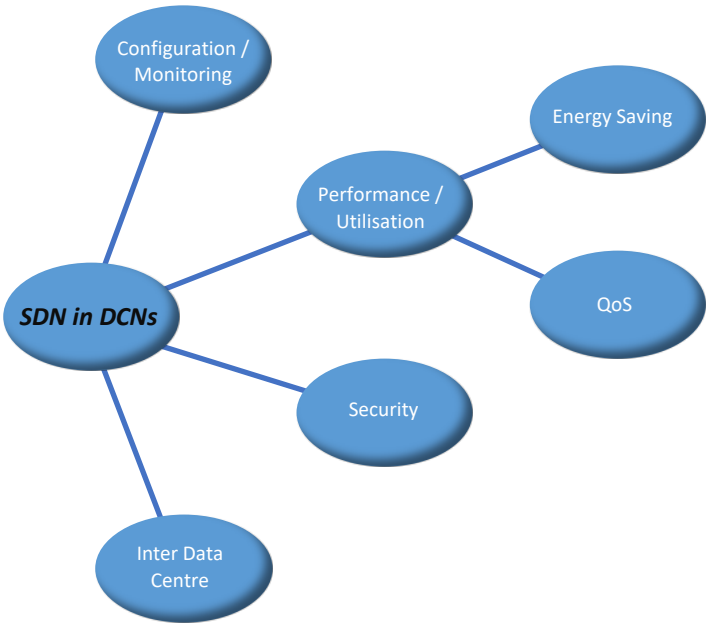


Figure 2.8: SDN – Key Concerns of Relevance to Data Centre Managers

2.4.1 Network Configuration / Monitoring

In this context, configuration is the settings that control the operation of the devices that make up a network. In a DCN, these are switches, routers, and middleboxes, all of which could be physical, but are increasingly also virtualised. A cloud-scale DCN can have more than 1,000 physical switches, and those switches may have different feature sets and capabilities.

Network management tools provide a high-level, abstracted view of the network to operators. The tools map high-level operator actions to lower-level interfaces provided by individual devices or software elements. Integrated into a suite of network management tools, SDN facilitates a centralised, unified view of network

resources, and a single point for monitoring and configuration, allowing fine-grained control over how those resources are shared amongst applications and users. A network needs not be a collection of devices that are individually statically configured according to loose parameters of expected traffic levels and mixes, and that have limited capacity to dynamically react to changes in traffic conditions. SDN-based tools can allow the network manager to set a coherent policy, which is then implemented and policed automatically [41]. The policy can be complex and dynamic. In a cloud data centre, for example, the policy may need continuous adjustment as new tenants are added, old tenants leave, or existing tenants modify their service requirements. In fact, tenants should have the same control over their subset of DCN resources as the network operator has over the DCN as a whole, and SDN has been applied to facilitating this [42].

Presenting a unified view of network resources is not trivial. Not only are there devices with different roles such as switch, router, middlebox to be integrated into that view, but devices performing the same role can have different capabilities – switches that have differently sized CAM/TCAM (used for storing OpenFlow rules and/or caching other forwarding information); switches that support different versions of OpenFlow; switches that can run user-defined code locally, and those that do not. Researchers have tackled some of these issues by taking a software-defined approach to mask the underlying complexity with sophisticated abstractions, for example [43, 44].

Configured network devices need to be monitored to ensure correct operation. There is a benefit to applying SDN here also – for example, by dynamically adjusting the amount of switch resources (CPU and CAM/TCAM) required for monitoring in an OpenFlow network, balancing the resource usage with the accuracy of measurement required [45]. The granularity of the information being collected, and the frequency at which it is collected, can be varied depending on requirements - at a congestion hotspot, it might be acceptable to reduce the detail of monitoring, to prioritise getting data packets through. On the other hand, more detailed monitoring might be required in some part of the network to positively identify a

denial-of-service attack. SDN allows this flexibility in contrast to established practice of dedicating a fixed set of resources to monitoring in a network [46].

Recently, attention has focussed on network telemetry – collecting data on specific packets sent through the data plane for the purpose of fault detection and troubleshooting [47]. A new generation of programmable switches facilitates telemetry, however processing power and memory are limited, and updating the programs requires a switch reboot. Researchers have viewed this set of constraints as a challenge to be overcome [48, 49].

2.4.2 Performance / Utilisation

In the largest data centres, for example those owned and run by public cloud providers, a small increase in performance (greater throughput, lower latency) can translate to a competitive advantage. Techniques used to improve performance include network traffic splitting and traffic engineering, layer 4-7 load balancing, and virtual-machine (VM) consolidation.

Network traffic splitting makes use of redundant network paths within a network. Redundant paths are part of a network design, allowing for device or link failure; however, they represent a significant financial investment, so the motivation is to make use of the resources during normal network operation thereby increasing overall utilisation of network resources. Equal-Cost Multi-Path (ECMP) forwarding is a non-SDN technique commonly used in data centres that splits traffic, on a per-flow basis, across available paths using layer 2 or layer 3 devices. However, ECMP does not take the load of each individual flow into account, so traffic is not fully balanced. An SDN-based solution can do better than simple traffic splitting: Planck [50] built on previous work to monitor data centre switches and provides an application that performs fine-grained traffic engineering by re-routing congested flows within milliseconds (<10ms on 1Gb/s and 10Gb/s networks). Later research initiatives [51, 52] present further refinements to traffic-engineering in DCNs.

Layer 4 load balancing is the distribution of client flows across several data centre servers providing a common service identified by TCP port number. Rather than

replace a heavily loaded server with a bigger server – 'scale-up' – load balancing allows the problem to be solved by adding more servers to share the work – 'scale-out' – and this is transparent to clients. Layer 4 load balancing has traditionally been implemented using dedicated hardware appliances, although more recently vendors have offered virtual appliances. Duet [53] is a combined software and hardware SDN load balancer. Most of the load balancing work is offloaded to DCN switches, tapping under-used resources in switches already in place in the data centre network. This naturally scales with the data centre with minimum cost. Software load-balancing in Duet primarily provides fault-tolerance. Duet uses vendor-specific APIs for controller-to-switch communication, rather than OpenFlow. SilkRoad [54] achieves high-performance layer 4 load balancing using programmable switches.

Layer 7 traffic steering directs packets to servers based on deep packet inspection (DPI) of application-layer data – for functions such as TCP splicing, NAT, layer 7 server selection, or firewall. While traffic steering can be done through other methods – e.g., MPLS tunnels – using SDN allows the steering to be dynamically reconfigured in response to changes in policy, load, or other factors. The authors of [55] propose an SDN architecture in which application-specific packet processing capabilities are distributed to switches in a data centre as 'apps', eliminating the need for dedicated appliances to act as middleboxes. The apps on an individual switch can be chained together to achieve service-chaining, simplifying the path along which packets must be steered. Traffic steering is essential for service function chaining (SFC), and [56] describes how it can be achieved dynamically using SDN.

One way to reduce the latency between two communicating VMs within a data centre is to move them 'closer' to each other – shortening the path between them. This also reduces overall network utilisation because less links and intermediate devices are needed to carry the inter-VM traffic. The role of SDN is firstly to ensure that data packets continue to reach the VM after it has been migrated, so that the move is transparent to the VM itself and communicating partners – in particular so

that no TCP connections are dropped. Secondly, SDN can be used to reserve or allocate network resources to facilitate a quick migration of a VM across the DCN. Of course, the benefit of re-locating a VM must outweigh the cost of migration which itself uses network, CPU, and storage resources [57].

All switches are not created equal – some may have larger flow-tables, higher speed interfaces, faster CPUs. Researchers have addressed the problem of where switches should be placed in a DCN to take best advantage of their individual flow-table capacities [58].

2.4.2.1 Quality of Service

Quality of Service (QoS) implies giving some traffic classes higher priority service than others, possibly to the extent of giving guarantees on latency, bandwidth, and/or packet-loss to specific data flows.

In a data centre, congestion hotspots have traditionally been minimised firstly by generous provisioning of bandwidth, and secondly by careful planning and application of traffic engineering to make balanced use of bandwidth, or to deliberately skew the balance in favour of specific, recurring flows. However, lack of bandwidth is only one of the causes of congestion. Other factors that contribute are buffer capacity in switches and routers, and operations that must be applied on a per-packet basis to packets received or forwarded. In addition, in a cloud data centre, traffic profiles change as tenants arrive and leave, scale their usage up and down, and change the applications they are using – a scenario that is too dynamic to suit traditional traffic engineering. QoS has a role in isolating traffic that requires guaranteed service from the effects of unavoidable congestion, and in prioritising resource allocation to other traffic according to administrative policy. QoS implemented through SDN can flexibly accommodate different service granularity for different flows, and constantly changing policies. SDN can be used to dynamically re-route flows out of a congestion hotspot while maintaining service guarantees. DCN applications that benefit from service guarantees include distributed databases, map-reduce, and virtual networking [59].

One way that QoS has been implemented for Data Centres using SDN is to modify applications to request guarantees in advance for their traffic. In [60], requests are added to a policy tree which is used to resolve conflicts between competing demands – important in an oversubscribed network. The researchers in [59] implement explicit path control - allowing an application to choose paths through the network for its traffic in order to achieve bandwidth/latency guarantees.

2.4.3 Security

SDN has many benefits for network security. In the context of a DCN, SDN can be used to implement a security policy, in place of, or more likely augmenting, the traditional security mechanisms such as VLANs and firewalls. If the security policy changes, an SDN-based solution should dynamically change the operation of the network to match. If the security policy is too complex and nuanced to be realised entirely with static mechanisms or on devices with limited resources, SDN can provide a means to dynamically reconfigure both based on time-of-day, demand, traffic profile, or other criteria.

While the VLAN (or newer VXLAN, NVGRE and GENEVE) functionality of traffic isolation can be provided solely with SDN, researchers have used the flexibility of SDN to leverage VLAN features to do more. For example, deciding VLAN membership based on arbitrary criteria (for example, tenant ID, as in [61]), or re-writing VLAN tags as packets cross a network in order to merge or partially overlap VLANs [62].

SDN is particularly useful for implementing security policy in a cloud data centre where tenants are allowed to do self-service provisioning – creating and migrating virtual machines, creating virtual networks, adding and configuring virtual security devices, setting their own security policies for their virtual cloud [63]. Validating and merging these sub-policies, and verifying that the result does not violate the provider's super-policy, is a complex task that can be solved with the help of SDN: for example, by translating the policies to underlay device configurations, and verifying implemented policy through active packet probing [64].

The widely-used technique of anomaly detection has benefited from integration in an SDN framework [65], to achieve the scalability required for operating machine-learning-based network anomaly detection in a cloud DCN.

2.5 Challenges of SDN in DCNs Selected as the Focus of this Work

Having studied the state of the art for SDN in DCNs, three open research challenges were identified and are presented below.

2.5.1 Challenge 1: Recording and Reproducing the Past State of a Software-Defined DCN

The first challenge is how to accurately record the state of a software-defined DCN, and reproduce that state for the DCN operator. This has received some attention from other researchers. The most relevant contribution is OFRewind [66]. OFRewind records and replays OpenFlow control-plane traffic in a campus network. OpenFlow message recording is effected via a proxy between switches and controller(s). To replay the recorded messages, a target network must be provided by the network operator: either the original physical network, or an emulated network created by some means not integral to OFRewind. OFRewind guarantees to maintain the order of OpenFlow messages as they were recorded, however it does not record the actual timestamps.

While OFRewind was an important advance, it is deficient in several respects for use in a DCN environment. The imposition of a proxy is undesirable, as it introduces latency into the connection between controllers and switches. Furthermore, it is infeasible to take a DCN off-line to replay recorded control-plane messages. Neither is it reasonable to require a DCN operator to construct an emulated copy of their large, complex and dynamic network as a target for replayed control-plane messages. Not having accurate timestamps makes it impossible to reproduce the DCN state for a specific timepoint of interest to an operator. Lastly, control-plane recording with OFRewind is intended to be run over short periods, for the purpose of troubleshooting and debugging the configuration of the forwarding devices on a

campus network. It is not intended or designed to be run over an extended period of weeks, months or years that a DCN operates, or at the scale of a multi-tenant data centre.

For contributions from other published research papers, the methods have mostly been to either sample data-plane packets, or simply gather statistics, for example Planck [50], and SketchVisor[67]. The collected data, in these instances, does not describe the topology of the network, or reveal the forwarding logic of network devices. NetSight [68] is of particular note, recording complete packet histories for debugging purposes. A packet history is a record of the path followed by a packet during its lifetime. The packet history could be used to (at least partially) infer the topology and forwarding logic of a network, if collected for all packets. Of course, collecting all data-plane packets is not feasible for anything more than a short interval on networks larger than a few devices. Furthermore, a drawback to NetSight is that it is not passive, requiring extra work to be done by switches, and imposes a proxy between switches and controllers.

Several works have been published that employ control-plane monitoring of OpenFlow networks. In common with data-plane tools, though, most of these (for example OFMon [69], OpenNetMon [70], OpenTM [71]) gather statistics, rather than recording control-plane messages. Logentries [72] collects both data-plane and control-plane log data for central storage. However, it is used for event logs such as those recorded by controllers and switches as part of their normal operation, rather than the full control-plane interaction between controllers and switches.

Most production-quality SDN controllers do not offer a built-in facility to record control-plane messages, with the exception of Floodlight [73], through its PacketStreamer module. With this module, OpenFlow messages can be filtered (for example by OpenFlow field values) before being streamed via a brokered message service to a waiting client. PacketStreamer is a Floodlight-specific module. ForenGuard [74] is a Floodlight-based system that logs and monitors OpenFlow

messages to identify root causes of forwarding problems in an SDN network. ForenGuard does not make use of the PacketStreamer module. The ONOS [75] controller offers statistics, through its CPMan app. Similarly, OpenDaylight [76] produces statistics via its Time Series Data Repository project. Note that each of these initiatives is controller-specific.

An opportunity was identified to devise a passive, controller-independent, scalable approach to logging SDN control-plane messages at the southbound interface (between controllers and switches) in a DCN, gathering sufficient detail over long durations to provide an accurate reproduction of the network topology and configuration for the DCN operator, for any point in the monitored period. Chapters 3 and 4 document that approach.

2.5.2 Challenge 2: Model Querying to Provide Deeper Insight into the Past State and Behaviour of a Software-Defined DCN

Knowledge modelling [77] is the process of synthesizing expert knowledge into a model that can be computationally interpreted. While there are different methods of knowledge modelling, the most used approach is ontology construction. Ontological models have been created by other researchers working to improve the task network management in SDNs.

ReasoNet [78] is a Network Operating System (NOS) architecture that enables SDN controller applications to take advantage of semantic web technologies. It is also a component that integrates with an SDN controller and uses an ontological model to validate requests from controller applications before applying them to the network. The researchers developed an OpenFlow ontology, claimed to be the first of its type. The ontology contains concepts and properties to represent the current state of an OpenFlow network. Six types of OpenFlow messages are collected from the SDN controller with which ReasoNet is integrated. The ontology is populated with those messages, and network state information is compiled, including switch and link status, topological data, flow table contents and flow statistics.

Unfortunately, the ReasoNet OpenFlow ontology has not been made publicly available for inspection, re-use or extension. However, in their paper the researchers make no mention of a temporal dimension to the ontology, or any application to providing insight into the historical operation of the network. While it is stated that the architecture allows ReasoNet to be integrated with any SDN controller, clearly any implementation is controller-specific because of the integration that is required. Furthermore, while the issue of scalability is addressed via a performance comparison with ONOS, the topologies on which it was tested were not of the scale of a DCN. Lastly, the focus of ReasoNet is controller application development and verification, rather than answering operator queries.

In B2D2 [79], machine learning is applied to data collected using the northbound interface of an SDN controller, to identify symptoms and causes of faults. Knowledge about a telecommunications network is encoded in their SDNDL (SDN Description Language) ontology. This ontology contains concepts for components of an OpenFlow network, such as controllers, switches, flow tables and flow rules. SDNDL can represent the data plane of a network, and the control plane from the point of view of the controller. Using SDNDL in combination with other components of a knowledge model, B2D2 identifies symptoms to diagnose possible faults that could cause those symptoms, and generates reports detailing the faults and the reasoning that has led to their diagnosis. The researchers have made the SDNDL ontology publicly available.

In focussing on the northbound controller interface, the approach in B2D2 misses out on the detail of control-plane interactions. The control-plane messages that are not represented in SDNDL are a rich source of information about events on a network. In particular, accurate timestamping of events will not be possible if only the northbound (rather than southbound) controller interface is monitored. The less granular timing information that is available to B2D2 is likely acceptable on telecommunications network (or WAN) where the configuration is relatively static, but is not sufficient in the dynamic environment of a multi-tenant DCN. The only

temporal element found in SDNDL is a 'lastSeen' property, limiting the temporal reasoning, comparison and inference that may be performed.

Ontological models may be used to represent complex temporal data and relationships. In the discipline of Computer Science, the World Wide Web Consortium (W3C) has taken a lead in developing relevant standards and providing guidance on their use [80].

Temporal logic has been applied in the context of SDN using non-ontological approaches. For example, in [81] extended automata are used to simulate the behaviour of SDN components, and their interactions are checked for potential race conditions in the execution of controller application requests of an SDN controller. Temporal properties of the interactions are expressed as Linear Temporal Logic formulas, which are verified by a Spin model checker. The extended automata are based on the OpenFlow specification, and a formal description of an SDN controller.

The idea was conceived of creating a logical model of a software-defined DCN, incorporating temporal and topological concepts. The model, populated with information extracted from recorded control-plane data, would represent the state and state changes of the DCN over the full duration of the recorded data. The purpose of this rich resource is to serve as the target of a diverse set of operator queries. Chapters 5 and 6 describe the work undertaken to crystalise this approach.

2.5.3 Challenge 3: Addressing Control-Plane Performance Issues in a Software-Defined DCN

Control-plane performance was recognised as an issue for SDN from its early days. Suspicions about the part that switch implementations of OpenFlow played in the performance issue were confirmed and qualified by work such as [82, 83]. It was clear that the limitations of hardware switch CPUs and the bottleneck of updating TCAM memory were underestimated in the design of OpenFlow, where it was assumed that flow-rule modification requests would be handled promptly by switches.

Creative solutions were proposed, sharing an approach of proactive configuration of the required set of rules. Since OpenFlow tables typically would not have the capacity for a rule for every path, the innovation in PAST [84] was to implement pre-configured per-address spanning trees in the layer-2 forwarding tables of switches, reducing the total number of rules required. Planck [50] added monitoring and control to PAST, and demonstrated those features being used to provide redundant paths and rerouting capabilities. Both use switches that specifically support programmatic access to the layer-2 forwarding tables (rather than just to the TCAM tables used by OpenFlow). Also, both provide paths between all pairs of hosts, which is commonly the policy applied in a campus network. This would not be acceptable in a multi-tenant DCN where paths should only be configured to match the security policies.

PARIS [61] forwards traffic through pre-configured summary routes to the core, where Equal Cost Multipath (ECMP) routing then decides the route to edge switches that have a forwarding entry for every local virtual machine (VM). This requires a layer 3 addressing scheme that lends itself to summarisation – not a reasonable assumption in a multi-tenant DCN. Although PARIS describes an enhancement to support multi-tenancy, it depends on MPLS or VLAN tags – increasing the complexity and reducing the flexibility of the solution.

Devoflow [85] identified the cause of flow-setup latency as a bottleneck between the data-plane and the control-plane, and proposed more autonomy in switches with aggressive use of wildcard OpenFlow rules, although security-sensitive flows must still be handled centrally. The approach of giving switches more responsibility is also taken in [55]. Both [85] and [55] require modified switch hardware.

Monocle [86] verifies that flow-rule modifications have been applied at the data plane, delaying confirmation from OpenFlow switches to the controller until data-plane probes have shown the modified rules to be active. It does not make any effort to reduce flow-setup time, rather it reflects the true flow-setup time.

In a multi-tenant OpenFlow DCN, it is not feasible to proactively install all rules prior to any tenant flows arriving. Flow-rules will need to be modified as tenants arrive and leave, or reduce or increase their set of leased resources. However, nor is it acceptable to have long flow-rule modification delays. Any approach should be applicable using commodity hardware.

Reducing the gap between a flow-rule modification request and confirmation that the modification has occurred increases the accuracy and usefulness of a historical record of these events.

This informed the third challenge, which was to find an approach to reducing control-plane latency (in particular, flow-rule modification times), to support flow-rule installation in response to changes in tenancy or tenant requirements in a multi-tenant DCN. The resultant improvement to the accuracy of a historical record will clearly improve the accuracy of work that leverages that record. The selected approach is described in Chapter 7.

2.6 Conclusion

Three challenges have been selected from the open research areas of the management of a multi-tenant software-defined DCN.

There is a requirement to devise a passive, controller-independent, scalable approach to SDN control-plane monitoring in a DCN, gathering sufficient detail over long durations to provide an accurate reproduction of the network topology and configuration for the DCN operator, for any point in the monitored period.

In order to provide a target for key operator queries, create a logical model of a software-defined DCN. The model must incorporate temporal and topological concepts, and represent the changing state of the DCN over the full observation period.

To increase the accuracy with which the recorded state of the network at a given time corresponds to the operational network, formulate an approach to reduce

SDN control-plane latency and its resultant effect on timely flow-rule installation in a multi-tenant DCN.

The following chapters set out how the challenges have been addressed, and the novel contributions of the selected approaches.

3 Reproduction of the Topology and State of a Software-Defined DCN from a Control-Plane Message History

3.1 Introduction

Chapter 2 reviewed other researchers' work on the application of software-defined networking techniques in data centre networks, and described the state of the art. The first novel contribution of this work, presented below, is a method to create a functioning, interactive reproduction of a Data Centre Network (DCN) in a historical state, for a timepoint of the DCN operator's choosing, within a time range for which the exchange of control-plane messages on the network have been recorded.

Section 3.2 explains why operators need a way to view the past state and topology of a DCN. The research challenges are set out in section 3.3, and section 3.4 describes the architecture that was devised, with the key requirements identified to address the research challenges. An implementation was constructed based on the architectural requirements, as described in section 3.5. The implementation was evaluated, with the test environment, testing methods, and results discussed in section 3.6.

3.2 Motivation

Multi-tenant DCNs operate under detailed, potentially complex and frequently-updated policies. The detail and complexity are present for several reasons. Each tenant will have their own requirements for reachability between their hosts and services. A tenant's resources must be secured from access by others; however, access must still be provided to shared services such as DNS and path(s) to and from the Internet. The frequent updates are caused by new tenants arriving in the data centre, and existing tenants changing their requirements, or leaving the data centre. The policies define what host devices are allowed to communicate with each other, and where traffic (dependent on packet attributes) is allowed to go.

In an OpenFlow-based Software-Defined Network (SDN), a policy update will result in changes to OpenFlow rules, which represent part of the network state. Even without policy updates, flow rules may be added, modified and removed by an SDN controller as it attempts to perform load-balancing or recovery from a failed link or DCN switch. When a problem occurs, by the time the DCN operator has an opportunity to address it, the network state most likely has changed, and it may never be possible to identify the root cause of the problem, or the conditions under which it occurred, without a means (e.g. network tools) to provide a view of the network state at and around the time at which the problem occurred.

Clearly, DCN operators do not want tools that will increase their workload by requiring constant monitoring, or that require changes to switch or controller configurations to accommodate the tools. Nor do they want tools that consume a significant quantity of resources. The opportunity addressed by this work is to allow the reproduction of a DCN in historical states. The proposed solution, presented in this chapter, is to unobtrusively gather control-plane information that is already present in the network, but that is usually not recorded. Using a log of this information, the state of the network as it was at any time can be reproduced, to provide a platform for investigation and problem-solving using tools with which the DCN operator is already familiar. Examples of such tools are *ping* for reachability testing and *ovs-ofctl* for dumping flow-table contents from switches.

The decision was taken to use OpenFlow messages as the basis for a historical log as OpenFlow is the mostly broadly-used open standard SDN protocol, but the approach applies to, and the implementation could be adapted for, any control-plane protocol used between SDN controllers and network devices. NetConf, OVSDB, and BGP are examples of other control-plane protocols, and they may be used in conjunction with each other. It should be noted that other control-plane protocols may not contain the same level of detail as OpenFlow, and may need to be supplemented with information from other sources to achieve an equivalent quality of reproduction.

Using only OpenFlow messages brings its own limitations: it does not capture all data plane activity or packet details, and inactive hosts are not discovered. Clearly, however, there must be a trade-off between the quantity of data to be recorded, and the utility of that data. A further issue is that the OpenFlow protocol has no explicit support for multi-tenant environments. Tenant-related policy is defined at a higher level, for example in an app running on the SDN controller, and the OpenFlow rules are subsequently configured to be compatible with the policy. It is expected that the DCN operator will have defined the policy, and will have ways of identifying what hosts are associated with which tenant, e.g. based on IP address. OpenFlow messages contain sufficient detail to meet the goals of this work, but if the goals were expanded in future, then information from other network management sources could be combined with the OpenFlow messages to target those goals.

3.3 Research Challenges

The research challenges to be addressed were formulated as:

- What approach should be taken for passive, accurate recording of the history of OpenFlow messages exchanged on a DCN?
- How could a DCN be accurately reproduced in a resource-efficient manner?

It was determined that addressing the challenges required the production of a set of tools structured as a system to record network configuration changes, to allow the DCN operator to view and operate the network in its historical state, and to do so without creating extra overhead for the operator. SDN makes the recording of configuration changes possible because the control-plane is separated from the data-plane. Moreover, in a DCN the control-plane is centralised, providing a single point at which data can be collected. OpenFlow control-plane messages contain the required information about configuration changes. All control-plane messages are accessible for collection, because the entire DCN environment is under the complete control of the DCN operator.

For the recording mechanism to be ‘passive’, it needed to be controller- (and switch-) independent, and not introduce extra latency or interruption to switch/controller communication. To be ‘accurate’, it needed to capture all control-plane packets under normal circumstances, or under abnormal circumstances indicate if any packets were not captured. Due to the potential scale of a DCN, a high control-plane packet arrival rate is to be expected – however, abnormal circumstances are when control-plane packets arrive at a faster rate than the SDN controller itself can process them.

For TCP connections between switch and controller, transport-layer flow-control and retransmission of lost packets should mitigate against those abnormal circumstances. However, OpenFlow also allows UDP connections between switch and controller, in which case there is no transport layer flow-control or retransmission, and OpenFlow itself does not include such mechanisms. Hence the need to detect when packets have been missed.

All control-plane messages (extracted from control-plane packets) needed to be recorded. Given the potentially high packet arrival rate, and the fact that the history would need to be recorded for weeks, months, or even years, scalable storage was required that could be constantly appended to. It must also be query-able to allow the messages required for network reproduction to be extracted.

Reproduction of a DCN in a historical state needed to be achievable for any timepoint (covered by the historical log) chosen by the operator. The 'state' of the DCN at a single point in time is defined as the topology (hosts, switches, links) and the forwarding device configuration. Clearly, for the reproduced network to be accurate, the topology must be the same as the original network at the selected timepoint in history, and the forwarding-device configuration must be the same. In OpenFlow terms, a forwarding device’s flow-table contents represent the device’s configuration.

For a production data centre, clearly it is infeasible to expect the operator to take the DCN offline to revert its topology and configuration to a historical state,

therefore some external hardware and software is required. However, it is also unrealistic to expect the DCN to be fully duplicated in hardware. In considering alternative solutions, it was a key priority to minimise the extra hardware required.

3.4 Requirements and Architecture of the Proposed Solution

A high-level architecture diagram presented in Figure 3.1 illustrates the major components required to meet the research challenges. Those components are:

- *OpenFlow message logger*. This component must un-intrusively capture and log OpenFlow messages. All OpenFlow messages exchanged between controller and switches should be captured. The logger must detect if any messages have been missed, log the event, and continue capturing and logging.

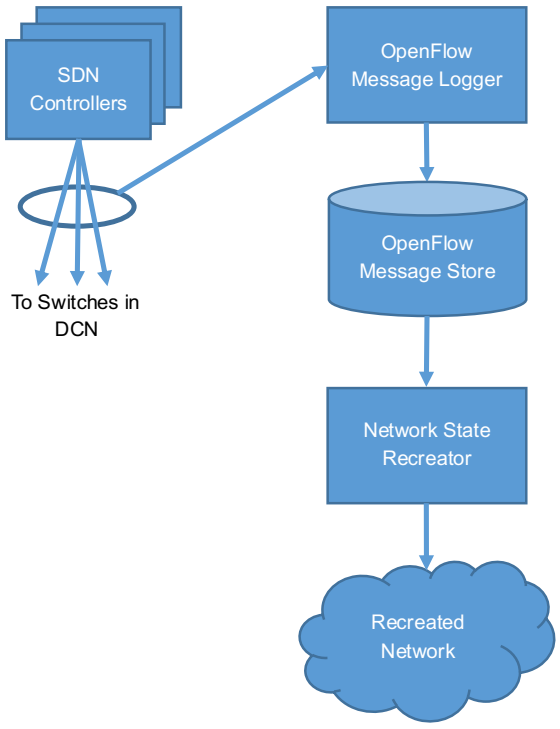


Figure 3.1: Architecture Diagram

- *OpenFlow message store*. Every captured OpenFlow message must be stored in full, with the associated timestamp. The store must be searchable by timestamp, by date range, and by OpenFlow message attribute.
- *Network state recreator*. Using messages from the OpenFlow message log, this component must reproduce the topology and flow-table state of the DCN from which OpenFlow messages were logged, for a specified time and date.

The components are described in more detail in the next sub-sections.

3.4.1 DCN Control-Plane Message Capture and Storage

The goals of the message capture and storage components reflect the nature of the DCN environment in which the system was to run:

- Scalability
- Controller- and switch-independence
- Passive operation

A DCN contains potentially hundreds of switches, connecting thousands of servers with each other, with the Internet, and possibly with other data centres. The message capture component is responsible for capturing OpenFlow messages exchanged between switches and controllers. Controllers address scalability by clustering – running multiple instances of the controller and partitioning the set of switches so that each controller instance has only to deal with a manageable number of them. Furthermore, the number of interactions between controller and switches is often reduced by having the controller proactively configure most flow-rules rather than doing so reactively. The proactive approach means that the controller does not need to be consulted each time a new flow arrives at a switch, only when a flow arrives for which it does not have a matching proactively configured rule. In a multi-tenant DCN, rules are likely to be proactively installed or removed when a new tenant arrives, when an existing tenant leaves or changes their set of leased resources, or when an existing tenant's resources must be

redistributed within the DCN. The capture component was designed for scalability, accommodating (and taking advantage of) features of controllers intended to improve their scalability.

Logging is required to operate independently of which controller is in use within the DCN. This ruled out the use of a controller-based application to log OpenFlow messages, or use of information outside of the control-plane, for example through the use of controller APIs. Logging should not depend on which brand or models of switches are in use on the DCN. The system implementation is restricted to an environment where all controllers and switches use OpenFlow, however there are different versions of OpenFlow, and not all features are implemented universally.

To address the requirement for passive operation, logging should not require any change to the setup or configuration of the controller or switches, and should have minimal impact on their operation. For example, a proxy-based design would have required controller and switch configurations to route OpenFlow messages through the proxy, increasing latency and risking the proxy becoming a bottleneck as the network scales.

3.4.2 Reproduction of DCN Topology and State

The goals of the DCN topology and state reproduction component were as follows:

- Resource efficiency
- Accurate topology
- Accurate flow-table state

As stated in section 3.3, a DCN itself could not reasonably be taken offline to revert its topology and configuration back to a historic state. Nor is it feasible to expect a complete set of duplicate equipment to be available as a target for network state reproduction. To minimise hardware requirements, it was decided that DCN reproduction would be in an emulated environment. The decision meant that no extra hardware DCN switches would be required, and consequently no physical connections to establish and maintain between hardware switches. Server

resources would be required, sized based on the DCN to be emulated and the emulation environment chosen for implementation.

Clearly an emulated DCN will not match its physical equivalent in terms of data-plane performance. However, the purpose of the reproduced network is not performance testing, it is to faithfully reproduce the state and topology of the original DCN.

The reproduced network must match the original DCN in numbers of SDN controllers, switches, hosts, and the links between switches and hosts. This topology information must be gathered solely from control-plane messages. The topology of the DCN can change over time, due to switches being replaced because of failure, for example, or because of links being added or removed. The reproduced topology must be temporally accurate, that is: it must faithfully reflect the topology of the DCN as it was at the timepoint selected by the operator.

Flow-table state changes more frequently than the topology. Specific control-plane messages prompt changes in flow-table rules, others notify the controller of rules being removed from flow-tables. The timestamp of each message can be used to ensure temporal accuracy of reproduced flow-table contents.

3.5 System Implementation

A system to reproduce the topology and state of a software-defined DCN was implemented, as illustrated in Figure 3.2. The system was developed to meet the design goals identified in the sections above. Off-the-shelf components were used in the implementation where possible, chosen for being widely used (therefore, well-tested) and for scalability. Other components were bespoke.

Control-plane message capture and storage is achieved by the Logging Subsystem. An emulated reproduction of a DCN is generated from the stored messages via the Network State Recreation Subsystem. The two subsystems share a database – the Logging Subsystem continuously writes to the database while control-plane messages are being recorded; the Network State Recreation Subsystem reads from

the database whenever a DCN operator requests reproduction of their network. Both subsystems are now described.

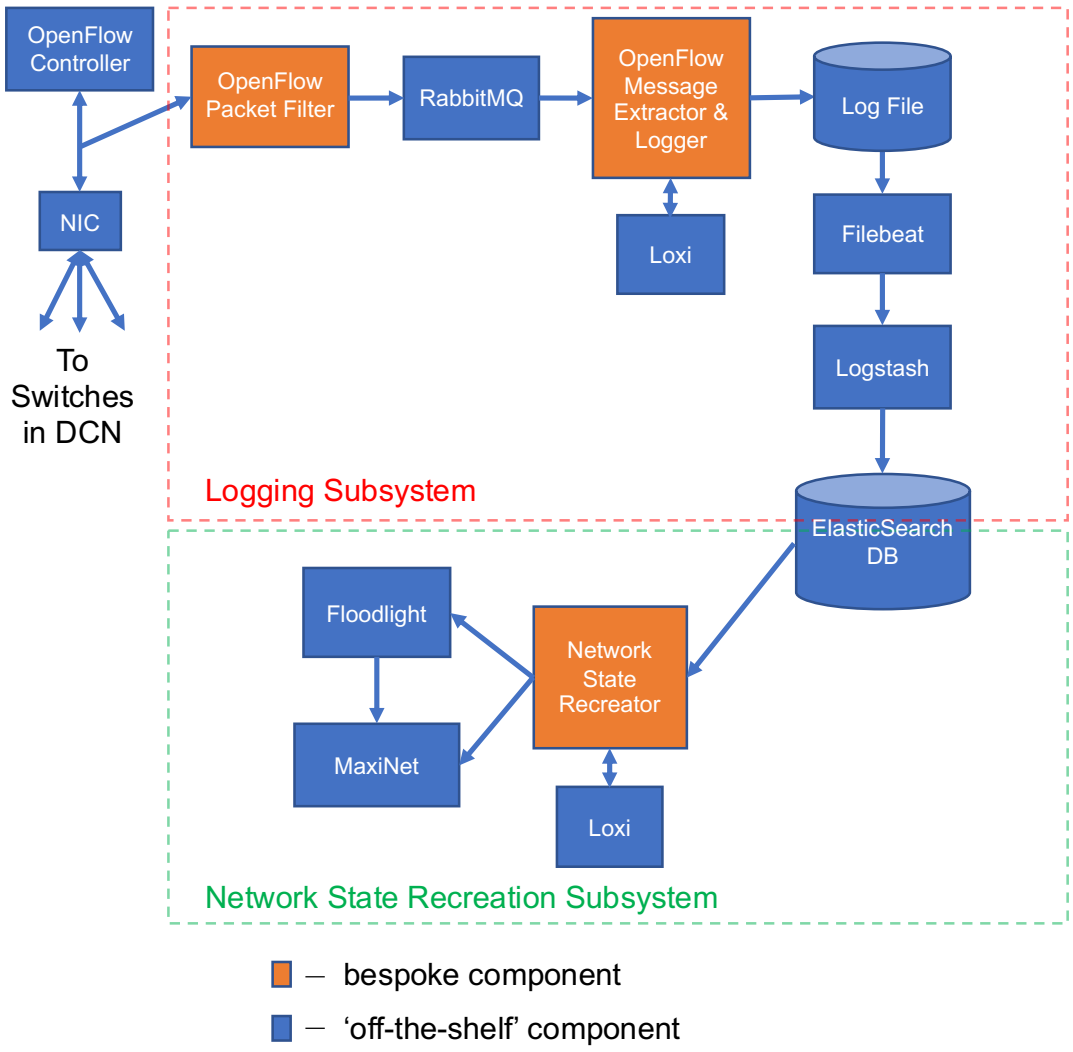


Figure 3.2: Implementation Diagram

3.5.1 Logging Subsystem

The OpenFlow Packet Filter uses the libpcap packet-capture library [87] to passively capture packets containing OpenFlow messages, achieving the goal of controller- and switch-independence. libpcap grabs packets directly from the network interface adapter, i.e. as layer 2 frames. Captured frames are tagged with their capture time, and enqueued to a RabbitMQ [88] message queue for subsequent parsing and further processing by the OpenFlow Message Extractor & Logger.

RabbitMQ is a widely-used open-source message broker, for relaying messages between processes or applications. It has a reputation for reliability and scalability.

Separating capturing from logging has several benefits: Only the OpenFlow Packet Filter needs to run on the same physical or virtual host as the OpenFlow controller, minimising use of resources that might be needed by the controller and supporting the goal of passive operation. The clustered controller scenario mentioned in section 3.4.1 can be leveraged for the goal of scalability by deploying an OpenFlow Packet Filter instance for each controller instance. The implementation accommodates a single controller or a cluster of controllers running on a single physical server. Extending the solution to support distributed controllers would require time-synchronisation between servers to allow accurate merging of timestamped data recorded from those servers, for example by using the Precision Time Protocol (PTP) [89]. Even with a single controller instance, measurements in a DCN from PTPmesh [90] could be used to adjust timestamps to more accurately pinpoint the time at which captured packets were sent if that was a requirement.

A DCN operator will be familiar with the importance of timestamps for fault management and event correlation with traditional network management tools and may already be running a clock-synchronisation service such as Network Time Protocol (NTP) [91]. NTP, however, is not precise enough for the application described here, only providing millisecond accuracy rather than the sub-microsecond accuracy provided by PTP.

Recent revisions of the OpenFlow Switch Specification (e.g.[26]) state that “The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP.” The authors of [21] provide reasons for not using TLS in a DCN. The implementation described here currently assumes TLS is not in use, however it could be extended if necessary to decrypt captured packets using the DCN TLS keys, as can be done with Wireshark, for example. Decryption would be best added at the point of message extraction and logging, rather than at the point of capture, thus keeping overhead minimal when capturing packets.

With a passive capture approach there is a risk of packets being missed at very busy times, with no option to request packet retransmission, or to exercise flow-control. Since a missed OpenFlow packet could seriously impact accurate reproduction of network state, the capture filter code was stress-tested, as described in section 3.6.4. While the results instil confidence in the packet filter, nevertheless for a TCP connection it detects if a packet has been missed and the event is logged in the message log to subsequently notify the DCN operator. The mechanism for detecting missed packets monitors TCP sequence numbers and payload sizes, accounting for every octet of a connection between the initial SYN packet and final FIN or RST packet.

Clearly, other implementation options for capturing control-plane packets exist – a proxy between the controller and switches, or a controller app or extension, or mirroring packets from switches to a central collection point. However, none of those alternatives would meet the goals of scalability, controller-independence and passive operation set out in section 3.4.1.

The OpenFlow Message Extractor & Logger reassembles the TCP stream for each switch-controller connection from the captured packets, and extracts OpenFlow messages from the streams. Since sequence numbers and payload sizes are monitored, retransmissions are detected and duplication of messages is avoided. Each message is parsed using Loxi [92] before being serialised and logged to a text-encoded log file. Loxi is an open-source OpenFlow protocol library. Using it to parse captured binary OpenFlow messages ensures that the messages are valid OpenFlow messages before they are logged, and to serialise them (converting Loxi message objects to YAML) for storage in an easily searchable form. Since Loxi supports OpenFlow versions from 1.0 to 1.5, use of it provides OpenFlow version-independence.

A single instance of the OpenFlow Message Extractor & Logger process can be used, or as many as one instance per switch, for the purpose of scalability. Each instance emits messages to a separate log file. A new line added to a log file by the Extractor

& Logger describes a single OpenFlow message. Filebeat ships each line to LogStash for processing and storage in ElasticSearch. Filebeat, Logstash and Elasticsearch are members of the open-source Elastic Stack [93] family of projects. The components operate in sequence, as shown in the Logging Subsystem of Figure 3.2, and described briefly as follows: Filebeat 'tails' log files: it monitors the files to extract any newly appended lines of log data, which it then 'ships' to Logstash for processing before their storage in the Elasticsearch full-text search engine. Elasticsearch provides fast, scalable storage, and allows fast retrieval of OpenFlow messages by timestamp.

Listing 3.1 shows an example of a single line from the log file. The line contains tab-separated values for these pieces of information regarding a captured OpenFlow message:

1) Timestamp - `2021-10-30T16:47:00.090811`,
2) Source MAC address - `08:00:27:09:cc:18`,
3) Destination MAC address - `08:00:27:47:00:e2`,
4) Source IP address - `192.168.56.100`,
5) Destination IP address - `192.168.56.102`,
6) Source port number - `6653` (this well-known port number is reserved for controllers),
7) Destination port number - `39900`,
8) The version of the OpenFlow header - `5` (identifying the captured OpenFlow packet as being a version 1.4 message),
9) The OpenFlow message type - `OFPT_FLOW_MOD`,
10) A serialization of the Loxi Python object representing the captured message – the rest of the data on the line.

All lines of the log file follow the same format. The line shown is less than 1,000 characters in length. Other lines lengths vary significantly. For example, a line with details of a *Hello* packet is less than 200 characters long, while a line containing Multipart-Reply (with statistics for each port of a switch) has been counted at almost 43,000 characters in the log files collected for this work.

The example in Listing 3.1 contains details of a *FlowMod* message sent from a controller to a switch instructing the switch to add a flow-rule to its flowtable, with rule priority of 10. The rule will match against packets that arrive in on switchport 17 of the switch and that have source MAC address of 02:d1:c2:00:00:05 and destination MAC address of b2:e0:22:f3:83:97. Note that these MAC addresses are represented as decimal byte values in the serialised object. Packets matching the rule will be forwarded through switchport 19.

A line from the log file is written to the ElasticSearch database with key index *timestamp*, and other field names *src_mac*, *dst_mac*, *src_ip*, *dst_ip*, *src_port*, *dst_port*, *of_version*, *msg_type*, and *msg_obj*. Other meta-data is also recorded by LogStash, including the name of and directory path to the log file from which the line was extracted by FileBeat, and the hostname of the device on which the log file was stored. The meta-data is not currently used by any component of the work described here but could be useful in the future for troubleshooting issues with the Logging Subsystem or with the DCN from which control-plane message were being logged.

```
2021-10-30T16:47:00.090811    08:00:27:09:cc:18 08:00:27:47:00:e2
      192.168.56.100    192.168.56.102    6653  39900 5
      OFPT_FLOW_MOD     !!python/object:loxi.of14.message.flow_add
{"buffer_id": ! "4294967295", "cookie": ! "32088151586711026",
"cookie_mask": ! "0", "flags": ! "1", "hard_timeout": ! "0",
"idle_timeout": ! "0", "importance": ! "0", "instructions":
[!!python/object:loxi.of14.instruction.apply_actions {"actions":
[!!python/object:loxi.of14.action.output {"max_len": ! "0", "port":
! "19"}]}], "match": !!python/object:loxi.of14.common.match_v3
{"oxm_list": [!!python/object:loxi.of14.oxm.in_port {"value": !
"17"}, !!python/object:loxi.of14.oxm.eth_dst {"value": [! "178", !
"224", ! "34", ! "243", ! "131", ! "151"]},
!!python/object:loxi.of14.oxm.eth_src {"value": [! "2", ! "209", !
"194", ! "0", ! "0", ! "5"]}]}, "out_group": ! "4294967295",
"out_port": ! "4294967295", "priority": ! "10", "table_id": ! "0",
"xid": ! "103"}
```

Listing 3.1: Example of Log File Entry

The most challenging part of the Logging Subsystem implementation was the capture filter. Initially, the component was built on the BPF Compiler Collection (BCC) [94], however the Python-based implementation was too slow to capture packets at the required rates. The capture filter was re-written in C, making use of libpcap. A large frame buffer was defined, to allow jumbo frames to be captured. The capture loop reads a packet from the frame buffer, timestamps it and sends it to RabbitMQ before checking for the next packet. The operation was tested and the code refined to achieve high packet capture rates, as quantified in section 3.6.4 below.

3.5.2 Network State Recreation Subsystem

A data centre operator uses the Network State Recreator to reproduce a DCN for a specific date and time covered by the set of OpenFlow messages captured from that network and logged to an Elasticsearch database. The Network State Recreator queries the Elasticsearch DB for OpenFlow messages with timestamps earlier than or matching the specified time and date. See Table 3.1 for the information extracted from specific types of OpenFlow messages, which is then used to compile the state of the DCN as it was at the desired time point.

OpenFlow Message Type	Encapsulated Information
FEATURES_REPLY	Switch DPID
MULTIPART_REPLY	Switch ports
PACKET_IN	Inter-switch links; host-switch links
FLOW_MOD	Flow-rule additions / modifications / deletions

Table 3.1: Key Information in OpenFlow Messages

An example of each of the listed message types is shown in Listing 3.2. The examples are extracted from a tcpdump packet capture performed in parallel with an experiment. Information extracted from messages for use in the compilation of DCN state is highlighted.

Example of a *FeaturesReply* message:

IP address and TCP/UDP port used to identify switch

```
23:02:58.066 192.168.56.101.49530 > 192.168.56.100.6653:
      OFPT_FEATURES_REPLY (OF1.4) (xid=0xfffffffe):
      dpid:0000047db6000004  n_tables:254, n_buffers:0
      capabilities: FLOW_STATS TABLE_STATS PORT_STATS GROUP_STATS
      QUEUE_STATS BUNDLES
```

OpenFlow identifier for switch

Example of a *MultipartReply* message containing port descriptions:

Identifies switch sending port descriptions

```
23:03:03.157 192.168.56.103.45806 > 192.168.56.100.6653:
      OFPST_PORT_DESC reply (OF1.4) (xid=0x0):
       1(mn_tun20): addr:66:e6:9b:49:c2:4a            config:      0
          state: LIVE        speed: 0 Mbps now, 0 Mbps max
       2(ss4-eth2): addr:66:fd:51:e4:ae:16            config:      0
          state: LIVE        current:    10GB-FD COPPER
          speed: 10000 Mbps now, 0 Mbps max
       3(ss4-eth3): addr:3a:7a:05:10:c1:06            config:      0
          state: LIVE        current:    10GB-FD COPPER
          speed: 10000 Mbps now, 0 Mbps max
       4(mn_tun4): addr:ca:96:e1:36:db:a2      config:      0
          state: LIVE        speed: 0 Mbps now, 0 Mbps max
       5(mn_tun9): addr:4e:a8:ae:51:3a:bb      config:      0
          state: LIVE        speed: 0 Mbps now, 0 Mbps max
       6(mn_tun2): addr:82:21:89:2a:d4:60      config:      0
          state: LIVE        speed: 0 Mbps now, 0 Mbps max
        LOCAL(ss4): addr:86:3f:a6:39:8b:48      config:      PORT_DOWN
          state: LINK_DOWN         speed: 0 Mbps now, 0 Mbps max
```

Port numbers on switch

Identifies the switch that received the encapsulated packet

Example of a *PacketIn* message with topology discovery information:

Port on which the packet was received

```
23:03:02.184 192.168.56.101.49532 > 192.168.56.100.6653:
      OFPT_PACKET_IN (OF1.4) (xid=0x0): cookie=0x10000d9e5112d
      total_len=81 in_port=4 (via action) data_len=81 (unbuffered)
      vlan_tci=0x0000,dl_src=02:eb:9f:67:c9:42,
      dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
      out_port=26 dpid=of:000004bbc9700003
```

LLDP Packet Type

Identifies sending switch and port*

Example of a *FlowMod* message:

Identifies switch to which *FlowMod* request was sent

```
23:03:42.327 192.168.56.100.6653 > 192.168.56.101.49536:
      OFPT_FLOW_MOD (OF1.4) (xid=0x1c8): ADD
      priority=10,in_port=11,dl_src=00:c8:5d:00:00:05,
      dl_dst=40:e2:fe:00:00:05 cookie:0x720000b7878f8c
      send_flow_rem actions=output:2
```

Match criteria

Port through which matched packets should be forwarded

- Packet details extracted from a capture file using Open vSwitch's ovs-ofctl tool;
- Additional information (*) from WireShark analysis.

Listing 3.2: Examples of OpenFlow packet types and encapsulated information

The operator may not initially know the appropriate time point for which to reproduce the DCN. The time point might relate to some event of interest, for example when a specific host (or pair of hosts) is present. The information to identify the timepoint is in the log of control plane messages in ElasticSearch, but not in a form that lends itself to extraction with a simple query. A more sophisticated and usable approach to querying the information in the log is detailed in Chapters 5 and 6.

For the selected timepoint, the DCN topology is reproduced with MaxiNet [95], a distributed extension of the Mininet [96] network emulator. Mininet has been tested [97] in creating networks of up to 1024 switches and 1500 hosts, however it is limited to running on a single host. MaxiNet allows the creation of very large networks (towards the goal of scalability) in emulated form, and the emulated networks can be distributed across multiple hosts.

To populate switch flow-tables, a customised build of Floodlight [73] is used. Floodlight is a well-established OpenFlow controller, although it has not achieved the same level of commercial adoption as others, such as ONOS, Open DayLight or Ryu. For this application, Floodlight was modified to act purely as a delivery mechanism for flow-rules captured from the original DCN. Any OpenFlow controller could be adapted for this purpose, but Floodlight was a natural choice as it uses the same Loxi protocol library used in the Logging Subsystem.

Using emulation to reproduce the DCN incurs a compromise: for example, the software switches in the emulated, reproduced network do not have the same performance characteristics as hardware switches. At the data plane, hardware switches outclass software switches forwarding ability – this is well known, and not a concern here since it is not a goal of this work to reproduce data plane traffic loads. Conversely, software switches can outperform hardware switches at the control plane – as hardware OpenFlow switches vary in how quickly they update their flow-tables under heavy load [83], although the high-end switches more likely to be used in DCNs perform best. While mimicking control plane performance is not

an immediate goal of this work, if a DCN operator required control plane performance to be matched then the Network State Recreation System could be extended to account for switch control plane performance – this would require a profile for each type of switch used. The issue of hardware switch control plane performance is addressed further in Chapter 7.

As stated earlier, it is unreasonable to require a DCN operator to take their network offline to return it to an earlier state, hence the reproduction of the network in emulated form. If, however, the operator chooses the former approach, or has replicated the DCN topology with physical hardware, the Network State Recreator could be adapted with minor changes to populate the flow tables on the physical switches.

3.6 Evaluation

The architectural goals of controller and switch independence and passive operation were addressed through the implementation, with the approach of capturing packets using a capture filter. Scalability was integral to the design also, with a capture filter instance for each controller instance, and separation of capture from message processing. Two questions remaining were: what the performance limits of the capture filter are; and how much storage is required for the message logs. Those questions are answered in sections 3.6.3 and 3.6.4 below.

For DCN topology and state reproduction, the goals were resource efficiency, accurate topology, and accurate flow-table state. Resource efficiency was achieved through use of emulation to reproduce the DCN. The implementation was tested to verify accuracy of the reproduced network's topology and flow-table state, as described below in sections 3.6.1 and 3.6.2.

3.6.1 Test Environment

A DCN topology was devised for which to test the system, illustrated in Figure 3.3. This network consisted of six racks of twenty hosts and one top-of-rack switch each. The six top-of-rack (also referred to as leaf) switches were connected in a folded-clos topology to six spine switches to create a spine-leaf network architecture, as is

commonly used in a DCN. As the scale of data centres has continuously increased over the years, there is no fixed definition of what constitutes a small, medium, or large data centre. However, several vendors and service companies [98-100] define a hyperscale data centre as having at least 5,000 servers, with potentially up to hundreds of thousands of servers. In that context, 120 servers represent a small data centre. Despite the small size of the test DCN topology, it is sufficient to demonstrate the functionality and accuracy of the implementation.

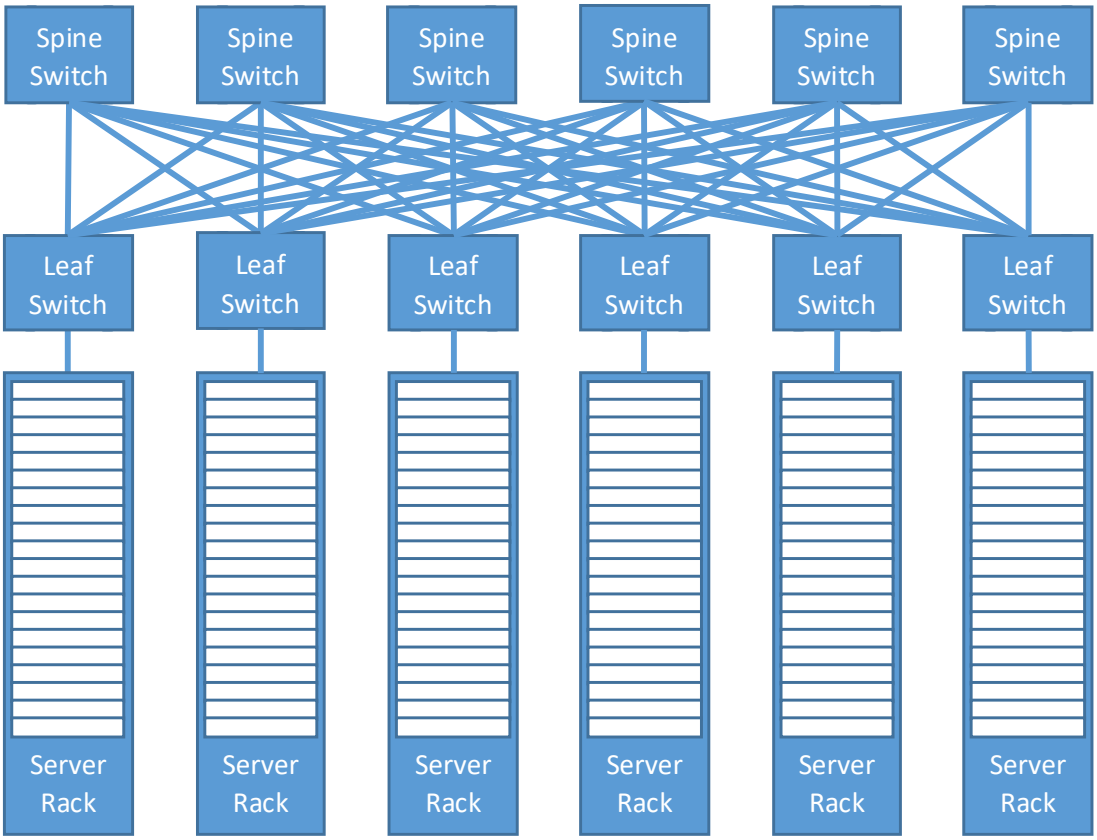


Figure 3.3: DCN Topology Diagram

The use of MaxiNet described so far has been to host the recreated network. However, for experimental testing for this work, MaxiNet was also used to initially host the original network – an emulated network designed to reflect the topology and conditions of a busy DCN. The OpenFlow control plane messages were captured in this environment, along with other information that could be used to

verify that the recreated network was accurate. While it would have been preferable to use a control plane message log from a production DCN, data centre operators are understandably reluctant to share such data. The limitation of using MaxiNet is that the switches are software switches rather than hardware switches whose performance characteristics would have resulted in timing differences to message exchanges. However, the functional operation of the protocol implementation is equivalent.

As illustrated in Figure 3.4, the test environment was hosted on three physical machines, connected via two hardware Ethernet switches. Three MaxiNet worker VMs were hosted on a single server (Cisco C200 M2 - Intel Xeon E5506, 2.13GHz, 8 cores, 32GB RAM). One MaxiNet worker VM could have supported the topologies tested in this work, however the figure of three was chosen to demonstrate the capability of the system for scaling horizontally. Further MaxiNet VMs could be added, and the VMs could be migrated to other server machines to take advantage of more hardware resources, if those were available.

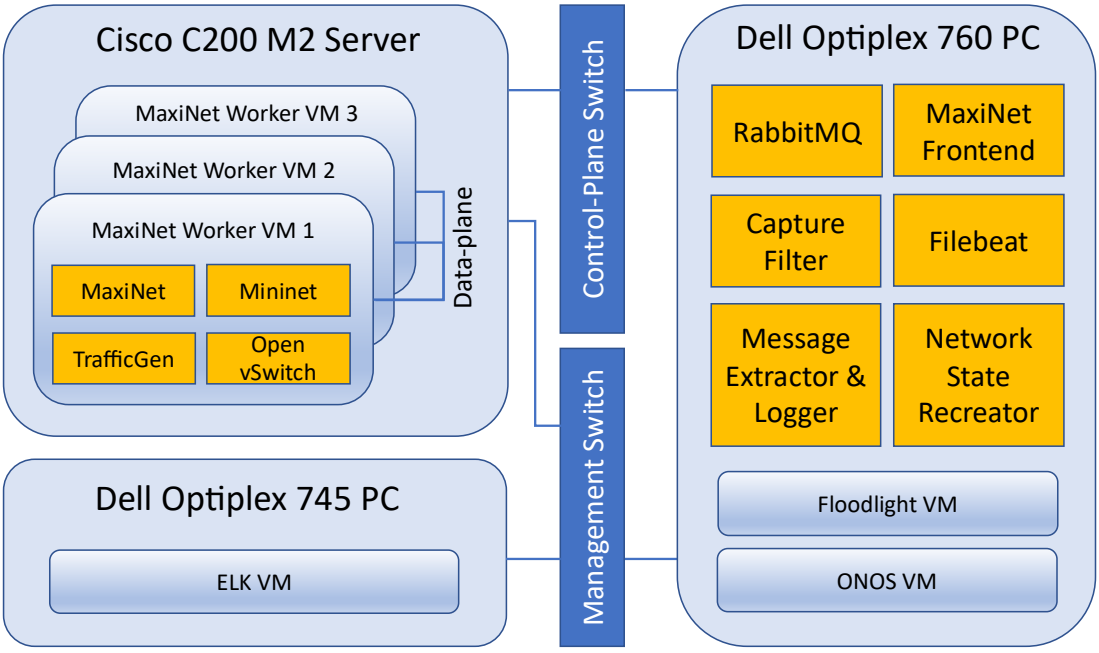


Figure 3.4: Testbed Diagram

The worker VMs ran Ubuntu Server 18.04, with MaxiNet 1.2, Mininet 2.2.2, and Open vSwitch [16] 2.9.0 installed. A PC (Intel Core 2 Quad Q9400, 2.66GHz, 4 cores, 8GB RAM) was used to control experiments, host a controller VM, run the MaxiNet FrontEnd server, RabbitMQ 3.7.8, and Filebeat 6.3.1. Depending on the stage of an experiment, it also ran the OpenFlow Packet Filter and OpenFlow Message Extractor & Logger, or the Network State Recreator. The third machine, a PC (Intel Core 2 6400 2.13GHz, 2 cores, 4GB RAM), hosted a database VM running Ubuntu Server 18.04, with Elasticsearch 6.4.3 and Logstash 6.4.3 installed. The three physical machines were connected to a Gigabit Ethernet switch to create a network used for control of experiments (the 'Management Switch'), and communication with the Elasticsearch database. Another Ethernet switch (the 'Control-Plane Switch) was used solely for the exchange of OpenFlow messages between controller and switches.

Clearly, the capture filter must run on the same physical machine as the SDN controller in the network from which control-plane messages are being logged. In the testbed, RabbitMQ, the Message Extractor & Logger, and Filebeat are on the same machine, but they could easily be moved to a different machine if more resources were required by the controller. In a clustered controller configuration, it is expected that each controller instance would run on a dedicated machine, in which case a separate capture filter instance would also run on each machine, sending captured packets to a single device running RabbitMQ and the other components. If necessary, RabbitMQ can be clustered, parallel Message Extractor & Logger process can be run, each writing a separate log file from which dedicated instances of Filebeat can ship log data to a shared ElasticSearch cluster. In short, the scalability that was described in the architecture earlier is supported by the implementation, although the actual deployed configuration depends on the hardware resources the DCN operator has available to dedicate to logging.

Network traffic on the original DCN was required to trigger exchanges between the SDN controller and switches that would result in flow-rules being added to and removed from flow-tables over time. Many traffic generation tools are available

[101], however the most widely used of these (iPerf [102], netperf [103]) are constant or maximum throughput traffic generators, intended for data-plane performance testing. For this work, the requirement was for a tool to generate realistic data centre traffic loads. D-ITG [104] was considered (and was used later, as described in section 3.6.4, for stress-testing the capture filter), and could have been used for all experiments if appropriate values for D-ITG parameters were known to represent DCN traffic. Only one traffic generation tool, DCT2Gen, has been identified as specifically targeting DCNs.

DCT2Gen [105], a tool provided by the MaxiNet developers, generates traffic based on profiles created from DCN traces. In the experiments described here, traffic was required between the hosts in the emulated network – within and between MaxiNet worker VMs. A test profile and the traffic generator (TrafficGen) component of DCT2Gen was used. The profile contains details of 3667 TCP connections that the traffic generator schedules over a 10 second period between 120 hosts (arranged evenly in six racks). The traffic generator can loop over a traffic profile for continuous operation. The loop feature was used for experiments in this work.

In the original network, the ONOS [75] SDN controller (version 1.13) was used, but any OpenFlow controller could have been used. ONOS and OpenDayLight are the two open source SDN controllers most likely to be found in a DCN [106]. As mentioned earlier, mostly pro-active forwarding is expected to be used in a multi-tenant DCN, with the arrival and departure of tenants or modification of tenant resource requirements triggering the majority of OpenFlow rule changes. Since traces or configurations from a multi-tenant DCN were not publicly available, reactive forwarding was enabled in ONOS for these experiments to reflect the level of rule changes that are expected in a large, multi-tenant DCN.

3.6.2 Functional Verification Testing

Tests were devised to verify the correct functioning of the system, by showing that a recreated network matched the original network. This verification was essential because recreated networks were generated solely from the information gleaned

from OpenFlow messages, and not, for example, from any externally provided model of a network.

To verify that the original and recreated networks were the same, experiments were run gathering extra information from the original network at specific points in the experiment while the Logging Subsystem captured and logged OpenFlow messages. Recreating the network for each specific point in time, the same set of information was collected from the recreated network and compared with the first set.

On the original network, ONOS' REST API was used to obtain the list of inter-switch links, hosts, and host-switch links that ONOS learned about on the original network. Open vSwitch's ovs-ofctl tool produced a dump of the flow-table from each switch on the network. Linux fping was used to test connectivity between all pairs of hosts on the network. Note that connectivity between a pair of hosts depends on whether flow-rules have been installed to allow packets flow between the hosts. The verification check was not that all host pairs tested could communicate, rather that the result of a connectivity test between a pair of hosts in the recreated network was the same as the result of a connectivity test between the same pair of hosts in the original network (whether that result was a pass or a fail). On a network containing 120 hosts, this generated 14280 results on the original network to compare with the results of the same tests on the recreated network. As an example, the output of a set of connectivity tests on an original network performed during one experiment showed connectivity between 48 host-pairs. On a network recreated from the log of messages captured from the original network during the experiment, repeating the tests gave the same results.

The modified Floodlight controller used to populate flow-tables on a recreated network also has a REST API, through which a list of inter-switch links, and a list of hosts and host-switch links was extracted. The output was compared with the data collected from ONOS. The ovs-ofctl and fping tools were used as they had been on the original network, and the results compared.

Each verification experiment ran with traffic generated for 120 seconds, split into four 30-second intervals. At the end of a 30-second interval, the traffic generator was stopped to free up resources on the MaxiNet workers to allow verification information to be gathered. At the start of the next interval, the traffic generator started again, but with TCP connections shifted between racks. This was to ensure that the same set of flows was not running unchanged in each interval. Between traffic generation intervals, the controller was quiesced, the switches given time to deal with any outstanding FLOW_MOD requests, and the switch flow-table contents and topology and connectivity information gathered – this sequence of actions took around 2 minutes. Since there were four traffic-generation intervals, in each experiment there were three verification points. For the full duration of the experiment, the Logging Subsystem captured OpenFlow messages. The Network State Recreation Subsystem recreated the network separately for each verification point, and verification information was gathered for comparison with that collected from the original network.

The set of verification checks for which the original and recreated networks were compared is listed in Table 3.2. Data was recorded over more than 20 experiment runs to confirm the correct operation of the implementation. The results of the host checks initially gave pause: while a recreated network consistently had the same number of switches, switch links and flow-table contents as the original network, sometimes it had fewer hosts. This happens more often on a network that was recreated for a time close to the startup-time of the original network. The reason for the difference is that a host is only learned of at the control plane when that host actively sends data or has a packet addressed to it. It can be stated definitively that the recreated network had the same number of *active* hosts as the original network at the times for which the verification data was recorded.

3.6.3 Message Log Storage Requirements

The space required to store the message log relates to the number of messages in the log, although not all messages are the same size. As an example, a ~500K-message log collected from the Spine-Leaf DCN topology illustrated in Figure 3.3

during a 3-minute experiment took 318MB of space in Elasticsearch, without any optimisation to reduce disk usage. Extrapolating, this grows to 153GB per day, or 56TB per year. Such space requirements are not excessive, given current public cloud storage costs.

Network Elements Checked	What was confirmed to match between Original and Recreated Network
Switches	• Number of switches • Switch DPIDs
Links between switches	• DPIDs of switches connected by each link • Switchport numbers
Flow Tables	• Number of flow-rules on switches with matching DPIDs • Content of flow-rules on switches with matching DPIDs
Hosts	• Number of hosts • Host IP addresses
Host Connectivity	• ICMP success or fail between host pairs

Table 3.2: Network Recreation Functional Verification

As an indication of potential storage costs, the cost of storing 56TB of data on Amazon S3 in the 'Europe (Ireland)' region for one year, where the data is written at 153GB per day in units of 1MB (and allowing for reading at ten times that rate, to allow for frequent reproductions of the DCN), has been calculated at $16,544 using the AWS pricing calculator [107]. Compression and other forms of optimisation would reduce the storage and cost requirement. Furthermore, it should be noted that the 500K+ messages given as an example were collected from a DCN configured for reactive forwarding, rather than the proactive forwarding that would be primarily used in a multi-tenant DCN. Volumes of 500K+ messages in 3 minutes are possible with proactive forwarding but would be expected on a much larger DCN than the one used for experiments here.

3.6.4 Packet Capture Tool Stress Testing

The capture filter code was stress-tested through use of the D-ITG traffic generator, which allowed fine control of packet size and frequency. In the test environment,

the filter captured small packets of 125 bytes each at 355 Mb/s (355,000 packets per second) with no packets lost, more than an order of magnitude greater than the control-plane data rates observed in the verification experiments described in section 3.6.2. Interestingly, larger packets of 8Kbytes (a packet size of the order likely to be seen in jumbo frames on a DCN) could be captured with no loss at a rate of 849 Mb/s (13,625 packets per second).

The packet sizes selected for stress testing were guided by actual OpenFlow packet sizes observed during experiments. A chart showing the distribution of OpenFlow packet sizes captured during a verification experiment in shown in Figure 3.5. Note that the y-axis of the chart has a logarithmic scale, to allow the small quantities of the largest packets to be represented. For the 468K OpenFlow packets analysed for the chart, the average packet length was 183 octets, minimum 70 octets and maximum 7306 octets.

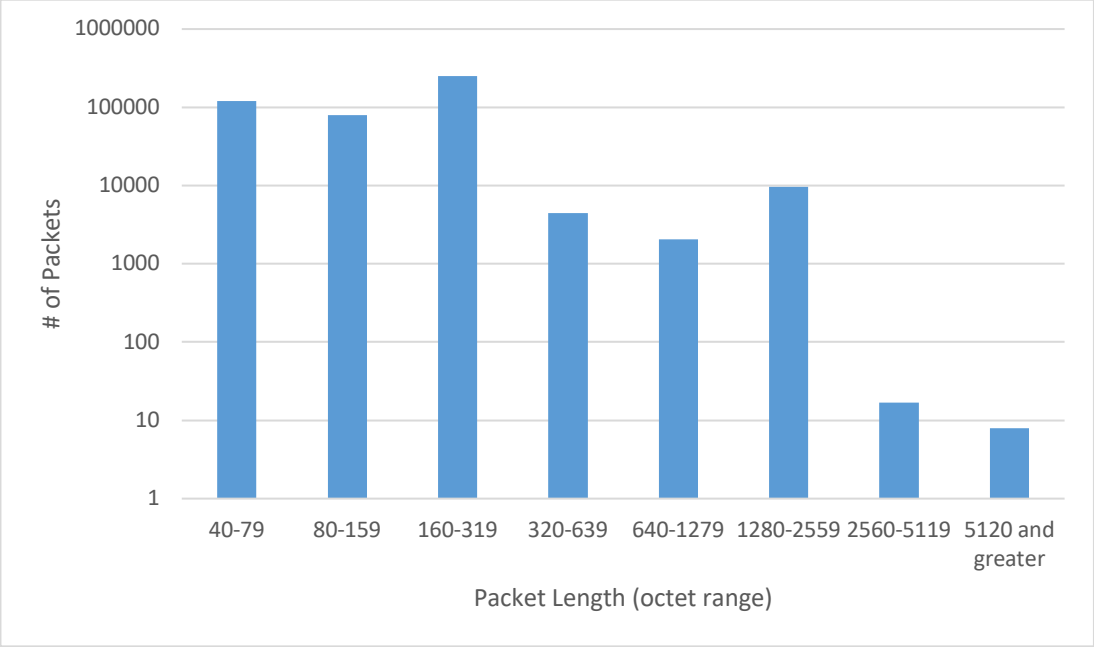


Figure 3.5: Distribution of OpenFlow Packet Lengths

ONOS has been performance tested in reactive mode by its developers and a single controller was capable of adding flow rules to a switch at a rate that matched the

rate of *Packet_In* messages up to approximately 3,000 per second [108]. In other words, a combined message rate of 5,997 per second (comprised of 3,000 *Packet_In* messages and the resulting 2,997 *Flow_Mod* messages) stretched the capacity of a single instance of the controller to its limit.

Information about the internal operation and usage of multi-tenant data centres is scarce, as cloud operators are slow to share competitively-sensitive information, making it difficult to quantify typical tenant 'churn' and resulting changes to proactively configured rules. A figure of 360,000 VMs in 24 hours (equating to 4.2 VMs per second) has been quoted [109] as the virtual machine deployment rate for a large cloud data centre in the United States of America. Researchers have estimated that, on average, each VM in a data centre has 10 active flows [110]. Rack-local flows require 2 flow-rules for bi-directional communication; rack-remote flows require 6 flow-rules on a spine-leaf network. Assuming a 75/25 split of rack-local to rack-remote flows, then the number of flow-rules to be configured for each VM is 30. With 4.2 VMs deployed per second, that would cause 126 'add-flow' *Flow-mod* messages to be generated in each second. Assuming that existing VMs are taken out of service at the same rate as those being deployed, requiring the same number of flow-rules to be removed from switches, this results in a total of 252 *Flow-mod* messages per second. That rate is an approximation, but is far less than the limit of the tested capabilities of a single ONOS instance mentioned above, and less again than what the capture filter can accommodate. Moreover, a large DCN would employ a clustered controller installation for fault-tolerance primarily, but also to comfortably accommodate spikes in flow-rule management demands. The experiments performed for this work produced message rates averaging approximately 400 *Flow-mod* messages per second on average, peaking at 600 per second.

3.7 Conclusions

This chapter described a method for passive, accurate recording of the history of control-plane messages on a DCN, and accurately reproducing the DCN in a resource-efficient manner. The purpose is to provide a DCN operator with the

ability to view and interact with their network in its historical state and topology for any point in time covered by the recorded history. An architecture was defined to meet the challenges of realising the method, prioritising scalability, accuracy, and resource-efficiency. Furthermore, the architecture showed how goals of passive operation controller- and switch-independence were achievable. The architecture was implemented, and assessed to verify the accuracy of the reproduced networks. shown to validate the selected methods. The results show that the method employed scales to the requirements of large data centres, accurately reproducing a DCN solely from the contents of a control-plane message log. The method does not add to an over-burdened DCN operator's workload, or interfere with the operation of the network, and requires minimal extra hardware. Chapter 4 describes an optimisation to the approach to address the time it can take to reproduce a network.

4 Using Snapshots to Speed up the Reproduction Time of an Emulated DCN

4.1 Introduction

Chapter 3 introduced a method to reproduce a Data Centre Network (DCN) with topology and flow-table state as it was at some timepoint in the past, from a log of control-plane messages. An issue that was not addressed in that chapter is the length of time required to reproduce the network. One component of that, the task of compiling the state and topology of the network, clearly increases in duration as more messages need to be processed to reach to desired timepoint. The issue is that the log files can be very large, containing messages covering weeks, months of years of DCN operation. The method presented in Chapter 3 requires that the state and topology of the DCN is compiled starting with the first message in the log. This chapter presents an optimisation to increase the efficiency of the reproduction process, through the creation of periodic snapshots of state and topology. The benefit of the optimisation is quantified and charted.

4.2 Motivation

Using the tools presented in Chapter 3, reproducing a DCN takes longer the greater the difference between the timepoint for which the network is to be reproduced and the time at which control-plane logging started. Naturally, the time taken is longer if more control-plane messages need to be processed to reproduce the network. Therefore, the delay is longer on a network with a 'busier' control plane – that is, where more control-plane messages were logged per second. It would be unreasonable to expect a DCN operator to use tools that take an indeterminate length of time to provide useful results. Running the tools on more powerful hardware could shorten reproduction times but their usefulness would still be limited in larger DCNs and across longer durations of historical message logs.

Message logs could contain messages captured over weeks, months, or even years. A figure of ~500K messages captured in 3 minutes was given for an experiment

described in Chapter 3. Reproducing the network for a timepoint corresponding to the timestamp of the last message in the log was timed at almost 5 minutes. Using those raw figures as a basis for calculations, a log of messages captured over 24 hours could contain 2.4×10^6 messages, and it could take 40 hours to reproduce the network with the state and topology it had at the end of the 24-hour period. This is clearly too long to expect a DCN operator to wait, hence the need for an optimisation to address the issue.

Splitting log files is not a solution – for example, starting a new log file every 6 hours. The problem is that, unless the DCN is started afresh for each logging period, possibly by restarting all switches, the state of the network is not known for the start of the logging period, and so cannot be compiled for any subsequent timepoint. Restarting the network at regular intervals in a multi-tenant data centre is clearly not an option.

4.3 Research Challenges

The research challenges for this part of the work were formulated as:

- How to reproduce a DCN in *bounded time*, building on the tools presented in Chapter 2, and to minimise any extra storage requirements.
- Retaining the ability to reproduce the DCN for *any timepoint* in its history covered by the message log file.

Meeting the challenges is difficult because log files can contain an arbitrarily large, and constantly growing, number of messages. A DCN operator will probably accept waiting a few minutes for the reproduced network to be prepared, but not longer. Likewise, the operator would probably reject the option of fast DCN reproduction times if that was limited to a small number of fixed timepoints each day.

Any new approach would still need to achieve the goals set out in Chapter 3 – of scalability, controller- and switch-independence, passive logging of control-plane messages, resource efficiency, and accuracy of topology and flow-table state. For that reason, it was decided to take the approach described for logging control-plane

messages and reproducing a network described in the previous chapter but to devise an enhancement to address the additional research challenges given above.

4.4 Snapshot Generation and Utilisation

The methodology devised was to create snapshots containing all the information required to reproduce the network at specific timepoints. The use of snapshots is a well-known methodology, for example for backup and restore of databases [111]. In this work, a 'snapshot' is defined as the set of data required to describe the state of the DCN at a specific point in time. The definition of 'state' is the same as that given in section 3.3: details of the topology (hosts, switches, links) and the forwarding device configuration. Snapshots would be created at regular intervals in the historic record. While the interval was not necessarily time-based, an operator may have a requirement for some time-based snapshots - for example, every night at midnight, or at least at the same time every day to allow direct comparison, or at higher frequencies. Alternatively, a snapshot could be triggered based on the occurrence of an event of interest on or change of state in the network – for example, a switch that has been newly added to the network establishing a connection to the Software-Defined Networking (SDN) controller.

The key requirements for the design of the snapshot generator were as follows:

- It should be an ‘add-on’ to the existing message capture and network reproduction system.
- A snapshot must be *succinct*. That is, it must contain all the information required for the reproduction of the network topology and state for the required time, and only that information.
- Storage space requirements for individual snapshots should be low, to minimise both total storage requirements and the cost of increasing the frequency of snapshots.
- Retrieval of snapshots should be fast, to help keep the overall time to reproduce a DCN low.

Several options were considered for the approach to the form of a snapshot. The first option was to reproduce the network for each snapshot point desired, and then backup the entire emulated network, for example using a virtualisation hypervisor snapshot function. This would incur significant storage costs, particularly for large networks. Moreover, the research challenge of retaining the ability to reproduce the DCN for any point in time covered by the log would require an additional mechanism to read from the message store and augment the snapshot by directly manipulating the state of the live reproduced network – adding, modifying and removing flow-rules, host, switches and links as appropriate to advance the state to the desired timepoint.

A second option considered was to reproduce the DCN for each snapshot point, then extract the network topology and contents of each flow-table to store for later use. Regenerating the reproduced network would be then achieved by configuring a network emulator to match the extracted topology and re-populating the switch flow-tables with the previously extracted rules. This would have lower storage requirements than the first option above, but would still require the additional mechanism to augment the snapshot.

The third option (the one that was adopted) was based on data structures in the Network State Recreator (NSR). The NSR defines a network topology data structure, and a flow-table data structure for each switch in the topology. The data structures are empty when the NSR starts reading a message log and are updated as appropriate with each sequential OF message from the message store until the time specified by the operator is reached and the network reproduced from the contents of the data structures. The Snapshot Generator follows the same process as the NSR, but reads the message log from start to finish and, at intervals during the process, stores the contents of both data structures in the Snapshot Store. Creating snapshots with this approach is faster than the other options, storage requirements are minimised (see section 4.6.2), and a snapshot can quickly be augmented to reflect the state of the DCN at timepoints other than those at which snapshots have been taken, as described below.

Some types of control-plane messages signal a topology change or a change in the configuration state of the forwarding devices. Creating a snapshot entails the processing of all such messages that were recorded between a known start point, and the timepoint for which the snapshot is required. The start point is either the start of the message log, or the timepoint of a previous snapshot, which then provides the starting state that is modified as appropriate for each successive processed message. When the criterion for taking a new snapshot is met, it is simply a matter of recording the 'current' state – that is, the state after it has been modified according to all of the processed messages between the starting point and this snapshot.

The information in a snapshot is a record of the observed topology of the network, i.e. the hosts, switches, host-to-switch links, inter-switch links, and for an OpenFlow network the contents of the switch flow-tables at the time of the snapshot.

The approach as outlined thus far creates snapshots for certain timepoints covered by the historical log, and those snapshots can be used to quickly reproduce the DCN. One of the research challenges was to retain the capability to reproduce the network for any time in its history. Just as a snapshot is used to provide the starting state upon which to build the state for the next snapshot, the same method can be used to arrive at the state for any desired timepoint within the range of the historical log: find the snapshot with the closest, but earlier time than the desired timepoint, then incrementally modify that snapshot state based on messages that were captured between the snapshot timepoint and the desired timepoint. The selection of control-plane messages from a message log to augment a snapshot is visualised in Figure 4.1.

The *snapshot delta* is the metric controlling the point in the message log after a snapshot when the next snapshot should be taken. Clearly, the larger the delta, the longer it could potentially take to reproduce the DCN: reproduction is quickest for timepoints at, or shortly after a snapshot, slowest for timepoints shortly before a snapshot.

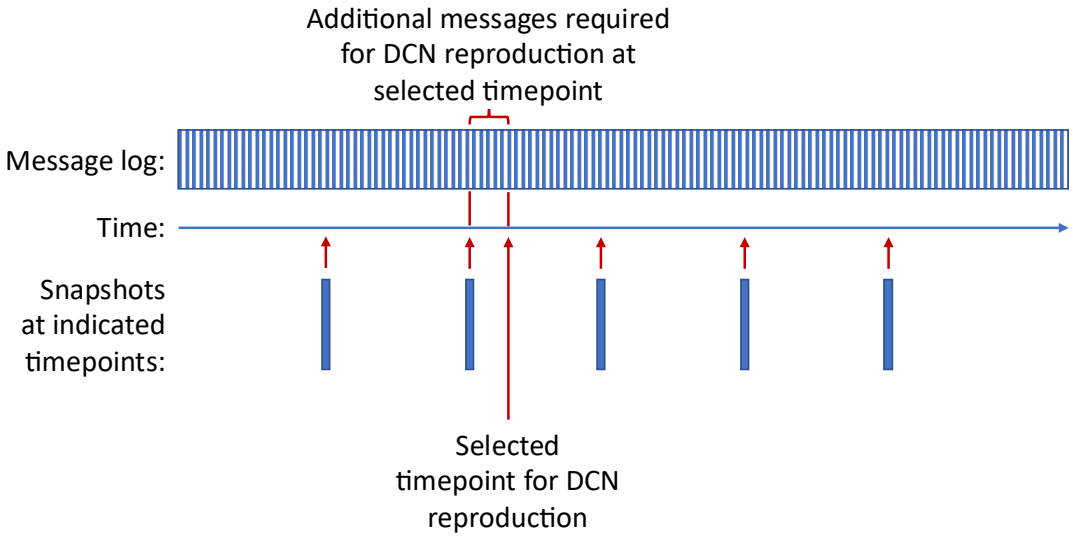


Figure 4.1: Augmenting a snapshot to compile DCN state for a selected timepoint

The snapshot delta could be quantified in units of time, for example create a snapshot for every 15 minutes of time elapsed in the history of the DCN. Or, it could be measured in units of control-plane messages, for example create a snapshot after 50K messages have been processed. Empirical observations of experiments performed for this work have shown that the latter metric results in a more consistent maximum-reproduction-time – that is, the time taken to reproduce the network for timepoints just before snapshots.

The architecture diagram in Figure 4.2 shows the addition of snapshot-related components to the existing message capture and network reproduction design presented in Chapter 3 (Figure 3.1). A Snapshot Generator processes logged messages to produce snapshots, which are recorded in the Snapshot Store. The existing Network State Recreator component takes a snapshot as input, augmented by input from the log of messages, as appropriate. The message log still contains the full set of captured control plane messages. The Snapshot Store is an additional component that provides multiple timepoints at which the state and topology of the DCN are known, and from which the state and topology at later timepoints can

be compiled from the message log, rather than the only option being to start from the beginning of the log.

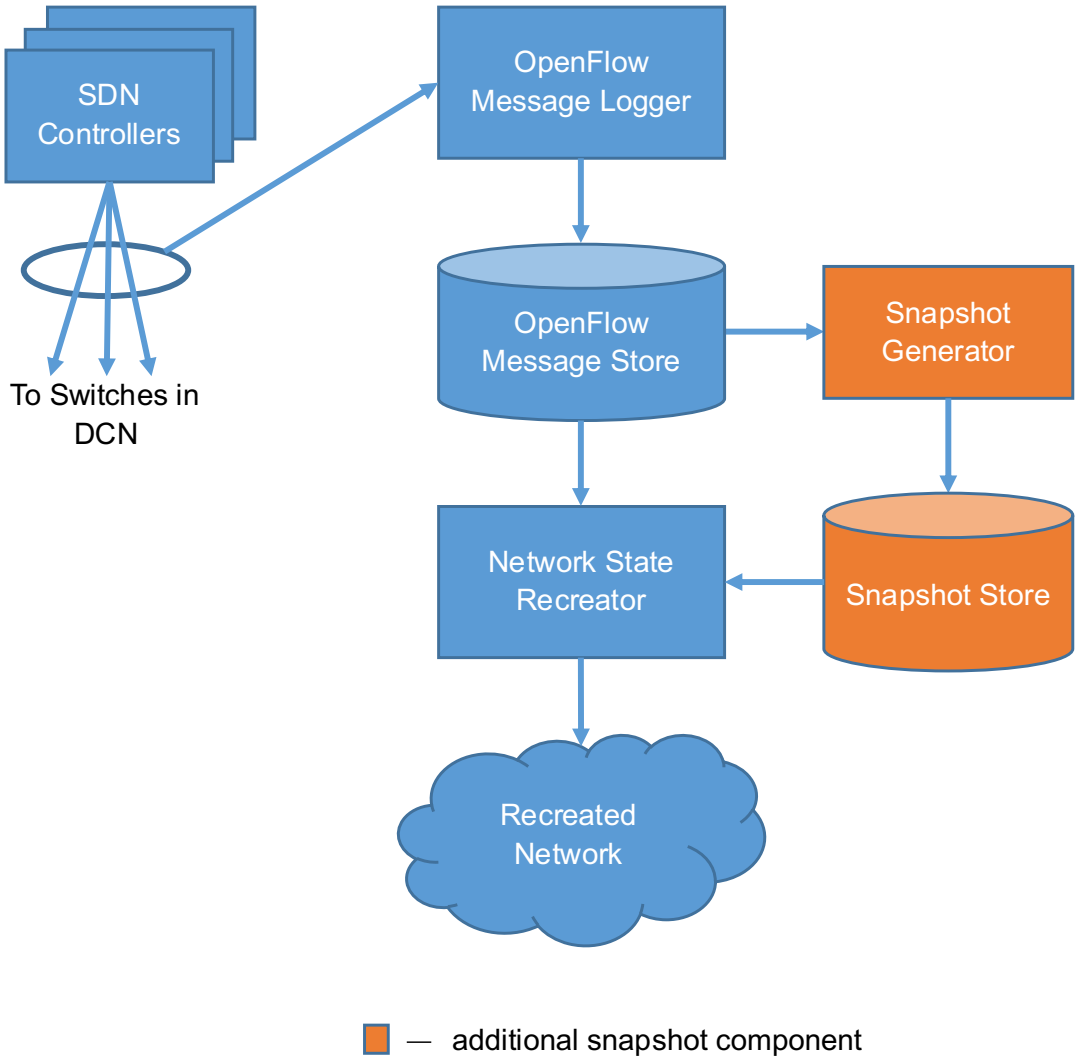


Figure 4.2: Architecture Diagram

4.5 Snapshot Components Implementation

The snapshot-related components were implemented and integrated with the system previously created for capturing control-plane messages and reproducing a DCN. The Network State Recreator was the only pre-existing component to require some modification - to search for and accept a snapshot as the basis on which to build the network state for a DCN operator-specified timepoint. The combined system, called LogSnap, is illustrated in Figure 4.3.

The Snapshot Generator queries Elasticsearch for OpenFlow messages, and processes them in chronological order. As it processes the messages, it learns about the topology of the network that the messages came from. It learns about the existence (and continued existence) and identities of switches and controllers from the messages that they exchange with each other. It learns about links between switches from the messages generated by the topology discovery process of the controller, for example LLDP packets contained in *Packet_In* messages. It learns about hosts, and how the hosts are connected to switches from ARP request packets contained in *Packet_In* messages. The Snapshot Generator also maintains a flow-table for each switch. As *Flow_Mod* messages are processed, rules are added, modified or deleted from the relevant flow-table. The logic is shown in Table 4.1.

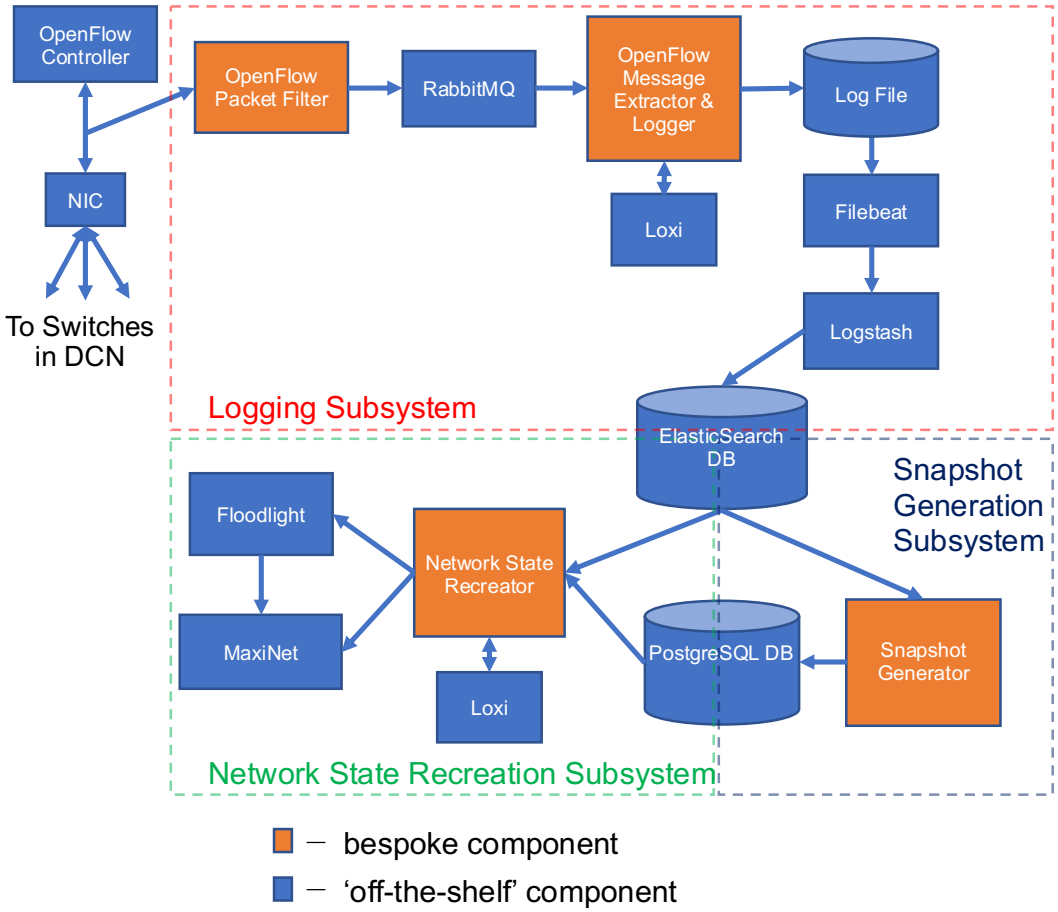


Figure 4.3: Implementation Diagram

Table 4.1: Snapshot Generator actions for messages processed from capture log

Message Type	Action(s) Taken

Any	• If the sender (Switch / Controller) is not in the topology object:
	  ○ Create a new Switch or Controller record
	  ○ Record the time at which the sender first seen
	• Else update the time the sender was last seen
	• Continue processing the message according to its type

Features_Reply	• Record the DPID of the sender in the relevant Switch record
Multipart_Reply	• If the message contains details of switchports, add / update the list of known ports for the Switch in the Switch record

Packet_In	• If the *Packet_In* contains an LLDP packet:
	  ○ If there is not a Link record for a connection between the source Switch of the *Packet_In* and port on which it was received, and the source Switch of the LLDP and the port on which it was sent:
	  ▪ Create a new Link record
	  ▪ Record the time at which the Link was first learned of
	  ○ Else update the time the Link was last learned of
	• Else if the *Packet_In* contains an ARP packet:
	  ○ If there is no Host record for the Host that sent the ARP packet:
	  ▪ Create a new Host record
	  ▪ Record the time at which the Host was first learned of
	  ○ Else update the time the Host was last learned of

Flow_Mod	• If the *Flow_Mod* contains an add_flow or modify_flow instruction:
	  ○ If there is not a Flow-Table object for the Flow-Table and Switch specified in the *Flow_Mod*, create it
	  ○ If there is already a Flow-Rule matching the specifications in the *Flow_Mod*, update the Flow-Rule in the relevant Flow-Table
	  ○ Else:
	  ▪ Create the Flow-Rule in the relevant Flow-Table
	  ▪ Record the time at which the Flow-Rule was added
	• Else if the *Flow_Mod* contains a delete_flow instruction, remove the Flow_Rule from the relevant Switch Flow-Table

Flow_Removed	• If the Flow-Rule exists in the relevant Flow-Table for the specified Switch, remove it.

TCP_FIN	• If the sender (switch / controller) has a record in the topology object, remove the record
	• If the sender is a switch:
	  ○ Remove all Flow-Rule records from all flow-table records associated with the switch
	  ○ Remove all Flow-Table records associated with the switch
	  ○ Remove all Host records for Hosts that are connected to the Switch
	  ○ Remove all Link records associated with the Switch

Snapshots, written to a PostgreSQL [112] database, consist of the set of controllers, switches, hosts, links, and flow-table contents that were known and valid at the time of the snapshot. A new snapshot is written when either all OpenFlow messages within a time interval or a certain number of messages have been processed since the last snapshot. The metric type and value are operator-configurable parameters. Setting the value lower reduces the maximum time taken to reproduce a network but increases storage requirements because snapshots are recorded more frequently. The trade-off between reproduction time and snapshot storage requirements is explored further in section 4.6.2.

PostgreSQL was chosen as the snapshot store because the structures of elements of the network topology and flow-tables lent themselves to the table structure of a relational database, allowing efficient addition and retrieval of snapshots compared to using flat file storage or the ElasticSearch database, for example.

From the DCN operator's point of view, reproducing a DCN using snapshots is no different to reproducing it without snapshots (as was achieved in Chapter 3). When invoking the Network State Recreator, the operator specifies the timepoint for which the DCN should be reproduced. The Network State Recreator then queries the PostgreSQL database for the closest snapshot prior to the timepoint (if such a snapshot has been recorded). It loads that information into a topology state object. It then queries the ElasticSearch database for any control-plane messages that were recorded between the timepoint of the snapshot and the timepoint specified by the operator. Next, the Network State Recreator modifies the topology state object as necessary by applying the effects of the messages returned by ElasticSearch. Lastly, the emulated DCN is created with the topology structure and flow-table contents defined in the topology state object.

There are some limitations to the current implementation. The link speeds of the original network are not reflected in the reproduced network. Although *Port_Status* messages are parsed to discover the addition and removal of ports, the 'blocked' state is not reflected. Furthermore, the implementation does not support multiple

OpenFlow tables per switch, group tables or meter tables. The design does not preclude support for these elements, and they could be added to the implementation in the future.

4.6 Evaluation

The snapshot functions of LogSnap were evaluated in the same environment as was described in Chapter 3. The primary purpose of the evaluation was to show the benefit of snapshots in setting an upper bound to the time it takes to compile the state information required to reproduce a DCN. Storage requirements of snapshots were also considered.

For the evaluation, the snapshot generator was run offline. In production, it is expected to run online, creating snapshots at intervals according to the metric type and value specified by the operator, and as control-plane messages arrive in the ElasticSearch database.

4.6.1 Demonstrating the Benefit of Snapshots

For a large DCN, with a sizeable number of and a turnover of tenants, the OpenFlow message history will be of volumes in the order calculated in section 4.2 – that is, 2.4×10^6 messages per day, or over 2,500 per second. Recreating a network from that history can be time, memory, and compute intensive. Experiments were devised to quantify the benefit of using snapshots. In the first experiment, using an OpenFlow message log recording the control-plane messages exchanged over approximately 3 minutes 15 seconds of the operation of a DCN, the network was independently reproduced for each second of time covered by the message log. For each reproduction, the message log was read from the start to compile the state for the desired reproduction timepoint, and the time taken to create the reproduction was recorded. In the second experiment, using the same message log, snapshots were created at intervals of 75K messages. Then, the network was again independently reproduced for each second of time covered by the message log. For this experiment, the closest previous snapshot was identified and retrieved, and the message log read from the time of the snapshot to the desired reproduction

timepoint. The time taken to create the reproduction, including identifying and retrieving a snapshot, was recorded.

Figure 4.4 plots the results of how long it took to recreate a network for each point in its history with and without the facility of being able to build the network state starting from a snapshot.

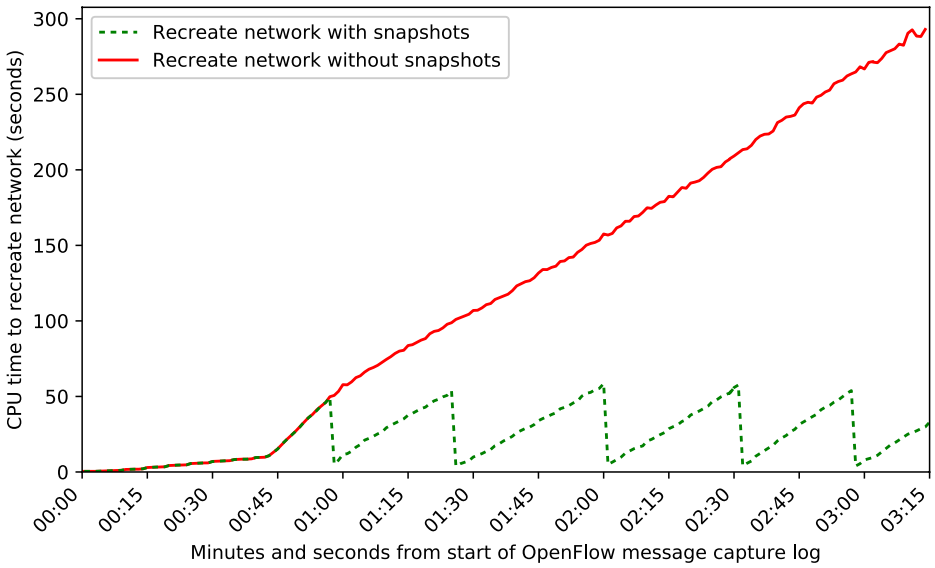


Figure 4.4: CPU time required to recreate a network as it was at a specific time

The original DCN is the one described in Chapter 3, consisting of six racks of twenty hosts and one top-of-rack switch each. The six top-of-rack or leaf switches were connected in a folded-clos topology to six spine switches to create a spine-leaf DCN architecture. The controller was operating in reactive mode, to reflect the conditions expected on a much larger DCN with a controller operating proactively.

Clearly, from the figure, the time to recreate a network without the use of snapshots increases for later network recreation times. It grows almost linearly with the number of OpenFlow messages that need to be processed to generate the state of the network at the required recreation time. It is not exactly linear as some OpenFlow messages require more processing than others - e.g. *Packet_In* messages are fully parsed because they may contain information about the network topology,

Features_Request messages are only taken as confirmation that the controller that sent the message still exists.

Conversely, there is an upper limit on how long it takes to recreate a network starting from a snapshot. The limit is at the point just before the next snapshot, and so depends on the frequency of snapshots. In these experiments, a snapshot was taken every 75K OpenFlow messages. It can be seen from the chart in Figure 4.4 that the maximum time taken to reproduce a DCN starting from a snapshot was approximately 50 seconds. The minimum was approximately 4 seconds. If a DCN operator was unwilling to wait 50 seconds, then a smaller snapshot delta could have been specific when creating the snapshots, at the cost of some extra space requirements. For example, if the operator did not want to wait longer than 20 seconds, then a snapshot delta of 25K OpenFlow messages would be appropriate to meet that goal with the specific set of hardware on which the Network State Recreation System was running when the data charted in Figure 4.4 was collected.

The results shown in Figure 4.4 were based on a message log containing approximately 500K OpenFlow messages. The log contents were captured over three minutes and fifteen seconds - Figure 4.5 shows the number of OpenFlow messages captured each second. The chart was generated based on the timestamps of the messages in the log. Traffic generation started about 35 seconds after message logging began, as can be observed by the spike to approximately 6,000 OpenFlow messages per second: the spike consists largely of ARP request packets in *Packet_Out* messages sent by the controller to switches.

SDN controllers commonly implement Proxy ARP to reduce the overhead of broadcast ARP packets on the network. With Proxy ARP, ARP requests from hosts are forwarded by switches only to the controller. The controllers maintains an ARP table, which it updates based on the source of the ARP request, and from which it extracts the information to answer the ARP request. If the controller's ARP table does not have the information required to answer the ARP request, it must broadcast an ARP request itself to all parts of the network by sending a *Packet_Out*

message containing the ARP request to each switch. At start-up the controller's ARP table is empty, and clearly there will be a surge of *Packet_Out* messages containing ARP requests if many hosts are attempting to send data to each other at that point.

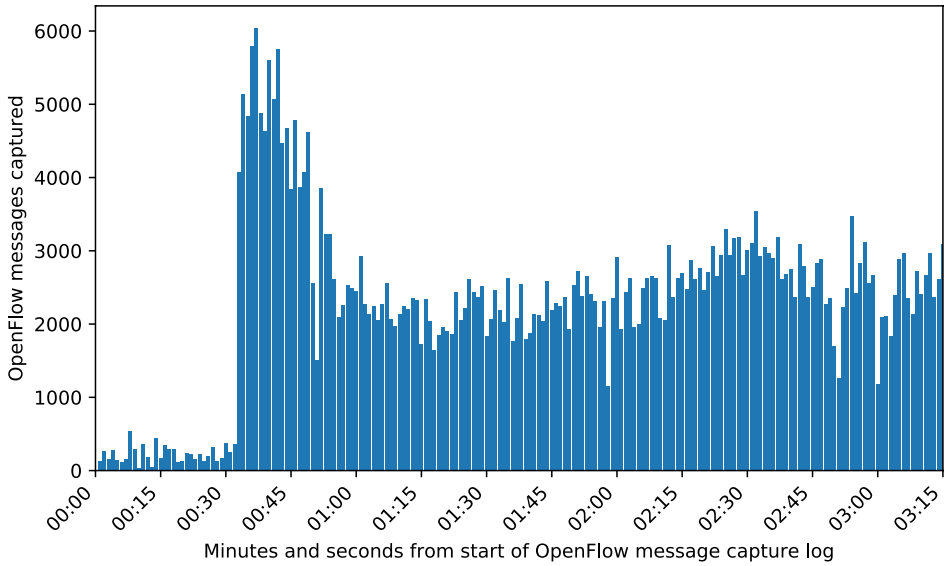


Figure 4.5: OpenFlow messages captured per second by capture filter

The short window of time represented by the two figures is sufficient to show the pattern of control-plane messages: following an initial spike, control-plane messages are sent at rate that remains broadly stable during normal operation of a busy DCN.

4.6.2 Storage requirements of snapshots

Storage requirements for snapshots can be calculated from some key parameters. A snapshot consists of information relating to controllers, switches, hosts, links, and the flow-rules for each switch. On any given network, the number of controllers (*C*), switches (*S*), hosts (*H*), and links (*L*) is known and is reasonably static – with infrequent changes due to DCN maintenance or reconfiguration. The set of flow-rules is more dynamic, depending primarily on how many traffic flows are active (*F*) at the time of the snapshot. Most flow-rules are only configured on switches on the path that packets need to follow to get from source to destination.

According to [113], approximately 75% of Cloud DCN traffic is rack-local, so packets only need to traverse one top-of-rack switch to get to their destination. The remaining 25% of traffic must traverse three switches (two leaf and one spine) to reach their destination in the DCN test network used in this work

Taking those factors into account, the storage requirement *B* for a single snapshot is calculated as:

$$B_a = CB_c + SB_s + HB_h + LB_l + (FF_l + FF_rF_x)B_f$$

Where the terms on the right hand side of the equation represent the following:

- *C* is the number of controllers,
- B_c is the number of bytes of storage required to represent one controller,
- *S* is the number of switches,
- B_s is the number of bytes of storage required to represent one switch,
- *H* is the number of hosts,
- B_h is the number of bytes of storage required to represent one host,
- *L* is the number of links,
- B_l is the number of bytes of storage required to represent one link,
- *F* is the average number of flows per second,
- F_l is the ratio of flows that are rack-local,
- F_r is the ratio of flows that are rack-remote, (and therefore require 3 flow-rules to be configured for end-to-end, unidirectional communication over a leaf-spine network),
- F_x is a multiplier for remote flows, reflecting the fact that a flow-rule must be installed on each switch along the path. The multiplier should be set to the average remote path length.
- B_f is the number of bytes of storage required to represent a flow-rule.

In the test network, element counts and byte requirements per element instance have been calculated from the experimental data: $C = 1$, $B_c = 67$; $S = 12$, $B_s = 173$; $H = 120$, $B_h = 78$; $L = 156$, $B_l = 83$; $F = 100$, $F_l = 0.75$, $F_r = 0.25$, $F_x = 3$, $B_f = 718$. This yields an estimated storage requirement $B = 132$ KB for one snapshot.

The number of snapshots per day depends on the total number of OpenFlow messages that will be captured in one day, which in turn depends on the size of the network (controllers, switches, hosts, links) and on either the flow arrival rate and mean flow duration in a reactive OpenFlow network, or on the rate of policy changes in a proactive OpenFlow network. Taking the figure of 500K messages in 3 minutes mentioned earlier: If the snapshot interval is 75K messages, then a snapshot will be taken approximately every 27 seconds, resulting in 3,200 snapshots in 24 hours and requiring approximately 421 MB of storage for that day of snapshot data.

Depending on the match criteria used, an individual flow-rule will require more or less space than the figure above. The flow-rules which that figure is calculated from match on source and destination MAC address. In other networks, those could be omitted from the match criteria, reducing the space required, or other fields added (e.g. source and destination IP address, layer 4 port numbers), increasing the space requirement for the rule. Otherwise the figures are expected to scale approximately linearly with the size of the network, e.g. 4 GB of snapshot storage per day for a DCN with 120 switches and 1,200 hosts, or 40 GB per day for 1,200 switches and 120,000 hosts.

Reducing the snapshot interval increases the number of snapshots taken. For example, halving the snapshot interval doubles the number of snapshots, and intuitively this should result in double the storage requirement. However, an optimisation has been applied to the snapshot database which reduces duplication between snapshots. The approach is to track the range of time over which elements of the network were inferred to be present – switches, hosts and links, for example. When an element is recorded in the database (as part of writing a snapshot), the time-range is recorded in the row for that element as a field of PostgreSQL's timestamp range (TSTZRANGE) data-type. If the element is still in the network at the time of the next snapshot, and has been there uninterrupted for the duration, the time-range field is updated rather than a new row being written. Reading a snapshot from the database is then a matter of searching for rows with time-ranges

within which the snapshot time and date falls. As a result of this optimisation, it has been observed that halving the snapshot interval causes only a 50% increase in the snapshot storage requirement.

4.7 Conclusions

This chapter described how the research challenge of making DCN reproduction time bounded was met. The novel idea of using snapshots was introduced, and a method presented by which the Logging and Network State Recreation Sub-Systems architecture from Chapter 3 was modified to integrate a Snapshot Generation Subsystem, to form the LogSnap system. The benefit of snapshots to control the length of time a DCN operator must wait for their network to be reproduced was demonstrated. The operator is provided with a flexible way to reduce the maximum wait time by increasing the frequency of snapshots, at a cost of greater snapshot storage requirements. Alternatively, the operator could decrease snapshot storage requirements by decreasing the frequency of snapshots, if they are willing to accept a longer maximum time to wait for DCN reproduction.

Chapters 3 and 4 have focussed on providing a DCN operator with a model of their network that is a functioning emulated copy with historical state and topology, and with which they can interact using familiar networking tools. Chapter 5 describes a different type of model, built from the same message logs as were used for chapters 3 and 4, to serve the purpose of answering sophisticated operator queries.

5 An Ontological Approach to Creating a Temporal and Topological Model of a DCN

5.1 Introduction

Chapter 3 presented methods to reproduce the state of a Data Centre Network (DCN) for any point in its recorded history from a log of control-plane messages captured from that network. It was shown how the state information could then be used to recreate an emulated copy of the network, to allow a DCN operator to test and experiment with the network in its historical state. With this facility, some questions the DCN operator has about the historical operation and configuration of the network can be answered, and in a very familiar, hands-on manner, using some of the tools that are commonly used in a live, production network. More complex queries, particularly temporal queries, require a different approach. In this chapter, an ontological approach to creating a temporal and topological model of a DCN is presented. The purpose of the model is to support temporal and/or topological queries. The relevant ontologies are described, and an implementation through which the ontologies are populated with instances is evaluated. Chapter 6 shows how the model can then be used to answer complex queries.

5.2 Motivation

The purpose of creating a temporal and topological model of a DCN is to provide the ability to answer queries about the past state of the network. The reasons for which a DCN operator is interested in such queries include:

- Auditing – for example, to check that the forwarding rules in place on a DCN at a particular time match the policy specified by the operator.
- Compliance – it might be necessary to show that forwarding behaviour was within the terms of a service level agreement with a tenant of the data centre - as is stated in [114], “In a multi-tenant environment, the ability to securely contain and isolate tenant traffic is a fundamental requirement”.

- Troubleshooting - having the ability to query past network state can be useful when troubleshooting one-off or intermittent problems.
- Verification - a DCN operator might want to verify reachability, isolation or other network properties for time points or intervals in the past.
- 'What-if' queries – these could be used to plan for failures, e.g. identifying what flows and tenants would have been affected if a specific switch or link failed at a particular point in time.

NetFlow / IPFIX [3, 4] and other protocols have been used by Network Management Systems to collect data-plane flow statistics to give network operators some visibility into historical traffic patterns. These protocols do not take advantage of the extra information contained in control-plane messages, and cannot provide answers to other types of queries. Other researchers have described their approach for recording and playing back packets on a campus LAN [66], and although this is useful, it does not directly provide answers to queries.

The queries listed in Table 5.1 are basic queries that must be answerable using any proposed temporal and topological model. These three queries are used in this chapter to verify the correctness of the model, working from a message log captured from a known network – i.e. answers to the queries should be predicted based on observations / measurements from that network, and the predicted answers should match the actual answers. Further queries are presented in Chapter 6, which focusses on the query mechanisms.

Table 5.1: Sample Queries

#	Query	Query Type
1	List the switches / hosts / controllers that were active in the network	Enumerating elements
2	For each inferred switch, show the durations for which the switch was active in the network	Temporal details for elements
3	For each inferred host, show the durations for which flow-rules were installed in switch flow-tables	Temporal details for elements based on relationships with other elements

All DCN elements, both physical and logical, in all of the listed queries need to be supported by the model. Physical DCN elements include switches, links, hosts, controllers. Logical DCN elements include flow-rules, flow-tables, host mobility. All

elements must be represented in the model in a way that includes temporal properties.

5.3 Research Challenges

The first challenge is to identify an appropriate method for modelling the DCN. The model must represent the topology of the network and the state of the switch flow-tables. It must also contain details of changes to the topology and flow-tables over time reflecting the evidence gathered from control-plane messages. A key requirement is that the timescale to be covered by the model is in the order of months or years. A Software-Defined Network (SDN) controller maintains a model of the network, but it reflects the controller's *current* view of the network and does not represent or retain historical information. Furthermore, the controller's model is designed to meet the functional requirements of the controller, not to provide an accurate target for complex operator queries. The modelling method must produce a topological and temporal model of a DCN that is accurate, extensible, temporally scalable, and lends itself to answering complex queries.

A second challenge relates to the available source data upon which the topological and temporal model is to be constructed. For the work described in this chapter, the source is a log of control-plane messages captured and stored by LogSnap. There are two caveats to the accuracy of the log that must be allowed for:

1. the timestamps on the messages are not necessarily the exact times at which the events occurred that are described by the messages, and
2. there is the possibility that some messages may not have been captured.

For example, when a switch tells a controller that it has removed a rule from a flow table, some time may have elapsed between when the rule was removed and when the switch sent the message to the controller. Or when a switch acknowledges a controller request to add a flow-rule to a flow-table, it might seem reasonable to assume that the rule was added at some point in time between when the switch received the *add_flow* request and when it sent the acknowledgement. In fact, other researchers have shown that this is not such a reasonable assumption [82],

and that it is possible for the flow-rule to be added after the switch sends the acknowledgement.

Additional uncertainty is partly the cost of the passive, unintrusive method used by LogSnap to collect control-layer information. While the work described here is based on message logs recorded by LogSnap, it could use any source of OpenFlow messages. It is worth noting that any packet-capture based source of such messages incurs the risk of missing data if, for example, the capture utility being unable to keep up with peak control-traffic flow.

There are other reasons for a message log appearing to be incomplete, including the failure of a device such as a switch, or a software process such as a controller crashing. These considerations need to be factored into the construction of the model, as they would need to be in any solution that aims to provide meaningful answers to DCN operator queries regarding the past state of the network.

5.4 Ontologies

5.4.1 Background

'Ontology' is a branch of philosophy, and is the science of 'what is'. It is the study of objects, or entities, which it seeks to classify and to explain their structure. The output of this exercise is called 'an ontology'. An ontological approach has been used in a variety of disciplines to create frameworks and taxonomies – for example, in linguistics, chemistry, biology, social sciences and computer science. For a general primer on ontologies, see [115].

Ontologies in Computer Science are most closely associated with the semantic web [116, 117] and artificial intelligence (AI) [118, 119]. In this context, an ontology formally encodes relationships between concepts, and those concepts can represent categories of physical or abstract items. Ontologies have been used successfully for SDN-related research efforts over the last number of years. One use-case for an ontological approach is to provide a mapping between overlapping sets of terminology used by different groups within the same domain. For example, creating a formal mapping between SDN controller requests and legacy network

device configuration languages [120]. Another use-case is the construction of a framework to define concepts for which instances can be constructed using data pulled from different sources – and this approach has been applied to combine collected network monitoring data to mine for troubleshooting purposes [121].

In this work, the use case for an ontological approach is to facilitate the construction of a logical framework based on control-plane messages, through which the challenge of inferring a topological and temporal model of a DCN can then be met. A less formal approach could have been taken; however the ontological approach has the advantage of making consistency and correctness easier to achieve, especially important when dealing with large sets of data. Moreover, ontologies can be extended and enhanced without impacting previously working code – for example, to accommodate changes in future versions of OpenFlow. Lastly, because an ontology is a structured definition of concepts and their relationships, and because it can be populated with data as properties of instances of the concepts, the ontology can be queried once it has been populated.

Ontologies are encoded in languages created for that purpose: Resource Description Framework (RDF)[122], RDF Schema (RDFS) [123], Web Ontology Language (OWL) [124]. The languages have different capabilities:

- RDF allows the description of things that are identified – i.e. basic facts to be stated.
- RDFS extends RDF by adding a means to identify groups of related facts, or concepts, and what the relationship between them is.
- OWL adds more vocabulary to RDFS, providing more expressive properties and classes.

OWL Version 2 is the most recent iteration of the OWL language. OWL Description Logic (OWL DL) is a subset of the full OWL language, restricting itself to features required for creating description logics for knowledge representation, and it has

several ‘profiles’ intended to make it more computationally suited for specific usage models:

- OWL 2 / EL, for large numbers of concepts and relationships, and where reasoning is the main requirement;
- OWL 2 / QL for large numbers of instances, and where querying is the main requirement;
- OWL 3 / RL for scalable reasoning using a standard rule language.

OWL depends on RDFS, which in turn depends on RDF. The relationship is often represented as a layered model, for example as shown in Figure 5.1.



Figure 5.1: Layers of Ontology Languages

There are further languages for serialising ontologies in text format for storage: Terse RDF Triple Language (TTL – also called Turtle), N-Triple (a subset of Turtle), Notation3 (a superset of Turtle), JSON for Linked Data (JSON-LD), and RDF/XML. While all listed storage formats can represent the same information, Turtle lends itself best to human readability. For a survey of serialisation formats, see [125].

Then there are querying and inferencing languages for ontologies: SPARQL Protocol and RDF Query Language (SPARQL) [126], SPARQL Inferencing Notation (SPIN) [127], Semantic Web Rule Language (SWRL) [128], Semantic Query-Enhanced Web Rule Language (SQWRL) [129], and Shapes Constraint Language (SHACL) [130].

The choice of which combination of languages to use is influenced by the requirements of a project, and what languages are supported by the tools selected to meet those requirements.

There are alternatives to the ontological approach that has been employed for this work. For example, a model-based approach using UML, implemented as an application using a traditional relational database. While this approach would

produce a functioning system, UML is not designed to support inference, meaning that inference features would be more difficult to include. Also, there are many features of OWL that are not supported (or are only partially supported) in UML. For more detail, see [131, 132]. Ontologies lend themselves to storage in graph databases (both the model and the instances), in which relationships are stored at the level of individual records, and new relationships can easily be added. In traditional relational databases, the relations are built into the table definitions, therefore new relationships require changes to the database schema. For a comparison of graph databases and relational databases, see [133].

5.4.2 Existing Ontologies

The ontologies created for this work are built on two existing ontologies: the Network Markup Language (NML) Base ontology [134], and the Change ontology [135]. The NML ontology contains some basic networking concepts for network objects such as nodes, ports, and links. It defines properties for these concepts – for example, *name*, *address*, *value*, *location*. It further defines relationships that may exist between the concepts, such as *hasLink*, *hasPort*. Its purpose is to facilitate the description of a traditional computer network, without SDN-specific characteristics. The basic concepts can immediately be built upon: one subclass of *node* could be *switch*, another could be *host*. An *address* could have subclasses of *ipAddress*, *macAddress*. The Change ontology gives us sophisticated temporal concepts, allowing us to describe elements of a network where some properties of those elements may change over time. For example, the IP address of a host may change, while the MAC address stays the same. Or a host may have a connection to one switch for a certain interval, but may be moved and connected to a different switch for a subsequent interval.

Several ontologies have been created previously by other researchers working to improve the task of network management in SDNs or of network monitoring. These have been evaluated and the decision made not to reuse them for this work, for the reasons described below.

ReasoNet [78] introduced an OpenFlow ontology (*reasonet_schema*), with concepts and relationships for network elements such as switches, ports, links, abstract path- and QoS-related concepts, and some OpenFlow-specific concepts. Multiple requests for access to the ontology sent to the paper authors only yielded access to a base ontology with no OpenFlow concepts. Therefore, it has not been possible to assess the potential for re-use of the full ontology. However, from the description in the paper, it is clear that *reasonet-schema* does not model the full set of OpenFlow messages – rather, it models an OpenFlow network, and *flow-mod* messages. Moreover, as noted in Chapter 2, *reasonet-schema* has no temporal component and has not been assessed for scaling to meet DCN requirements.

SDN Description Language (SDNDL) [79] is an ontology that defines a semantic model of OpenFlow SDN environments. It includes concepts of *Controller*, *OpenFlow Node*, *Flow Table*, and *Flow*. An *OpenFlow Node* is a switch. The *Flow* concept corresponds to a flow-rule, and has related concepts of *Timeout*, *Match*, *Instruction*, *Action*, *Flow Stats*, and *Priority*. While these concepts could be useful, they are not sufficient. There is no representation of OpenFlow messages, and no temporal aspect. As mentioned in Chapter 2, the model of the network constructed with SDNDL is limited to information gathered through the northbound interface of a controller instance, i.e. it provides a second-hand view of a network, subject to what the controller developers choose to expose through the northbound interface, bugs or design flaws in the controller, and the latency between when events occur on a network and when the controller's data structures are updated to reflect those events. Clearly, a control-plane message log provides a more accurate and detailed first-hand source of information.

The ontology created for FlowStats [121] is a model of a specific campus network, and is intended for keeping track of data-plane flow statistics gathered from the network. There are no SDN concepts, and no temporal dimension to the ontology that could reflect, for example, changes to the topology over time. Temporal data regarding flow statistics can be queried from a database.

Although the decision was taken not to reuse the ontologies described above, that does not mean that they could not be integrated in the future with the ontologies developed for this work, simply that it was not a requirement to integrate them at this point as no immediate benefit would have accrued. Where there is overlap between new ontologies proposed below and the ontologies described above, OWL has properties (*sameAs*, *equivalentClass*, and *equivalentProperty*) that can help map between the ontologies without necessarily having to redesign or refactor them.

5.4.3 New Ontologies

The basic concepts that need to be represented in the ontological model of the DCN, and that are not already present in the NML or Change ontologies, are the physical network components (switches, hosts, controllers), SDN-specific concepts (flow-tables, flow-rules), control-plane messages, as well as properties of and relationships between the concepts. The reason for including the control-plane messages in the model is to relate every inferred instance of a concept to the evidence that supports that inference.

The new concepts are split between two ontologies: an OpenFlow Message ontology[1], and a DCN History ontology[2]. The OpenFlow Message ontology is defined separately from the DCN History ontology for the reason that, although the message logs used in this work as source data contain OpenFlow messages, the approach can be applied using logs of other control-plane protocol messages (NETCONF, OVSDB, and/or BGP-LS, for example), or more generally any source of network management information. While the OpenFlow Message and DCN History ontologies are separate, they are related – since the OpenFlow messages are exchanged between controllers and switches, which are elements of the DCN. Figure 5.2 shows how the ontologies used in this work relate to each other. The two new ontologies are explored more below.

[1] Available online: https://www.purl.org/osearbhain/ontologies/openflow-message
[2] Available online: https://www.purl.org/osearbhain/ontologies/dcn-history

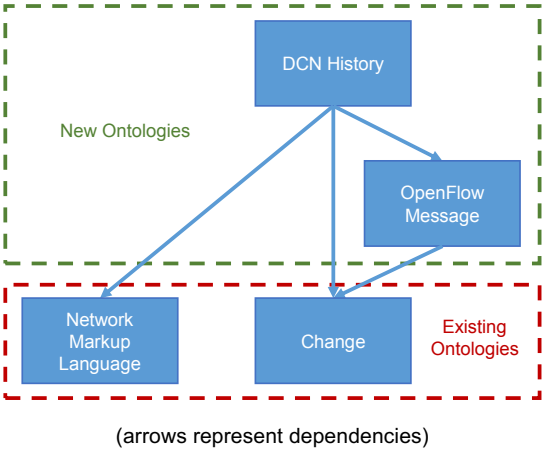


Figure 5.2: Hierarchy of Ontologies

The OpenFlow Message ontology describes types of OpenFlow messages and their properties, with the timestamp indicating when a message was captured as an additional property. Part of the ontology is shown in Figure 5.3. Omitted from the figure are the other OpenFlow message types, and the message properties (which correspond to the fields of each message type). Notice how an OpenFlow message has an 'existsAt' relationship with a temporal entity – its timestamp, with which other instances may have the same relationship if they represent events that occurred at the same time. A *FeaturesRequest* message is a subclass of an

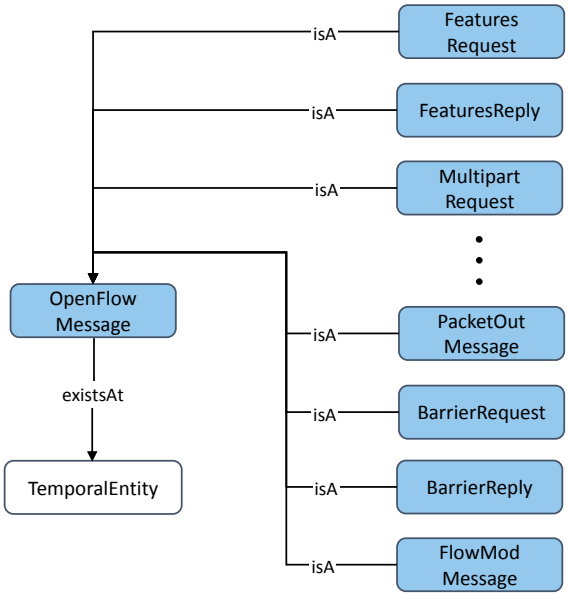


Figure 5.3: OpenFlow Message Ontology (Partial)

OpenFlow message (this is what the 'isA' relationship means), hence it inherits the 'existsAt' relationship and can have a timestamp associated with it.

Although the message ontology is relatively simple, its usefulness is the ability to, having populated it with message instances, link message instances with instances of other ontological concepts. For example, a message instance might be connected to a switch instance to represent the relationship "this is the first message that was captured from switch X", or connected with a flow-rule instance to represent the relationship "this is the message that caused flow-rule Y to be installed in the flow-table of switch Z".

The DCN History ontology describes the network elements that were present over time, and their relationships. These elements are the SDN controller(s) and SDN switches that exchanged control-plane messages, the links that the controller learned of through topology discovery, and the hosts that the controller learned about from data-plane messages forwarded from the switches to the controller – or potentially from other sources. SDN switches contain flow-tables, flow-tables contain flow-rules, and the tables and rules have properties that can change over time. Part of the DCN History ontology is shown later in this chapter.

5.5 Design and Implementation

In designing and implementing an approach to address the research challenges, it was important to firstly consider the key issue of uncertainty, and how to address it in an ontological model. This is discussed in 5.5.1 below. It was then necessary to identify what data in control-plane messages could be used to infer the existence of the components of a DCN, as detailed in 5.5.2 below. Section 5.5.3 describes the issue of how to represent change over time in a DCN History ontology. Lastly, the DCN History implementation workflow is laid out in 5.5.4.

5.5.1 Dealing with Uncertainty

The source data that will be used to populate the ontological model has inherent uncertainties, meaning that any instances that are inferred from the data will be subject to those uncertainties. OpenFlow messages are timestamped - those

timestamps are the times at which the packets containing the messages were captured, not the time at which they were sent or received, and not the time at which the event referred to by some message types (e.g. *FlowRemoved*, *PacketIn*) actually occurred. In a well-designed DCN, path lengths are short, packet loss low, and bandwidth should not be overly constrained, meaning that propagation times will be minimal. A delay between an event's occurrence and information about the event being relayed by the network device is a common attribute of network management systems, as is well understood by a DCN operator.

Researchers have proposed mechanisms to represent uncertainty in an ontology. PR-OWL [136] collates and refines previous work, and explains how random variables can be mapped to OWL properties, and multi-entity Bayesian networks (MEBNs) used to create probabilistic ontologies. This could be an interesting future addition to the ontologies presented here. However, a pragmatic, two-fold approach has been selected for this work:

1. Care has been taken not to introduce any extra uncertainty – there is no guessing or approximating.
2. Relationships are recorded between instances of inferred elements and the pieces of evidence that led to their inference.

Therefore, it is always possible to trace from any inferred element back to the origin data in the message log that was directly or indirectly used, possibly through multiple levels of inferencing, to conclude that the inferred element was present.

For example, a switch is inferred from control-plane messages that it sends, and the first and last messages of control-plane connection are linked with the switch instance. A controller is inferred from switch instances. Figure 5.4 illustrates a situation where multiple overlapping-in-time connections exist between a controller and several switches. Clearly, it can be confidently stated that the controller was continuously present for a period of time if it was continuously communicating with at least one switch during that period, even if it was not the same switch at all times of the period. A controller instance is linked with the set of

overlapping switch connections that led to its being inferred, and the controller instance's start and end times are related to the first message received by the first

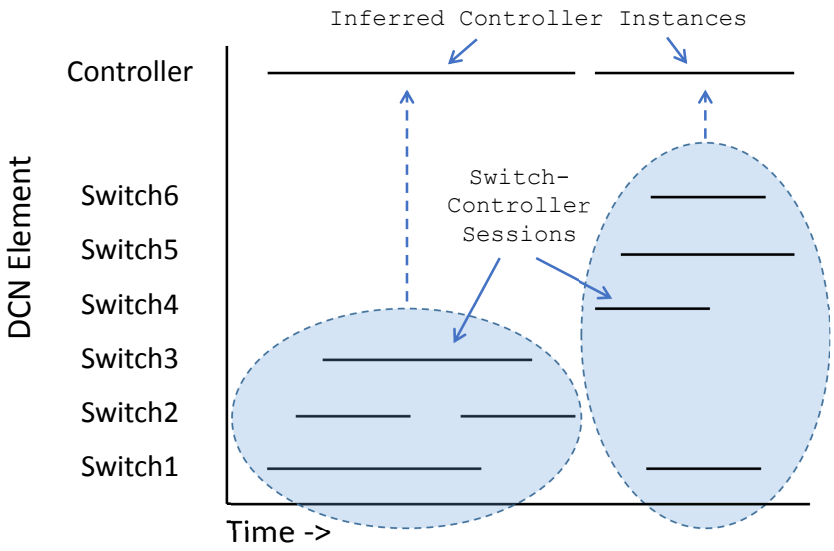


Figure 5.4: Inferring Controller Instances from Switch-Controller Sessions (illustration purposes only)

5.5.2 Inferring DCN Elements from OpenFlow Control-Plane Messages

Inference of elements of the DCN to populate the DCN History ontology depends on prior population of the OpenFlow Message ontology. A message instance in the OpenFlow Message ontology can be represented logically as:

$$(\exists M)(Message(M))$$

Every message has properties, for example source and destination IP addresses and layer 4 port numbers, a type and a timestamp:

$$(\forall M, \exists Q1, \exists Q2, \exists N1, \exists N2, \exists K, \exists T)(Message(M) \wedge Message.sourceIP(M, Q1) \wedge Message.destinationIP(M, Q2) \wedge Message.sourcePort(M, N1) \wedge Message.destinationPort(M, N2) \wedge Message.type(M, K) \wedge Message.existsAt(M, T))$$

Depending on the type of an OpenFlow message, it may have other properties. For example, in addition to the properties above, a FeaturesReply message contains the sending switch's datapath identifier:

```
(∃M, ∃D)(Message(M) ∧ Message.type(M, "FeaturesReply") ∧
      Message.DPID(M, D))
```

The detail of what evidence supports the inference of each DCN element is given in the subsections below, with a formal definition of inference statements encoded in 2nd order logic. This is followed by a summary of the inference dependencies. The logic symbols used in the statements above and below are listed in Table 5.2.

Table 5.2: Meanings of relevant logic symbols

Symbol	Meaning	Symbol	Meaning
∃	There exists	¬	Not
∀	For all	≺	Precedes (temporal)
∧	And	→	Implies
∨	Or		

As an aid to comprehension, variable identifiers are used consistently in the rules included here, representing instances of concepts and data properties as listed in Table 5.3.

Table 5.3: Logic variables used to represent instances of concepts and data properties

Variable	Represents	Variable	Represents
A	Match	N	Integer numeric value (TCP/UDP Port, Switchport number, Priority, Table Identifier)
B	InstructionSet	O	Object
C	Controller	P	Perdurant (Switch, Controller, Host, FlowRule)
D	Datapath Identifier	Q	IP Address
E	HostArpTemporalAdjacency	R	Switchport
F	Match Field	S	Switch
G	Mac Address	T	Timepoint
H	Host	U	Set (of Object, Switch, Message, HostActivity, Set)
I	HostMacFlowRuleOverlap	V	HostActivity
J	Graph	W	HostArpGroup
K	Type (of message)	X	HostFlowRuleGroup
L	Link	Y	Flow-table
M	Message	Z	Flow-rule (and HostMacFlowRule)

5.5.2.1 Inferring a switch

If a control-plane message log contains a message sent to TCP or UDP port 6653, then that message is:

- from a switch - identified by the source IP address and source port of the message, and
- to a controller - identified by the destination IP address and destination port number.

The message is evidence that the switch existed at the time that the message was sent. It is not necessarily evidence that the controller existed at the same time, since OpenFlow messages may be sent over connectionless UDP communications. However, a response or other messages from the controller to the switch provides the evidence that the controller existed. An exchange of messages between the two network elements over a period is evidence that both elements existed for that duration, assuming there are no significant gaps in the communication.

As a guide to what constitutes a significant gap: commercial implementations of OpenFlow from Arista and Juniper, for example, send *EchoRequest* messages every 10 seconds by default as a keepalive mechanism for control-plane connections. HPE devices use a default interval of 5 seconds for *EchoRequest* messages. If a maximum number (3 for Arista and HPE, 12 for Juniper) of successive keepalive attempts do not receive a response, the connection is assumed to have failed. For more examples of the sequences of messages exchanged between DCN switches and controllers, see Chapter 2.

Significant gaps in communication between switch and controller cast doubt on any assumption that the switch was continuously present in the network. It may have been there, and just become disconnected from the controller, or was idle. On the other hand, it may have been removed and replaced with a component that presents the same identity. Even in the latter case, clearly the intention is that the replacement is taking the role of the component that it replaces. To avoid uncertainty, separate instances of the switch are inferred, each with its own start and endpoint, as illustrated for instances of Switches 1 and 2 in Figure 5.4 earlier in this chapter.

The logic for inferring an instance of a switch, with properties of DPID, IP address, port number and start and end points, is as follows:

$$(\exists M1, \exists D, \exists Q, \exists N, \exists T1, \exists M2, \exists T2, \exists M3, \exists T3)(Message(M1) \wedge Message.type(M1, \text{"FeaturesReply"}) \wedge Message.DPID(M1, D) \wedge Message.sourceIP(M1, Q) \wedge Message.sourcePort(M1, N) \wedge Message.destinationPort(M1, \text{"6653"}) \wedge Message.existsAt(M1, T1)) \wedge earliestSwitchMessage(T1, Q, N, M2) \wedge Message.existsAt(M2, T2) \wedge latestSwitchMessage(T1, Q, P, M3) \wedge Message.existsAt(M3, T3) \rightarrow (\exists S)(\mathbf{Switch}(S) \wedge Switch.ipAddress(S, Q) \wedge Switch.tcpPort(S, N) \wedge Switch.DPID(S, D) \wedge Switch.startpoint(S, T2) \wedge Switch.endpoint(S, T3))$$

Informally: An instance of a switch is identified by the existence of a FeaturesReply message with the DPID of the switch;

- the start time of the instance is the timestamp of the earliest OpenFlow message with the same source port and source IP as the FeaturesReply, and that does not have a timestamp earlier than a TCP FIN or TCP Reset if that TCP FIN or TCP Reset has a timestamp earlier than the FeaturesReply message, and subject to there not being an intermediate FeaturesReply message with the same source IP and source Port;

- the end time of the instance is the timestamp of the latest OpenFlow message with the same source port and source IP as the FeaturesReply, and that does not have a timestamp later than a later FeaturesReply or TCP FIN or TCP Reset message with the same source port and source IP.

Note that the logic statement calls two functions, `earliestSwitchMessage()` and `latestSwitchMessage()`. These are defined as:

$$(\exists M1, \exists Q, \exists N, \exists T, \exists T1, \neg\exists M2)(earlierSwitchMessage\ (T, Q, N, M1) \wedge Message.existsAt(M1, T1) \wedge earlierSwitchMessage\ (T1, Q, N, M2) \rightarrow \mathbf{earliestSwitchMessage}(T, Q, N, M1))$$

Informally: The earliest OpenFlow message for a switch (identified by the IP address and port number of the switch) is the one for which no other message qualifies as being earlier.

$$(\exists M1, \exists Q, \exists N, \exists T1, \exists T2, \neg\exists M2)(laterSwitchMessage\ (T1, Q, N, M1) \wedge Message.existsAt(M1, T2) \wedge laterSwitchMessage\ (T2, Q, N, M2) \rightarrow \mathbf{latestSwitchMessage}(T1, Q, N, M1))$$

Informally: The latest OpenFlow message from a switch (identified by the IP address and port number of the switch) is the one for which no other message qualifies as being later.

The two function definitions above each depend on a further function, `earlierSwitchMessage()` and `laterSwitchMessage()` respectively. These are defined as:

```
(∃M1, ∃Q, ∃N, ∃T1, ∃T2, ¬∃M2)(Message(M1) ∧
      Message.sourceIP(M1, Q) ∧ Message.sourcePort(M1, N) ∧
      Message.existsAt(M1, T1) ∧ (T1 ≺ T2) ∧
      (isMessageTypeInRange(M2, "FeaturesReply", Q, N, T1, T2)
      ∨ isMessageTypeInRange(M2, "TCPFIN", Q, N, T1, T2) ∨
      isMessageTypeInRange(M2, "TCPRST", Q, N, T1, T2) ∨
      isMessageTypeInRange(M2, "Hello", Q, B, T1, T2) →
      earlierSwitchMessage(T2, Q, N, M1))
```

Informally: A message from a specified IP address and port number qualifies as being associated with a switch instance that has been identified at a later time, if there is no intervening FeaturesReply, TCP FIN, TCP Reset or Hello message.

```
(∃M1, ∃Q, ∃N, ∃T1, ∃T2, ¬∃M2)(Message(M1) ∧
      Message.sourceIP(M1, Q) ∧ Message.sourcePort(M1, N) ∧
      Message.existsAt(M1, T2) ∧ (T1 ≺ T2) ∧
      ¬(Message.type(M1, ”Hello”)) ∧ (isMessageTypeInRange(M2,
      "FeaturesReply", Q, N, T1, T2) ∨
      isMessageTypeInRange(M2, "TCPFIN", Q, N, T1, T2) ∨
      isMessageTypeInRange(M2, "TCPRST", Q, N, T1, T2) ∨
      isMessageTypeInRange(M2, "Hello", Q, N, T1, T2) →
      laterSwitchMessage(T1, Q, N, M1))
```

Informally: A message from a specified IP address and port number qualifies as being associated with a switch instance that has been identified at an earlier time, if the message itself is not a Hello message, and if there is no intervening FeaturesReply, TCP FIN, TCP Reset or Hello message.

Having identified a switch instance, a further check is required to ensure that there are no significant gaps between messages associated with that instance:

```
(∃S, ∃Q, ∃N, ∃M, ∃T1, ∃T2, ∃T3)(Switch(S) ∧
      Switch.ipAddress(S, Q) ∧ Switch.tcpPort(S, N) ∧
      Switch.startpoint(S, T1) ∧ Switch.endpoint(S, T2) ∧
      (Message(M) ∧ Message.sourceIP(M, Q) ∧
      Message.sourcePort(M, N) ∧ Message.destinationPort(M,
      "6653") ∧ Message.existsAt(M, T3)) ∧ ¬ (T3 ≺ T1) ∧ ¬
      (T2 ≺ T3) → (∃U)( MessageSet(U) ∧ (M ∈ U) ∧
      hasMessageSet(S, U)))
```

Informally: An instance of a switch has a message set, defined as all messages whose source port and source IP address match the TCP port and IP address of the switch instance, and whose timestamp is within the time range for which the switch instance existed.

```
(∃S, ∃U, ∃M1, ∃M2, ¬∃M3, ∃T1, ∃T2, ¬∃T3)(Switch(S) ∧
      hasMessageSet(S, U) ∧ (Message(M1) ∧ (M1 ∈ U) ∧
      Message.existsAt(M1, T1) ∧ (Message(M2) ∧ (M2 ∈ U) ∧
      Message.existsAt(M2, T2) ∧ (Message(M3) ∧ (M3 ∈ U) ∧
      Message.existsAt(M3, T3) ∧ (T1 ≺ T2) ∧ (T1 ≺ T3) ∧ (T3
      ≺ T2) ∧ ¬ (T2 - T1 > ConnectionExpiredThreshold))
```

Informally: For a pair of messages in a switch manifestation's message set, if there is no intervening message that is also in the message set, the difference between the timestamps of the two messages must not exceed the ConnectionExpiredThreshold.

The constant '`ConnectionExpiredThreshold`' is defined by the operator, most likely based on the type of switches used in the DCN. Based on the discussion earlier in this section, a value of 40 seconds would be appropriate for a DCN that uses Arista switches, for example.

5.5.2.2 Inferring a controller

Control-plane messages from a controller to a switch (within a session established by the switch) are evidence that the controller existed. An SDN controller typically communicates with multiple switches. Where these communication sessions overlap, it can be confidently stated that the controller was present from the start of the first overlapping session to the end of the last overlapping session. The logic for inferring an instance of a controller, with associated properties is as follows:

```
(∀S1, ∃Q1, ∃N1, ∃M1, ∀S2, ∃Q2, ∃N2, ∃M2, ∃Q3)(Switch(S1) ∧
      Switch.ipAddress(S1, Q1) ∧ Switch.tcpPort(S1, N1) ∧
      Message(M1) ∧ Message.sourceIP(M1, Q3) ∧
      Message.sourcePort(M1, "6653") ∧
      Message.destinationIP(M1, Q1) ∧
      Message.destinationPort(M1, N1) ∧ (Switch(S2) ∧
      Switch.ipAddress(S2, Q2) ∧ Switch.tcpPort(S2, N2) ∧
      Message(M2) ∧ Message.sourceIP(M2, Q3) ∧
      Message.sourcePort(M2, "6653") ∧
      Message.destinationIP(M2, Q2) ∧
      Message.destinationPort(M2, N2) ∧ (S1 ¬= S2) ∧
      overlaps(S1, S2) → (∃U, ∃C)( ControllerSwitchSet(U) ∧
      (S1 ∈ U) ∧ (S2 ∈ U) ∧ Controller(C) ∧
      hasControllerSwitchSet(C, U) ∧ Controller.ipAddress(C,
      Q3)))
```

Informally: Switch manifestations that have received control-plane messages from the same IP address and have overlapping time-ranges are elements of the same

ControllerSwitchSet, and furthermore there is a controller instance at that IP address that is associated with the ControllerSwitchSet.

The logic statement above depends on a generic function `overlaps()`, defined here:

$$(\exists O1, \exists O2, \exists T1, \exists T2, \exists T3, \exists T4)(Object(O1) \wedge Object(O2) \wedge Object.startpoint(O1, T1) \wedge Object.endpoint(O1, T2) \wedge Object.startpoint(O2, T3) \wedge Object.endpoint(O2, T4) \wedge ((T1 \prec T3) \wedge (T3 \prec T2) \vee (T1 \prec T4) \wedge (T4 \prec T2) \vee (T1 \prec T3) \wedge (T4 \prec T2) \vee (T3 \prec T1) \wedge (T2 \prec T4)) \rightarrow \mathbf{overlaps}(O1, O2))$$

The logic for identifying the lifespan of a controller instance based on the set of overlapping controller-switch connections is as follows:

$$(\forall C, \exists U, \exists A, \exists B)(Controller(C) \wedge hasControllerSwitchSet(C, U) \wedge earliestStartpoint(U , T1) \wedge latestEndpoint(U, T2) \rightarrow (Controller.startpoint(C, T1) \wedge Controller.endpoint(C, T2)))$$

Informally: The lifespan of a controller manifestation is from startpoint of the earliest switch manifestation that communicated with it to the endpoint of the latest switch that communicated with it, both switches being elements of the same ControllerSwitchSet.

The logic statement above depends on two further generic functions, `earliestStartpoint()` and `latestEndpoint()` which in turn depend on `earlierStartpoint()` and `laterStartpoint()` respectively:

$$(\exists U, \exists O1, \forall O2)(ObjectSet(U) \wedge (O1 \in U) \wedge (O2 \in U) \wedge \neg earlierStartpoint(O2, O1) \rightarrow \mathbf{earliestStartpoint}(U, O1))$$

$$(\exists U, \exists O1, \forall O2)(ObjectSet(U) \wedge (O1 \in U) \wedge (O2 \in U) \wedge \neg laterStartpoint(O2, O1) \rightarrow \mathbf{latestEndpoint}(U, O1))$$

$$(\exists O1, \exists O2, \exists T1, \exists T2)(Object(O1) \wedge Object(O2) \wedge Object.startpoint(O1, T1) \wedge Object.startpoint(O2, T2) \wedge (T1 \prec T2) \rightarrow \mathbf{earlierStartpoint}(O1, O2))$$

$$(\exists O1, \exists O2, \exists T1, \exists T2)(Object(O1) \wedge Object(O2) \wedge Object.endpoint(O1, T1) \wedge Object.endpoint(O2, T2) \wedge (T2 \prec T1) \rightarrow \mathbf{laterEndpoint}(O1, O2))$$

With the logic presented above, instances of the two types of DCN element that directly exchange control-plane messages are identified. Subsequently, instances of

other DCN elements are indirectly inferred from the contents of the control-plane messages.

5.5.2.3 *Inferring inter-switch links*

OpenFlow SDN controllers learn about inter-switch links through topology discovery – instructing switches to send LLDP (Logical Link Discovery Protocol) and BDDP (Broadcast Domain Discovery Protocol) packets out of each switch port, and to send any such discovery packets received back to the controller. These received discovery packets are preserved in *PacketIn* messages in the capture log. An LLDP or BDDP packet identifies the switch that originally sent it, and what port it was sent from, and the *PacketIn* message identifies the switch that received the discovery message and on what port it was received. The relevant fields of a *PacketIn*

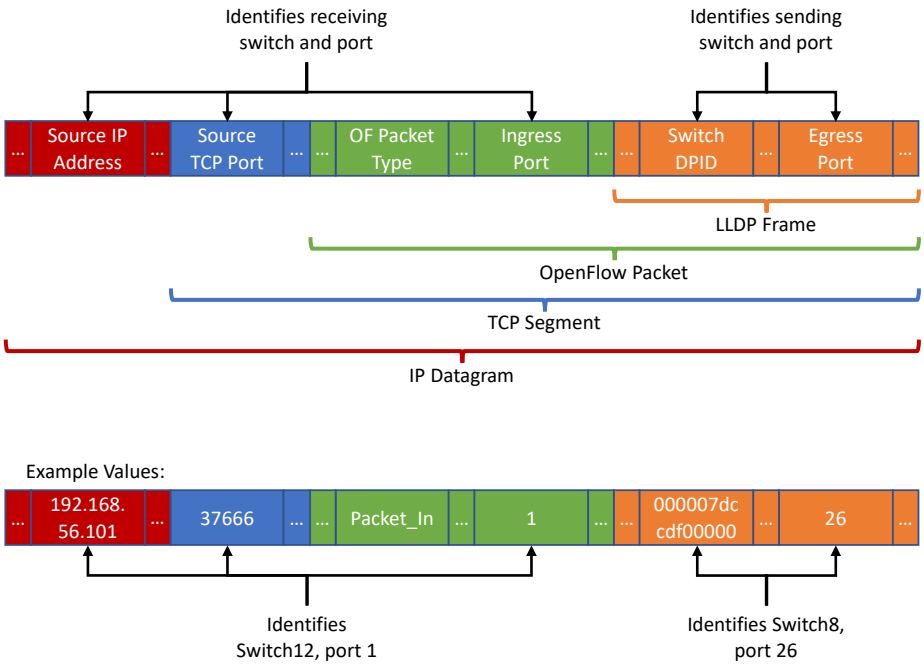


Figure 5.5: Relevant PacketIn and LLDP Packet Fields, with Example Values

message containing an LLDP packet are shown in Figure 5.5. The topology discovery process is re-run periodically (e.g. every 3 to 5 seconds, depending on the controller). A switch will also inform the controller via a *PortStatus* message if a port changes state (e.g. from up to down, or vice versa).

Inferring inter-switch links is achieved by processing the set of *PacketIn* messages containing LLDP and BDDP packets to identify what pairs of switch ports were

connected, and when and for how long they were connected. Relevant *PortStatus* messages refine the start- and end-points of an inter-switch link lifespan.

With respect to the logic for inferring inter-switch links, firstly switch ports are inferred from *MultipartReply* messages.

```
(∃M, ∃N1, ∃Q, ∃N2, ∃S)(Message(M) ∧ Message.type(M,
      "MultipartReply") ∧ Message.replyType(M,
      "PortDescriptions") ∧ Message.describesPort(M, N1) ∧
      Message.sourceIP(M, Q) ∧ Message.sourcePort(M, N2) ∧
      (Switch(S) ∧ Switch.ipAddress(S, Q) ∧ Switch.tcpPort(S,
      N2) ∧ existsDuringInterval(M, S) → (∃R)(Port(R) ∧
      hasPort(S, R) ∧ Port.portNumber(R, N1)))
```

Informally: An instance of a switch has a port if that port is mentioned in a MultipartReply message, if the message is from that switch's IP address and TCP port number, and the timestamp of the message is within the lifespan of the switch instance.

Inferring a switch port above depends on the generic function `existsDuringInterval()`:

```
(∃O1, ∃O2, ∃T1, ∃T2, ∃T3)(Object(O1) ∧ Object(O2) ∧
      Object.existsAt(O1, T1) ∧ Object.startpoint(O2, T2) ∧
      Object.endpoint(O2, T3) ∧ (T2 ≺ T1) ∧ (T1 ≺ T3) →
      existsDuringInterval(O1, O2))
```

After the switch ports have been inferred, *PacketIn* messages containing LLDP packets are parsed for details to infer the links between switch ports:

```
(∃M, ∃Q, ∃N1, ∃N2, ∃N3, ∃D, ∃S1, ∃S2, ∃R1, ∃R2)(Message(M) ∧
      Message.type(M, "PacketIn") ∧ Message.packetType(M,
      "LLDP") ∧ Message.sourceIP(M, Q) ∧ Message.sourcePort(M,
      N1) ∧ Message.receivedSwitchPort(M, N3) ∧
      Message.sentSwitchDPID(M, D) ∧ Message.sentSwitchPort(M,
      N2) ∧ Switch(S1) ∧ Switch.DPID(S1, D) ∧ hasPort(S1, R1)
      ∧ Port.portNumber(R1, N2) ∧ Switch(S2) ∧
      Switch.ipAddress(S2, Q) ∧ Switch.tcpPort(S2, N1) ∧
      hasPort(S2, R2) ∧ Port.portNumber(R2, N3) ∧
      existsDuringInterval(M, S1) ∧ existsDuringInterval(M,
      S2) → (∃L)(Link(L) ∧ hasConnectionFrom(L, R1) ∧
      hasConnectionTo(L, R2)))
```

Informally: An inter-switch Link is inferred from an LLDP packet that was received on a switch port, having been sent from another switch port. The lifespan of an inter-switch Link extends from the timestamp of the first LLDP packet between the two ports in question to the timestamp of the last such LLDP packet, subject to the

restriction that both LLDP packets must be within the lifespan of the two switch instances of which the switch ports are component parts.

With the logic presented so far, this significant knowledge about a DCN topology is inferred:

- The controllers and switches present and active in the network, and the times and durations for which they were active,
- The interconnections between switches, and any changes that occurred to those interconnections over the lifespan of the DCN.

The knowledge set is expanded further by inferring flow-rules that were present in switch flow-tables, and the active hosts that were connected to switches, as explored in the next sub-sections.

5.5.2.4 Inferring flow-rules and flow-tables

Details (match criteria, actions, priority) of individual flow-rules are extracted directly from the *FlowMod* messages instructing switches to add those rules. Flow-rule statistics are gathered from *MultiPartReply* messages sent by switches in response to periodic requests from controllers, and are a fallback for inferring the existence of flow-rules in the event of a *FlowMod* message having been missed. A flow-rule will normally be explicitly removed by a controller when it is no longer required, through a *FlowMod* with instructions to a switch to delete the rule. When initially adding a flow-rule, the controller can include an instruction to the switch to report if the rule is later removed for any other reason (e.g. idle-timeout, hard-timeout, or flow-table overflow). *FlowRemoved* messages, sent by a switch to a controller to report the removal of a flow-rule, are a further indicator of the end of the lifespan of a flow-rule.

A switch has at least one flow-table when it initially connects to a controller. If the switch has more than one flow-table, the number is recorded in a *FeatureResponse* message sent from the switch in response to a *FeatureRequest* from the controller at the beginning of a control-plane connection. A controller specifies to which table a rule should be added/modified, or from which table a rule should be removed in

the relevant FlowMod message. Later versions of OpenFlow allow a controller to query and re-configure flow-table configuration, and the detail of this interaction is recorded in *MultipartRequest* and *MultipartResponse* messages captured in the log file.

The logic for inferring flow-rules and flow-tables is as follows:

$$(\exists M1, \exists T, \exists Q, \exists N3, \exists S, \exists N1, \exists N2, \exists A, \exists B, \neg\exists M2)(Message(M1) \wedge Message.type(M1, \text{"FlowMod"}) \wedge Message.existsAt(M1, T) \wedge (Message.type(M1, \text{"flow_add"}) \vee Message.type(M1, \text{"flow_modify"})) \wedge Message.destinationIP(M1, Q) \wedge Message.destinationPort(M1, N3) \wedge getSwitchFromIPandTcpPort(Q, N3, S) \wedge Message.priority(M1, N1) \wedge Message.tableID(M1, N2) \wedge hasMatch(M1, A) \wedge hasInstructionSet(M1, B) \wedge earlierMatchingFlowMod(M1, M2) \rightarrow (\exists Z, \exists Y)(\mathbf{FlowRule}(Z) \wedge \mathbf{FlowTable}(Y) \wedge FlowTable.tableID(Y, N2) \wedge hasFlowRule(Y, Z) \wedge hasFlowTable(S, Y) \wedge FlowRule.startpoint(Z, T) \wedge FlowRule.priority(Z, N1) \wedge hasMatch(Z, A) \wedge hasInstructionSet(Z, B)))$$

$$(\forall Z, \exists T, \exists M, \exists S)(FlowRule(Z) \wedge (nextMatchingFlowRemoval(Z, M) \wedge Message.existsAt(M,T) \vee getSwitchFromFlowrule(Z, S) \wedge Switch.endpoint(S,T)) \rightarrow FlowRule.endpoint(Z, T))$$

Informally: A flow-rule, and the flow-table that contains it, is identified by the existence of a FlowMod (flow_add, flow_modify) message with specific values for these fields:

- *priority,*
- *match criteria,*
- *table ID,*
- *receiving switch IP and port.*

All previous FlowMod (flow_add or flow_modify) messages with the same field values refer to the same flow-rule, unless there is an intermediate FlowMod (flow_delete_strict) or FlowRemoved with the same values (except that the FlowRemoved message is sent by the switch rather than received). The timepoint of the earliest such FlowMod is the startpoint for the rule.

For every flow-rule, the endpoint is the timepoint of the first matching FlowMod (flow_delete_strict) or FlowRemoved following the flow-rule startpoint, if such a message exists. Otherwise, the endpoint is the endpoint of the switch to which the flow-table containing the flow-rule belongs.

The logic statements above depend on four functions –

`earlierMatchingFlowMod()`, `nextMatchingFlowRemoval()`,

`getSwitchFromIPandTcpPort()` and `getSwitchFromFlowrule()`:

$$(\exists M1, \exists T1, \exists M2, \exists T2, \exists Q, \exists N1, \exists S, \exists A, \exists N2, \exists N3, \neg\exists M3)(Message(M1) \wedge Message.type(M1, \text{“FlowMod”}) \wedge Message.existsAt(M1, T1) \wedge (Message.type(M1, \text{“flow_add”}) \vee Message.type(M1, \text{“flow_modify”})) \wedge Message.destinationIP(M1, Q) \wedge Message.destinationPort(M1, N1) \wedge Message.priority(M1, N2) \wedge Message.tableID(M1, N3) \wedge hasMatch(M1, A) \wedge Message(M2) \wedge Message.type(M2, \text{“FlowMod”}) \wedge Message.existsAt(M2, T2) \wedge (Message.type(M2, \text{“flow_add”}) \vee Message.type(M2, \text{“flow_modify”})) \wedge Message.destinationIP(M2, Q) \wedge Message.destinationPort(M2, N1) \wedge Message.priority(M2, N2) \wedge Message.tableID(M2, N3) \wedge hasMatch(M2, A) \wedge (T2 \prec T1) \wedge intermediateMatchingFlowRemoval(T2, M1, M3) \rightarrow \mathbf{earlierMatchingFlowMod}(M1, M2))$$

Informally: Given a FlowMod message, return a FlowMod message with an earlier timestamp if the two message are either ‘flow_add’ or ‘flow_modify’ instructions, and refer to the same rule – but only if there is not a FlowMod with ‘flow_delete_strict’ or a FlowRemoved message referring to the same rule and with a timestamp that falls between the timestamps of the other two messages.

$$(\exists Z, \exists T, \exists M1, \exists T1, \exists Q, \exists N, \exists S, \exists E, \exists F, \exists A, \neg\exists M2)(FlowRule(Z) \wedge FlowRule.startpoint(Z, T) \wedge hasFlowRule(Y, Z) \wedge FlowTable.tableID(Y, F) \wedge hasFlowTable(S, Y) \wedge Switch.ipAddress(S, Q) \wedge Switch.tcpPort(S, N) \wedge FlowRule.priority(Z, E) \wedge hasMatch(Z, A) \wedge Message(M1) \wedge Message.existsAt(M1, T1) \wedge (Message.type(M1, \text{"FlowMod"}) \wedge Message.type(M1, \text{"flow_delete_strict"}) \wedge Message.destinationIP(M1, Q) \wedge Message.destinationPort(M1, N)) \vee (Message.type(M1, \text{"FlowRemoved"}) \wedge Message.sourceIP(M1, Q) \wedge Message.sourcePort(M1, N)) \wedge Message.priority(M1, E) \wedge Message.tableID(M1, F) \wedge hasMatch(M1, A) \wedge (T2 \prec T1) \wedge intermediateMatchingFlowRemoval(T2, M1, M2) \rightarrow \mathbf{nextMatchingFlowRemoval}(Z, M1))$$

Informally: Given a FlowRule message, return the first message following the flow-rule’s startpoint that indicates the timepoint at which the flow-rule was removed – a FlowMod with ‘flow_delect_strict’ instruction sent to the switch or a FlowRemoved message sent from the switch to the controller.

```
(∃Q, ∃N, ∃S)(Switch(S) ∧ Switch.ipAddress(S, Q) ∧
      Switch.tcpPort(S, N) → getSwitchFromIPandTcpPort(Q, N,
      S))

(∃Z, ∃Y, ∃S)(FlowRule(Z) ∧ FlowTable(Y) ∧ hasFlowRule(Y, Z) ∧
      hasFlowTable(S, Y) → getSwitchFromFlowRule (Z, S))
```

The first two of the four functions above depend on one more function, `intermediateMatchingFlowRemoval()`:

```
(∃M1, ∃T1, ∃M2, ∃T2, ∃T3, ∃Q, ∃N, ∃E, ∃F, ∃A)(Message(M1) ∧
      Message.existsAt(M1, T1) ∧ Message.destinationIP(M1, Q)
      ∧ Message.destinationPort(M1, N) ∧ Message.priority(M1,
      E) ∧ Message.tableID(M1, F) ∧ hasMatch(M1, A) ∧
      Message(M2) ∧ Message.existsAt(M2, T2) ∧
      (Message.type(M2, "FlowMod") ∧ Message.type(M2,
      "flow_delete_strict") ∧ Message.destinationIP(M2, Q) ∧
      Message.destinationPort(M2, N)) ∨ (Message.type(M2,
      "FlowRemoved") ∧ Message.sourceIP(M2, Q) ∧
      Message.sourcePort(M2, N)) ∧ Message.priority(M2, E) ∧
      Message.tableID(M2, F) ∧ hasMatch(M2, A) ∧ (T3 ≺ T1) ∧
      (T3 ≺ T2) ∧ (T2 ≺ T1) →
      intermediateMatchingFlowRemoval(T3, M1, M2))
```

Informally: Return any relevant FlowMod with 'flow_delete_strict' instruction to the switch or FlowRemoved message from the switch with a timestamp in the range of a supplied timepoint (T3) and the timestamp of a supplied message (M1). The FlowMod or FlowRemoved is relevant if it refers to the same flow-rule as M1, and was sent or received by the same switch.

With the logic above, the existence and details of flow-rules and flow-tables are inferred from the FlowMod messages sent by controllers to switches to add, modify and delete the rules from the flow-tables, and the FlowRemoved messages sent by switches to controllers informing or confirming that flow-rules have been deleted.

5.5.2.5 Inferring hosts and host-switch links

The existence of hosts and links between hosts and switches are primarily inferred based on ARP request packets encapsulated in *PacketIn* messages in the capture log. The ARP packet identifies the sending host by MAC and IP addresses, and the *PacketIn* message containing the ARP packet identifies the switch and switchport on which the ARP packet was received. The relevant fields of a *PacketIn* message containing an ARP packet are shown in Figure 5.6.

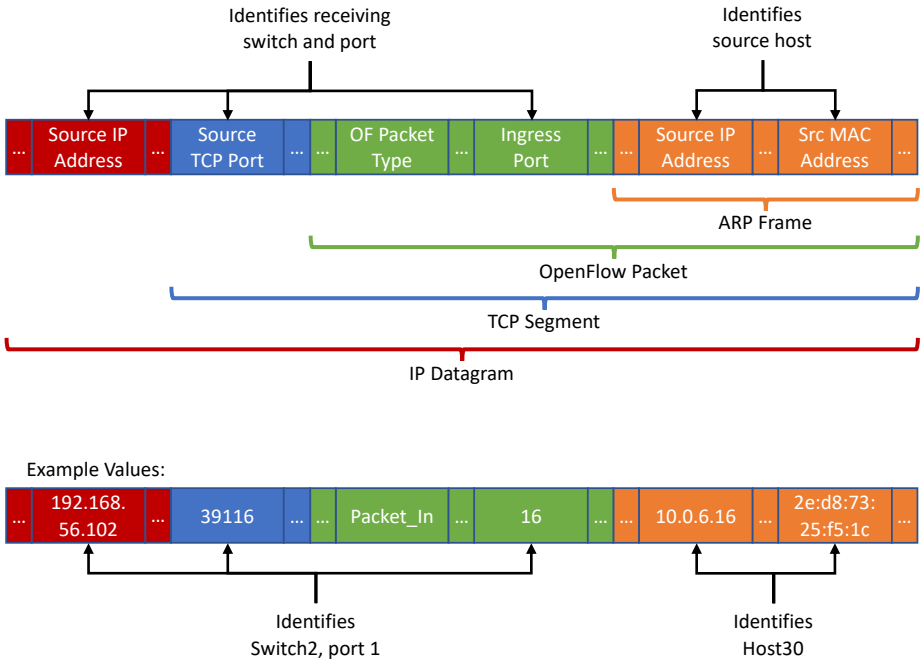


Figure 5.6: Relevant PacketIn and ARP Packet Fields, with Example Values

ARP request packets are useful data points that indicate when the host that sent an ARP attempted to initiate communication. While an ARP request is evidence that the source host was present at an instant identified by the timestamp, it can also reasonably be expected that the host was present for an interval after the ARP request. In the logic below, that interval is represented by the constant `ArpTrustLimit`. As a point of reference, host identities learned via ARP replies are trusted by modern desktop and server operating systems for between 15 and 45 seconds: the timeout for ARP table entries on Windows and Linux is a random value in the range of 15 to 45 seconds. In that context, a value of 5 seconds is a conservatively acceptable choice for `ArpTrustLimit`, although the value is configurable by the DCN operator.

The common response of an SDN controller to receipt of an ARP request is to install one or more flow-rules to allow the communication to progress. The lifespan of a flow-rule is used as evidence of the continued existence of the host if at least one of the flow-rules is host-specific (that is, has packet matching criteria identifying the IP address, MAC address and/or source port on which the host is connected to a switch). The capture log contains the *FlowMod* request sent from the controller to

instruct the switch to install the flow-rule. It also contains a subsequent *FlowRemoved* message from the switch to the controller indicating that the flow-rule was removed. SDN controllers periodically request statistics information about flow-rules from switches, and from the replies recorded in the capture log, the point at which a flow-rule stopped matching packets (i.e. the host stopped sending packets) is coarsely identified. To handle the case of there being no initial ARP request, or at least none recorded, the point at which a flow-rule started matching packets is also inferred.

The logic for inferring a host from the evidence of ARP requests and flow-rules is as follows:

$$(\exists G, \exists U1, \exists U2, \exists V, \exists W, \exists X, \exists Q, \exists A, \exists B, \exists S, \exists R, \exists I, \exists J, \exists U3, \exists U4, \exists T1, \exists T2)(\text{HostMacAddress}(G) \wedge (U1 = \{V : (\text{hasHostActivity}(G, V)\}) \wedge (V \in U1) \wedge (\text{HostArpGroup}(W) \wedge \text{HostArpGroup.hostIPAddress}(W, Q) \wedge \text{hasSwitch}(W, S) \wedge \text{hasInPort}(W, R) \vee \text{HostFlowRuleGroup}(X) \vee \text{hasSwitch}(X, S) \wedge \text{hasInPort}(X, R)) \wedge (U2 = \{I : (\text{HostMacFlowRuleOverlap}(G, I)\})) \wedge (J = \text{Graph}(U1, U2)) \wedge (U3 = \text{ConnectedComponents}(J)) \wedge (U4 \in U3) \wedge \text{earliestStartpoint}(U4, T1) \wedge \text{latestEndpoint}(U4, T2) \rightarrow (\exists H, \exists L)(\mathbf{Host}(H) \wedge \text{Host.macAddress}(H, G) \wedge (\neg Q \vee \text{Host.ipAddress}(H, Q)) \wedge \text{Host.startpoint}(H, T1) \wedge \text{Host.endpoint}(H, T2) \wedge \mathbf{Link}(L) \wedge \text{hasConnectionFrom}(L, H) \wedge \text{hasConnectionTo}(L, R)))$$

$$(\exists G, \exists O)(\text{HostMacAddress}(G) \wedge (\text{hasHostFlowRuleGroup}(G, O) \vee \text{hasHostArpGroup}(G, O) \rightarrow \mathbf{hasHostActivity}(G, O))$$

Informally: A host is identified by evidence of host activity. Host activity is defined as the existence of a set of ARP-groups and/or flow-rule-groups that are:

- *overlapping in time,*
- *relating to the same MAC address,*
- *relating the same IP address, where specified (for example, a flow-rule may specify MAC address and not IP address),*
- *and where relevant packets were received on the same switch port.*

The start time of the host is the start time of the earliest such evidence of host activity for this MAC address; the end time of the host is the end time of the latest such evidence of host activity for this MAC address. A host is accompanied by a link, connecting the host to a specific switch port, with a lifespan the same as the host.

Logic statements for inferring groups of temporally overlapping flow-rules that match the same host:

```
(∀G, ∃E, ∃N, ∃R, ∃S, ∃U1,∃L, ∃J, ∃U2, ∃U3,
      ∃U4)(HostMacAddress(G) ∧ (U1 = {Z :
      (hasHostMacFlowRule(G, Z)}) ∧ (Z ∈ U1) ∧ hasSwitch(Z, S)
      ∧ hasInPort(Z, N) ∧ (U2 = {I :
      (hasHostMacFlowRuleOverlap(G, I)})) ∧ (J = Graph(U1,
      U2)) ∧ (U3 = ConnectedComponents(J)) ∧ (U4 ∈ U2) →
      (∃X)(HostFlowRuleGroup(X) ∧ hasHostMacFlowRuleGroup(G,
      X) ∧ hasHostMacFlowRuleSet(X, U4) ∧ hasSwitch(X, S) ∧
      hasInPort(X, N)))

(∀X, ∃U1, ∃T1, ∃T2)(HostFlowRuleGroup(X) ∧
      hasHostMacFlowRuleSet(X, U1) ∧ earliestStartpoint(U1,
      T1) ∧ latestEndpoint(U1, T2) →
      (HostFlowRuleGroup.startpoint(X, T1) ∧
      (HostFlowRuleGroup.endpoint(X, T2)))
```

Informally: For a particular MAC address, create a graph formed by the edge-switch flow-rules that match against packets with that address as source on a common port. Each connected component of the graph is a set of flow-rules that overlap in time, referred to in the logic as a HostFlowRuleGroup.

The logic statements above depend on a function `HostMacFlowRuleOverlap()` to return pairs of rule with common match criteria for a host. Note that this function makes use of the generic `overlaps()` function defined earlier:

```
(∃Z1, ∃Z2, ∃Y1, ∃Y2, ∃S1, ∃S2, ∃A1, ∃F1, ∃A2, ∃F2, ∃G,
      ∃N)(HostMacAddress(G) ∧ hasHostMacFlowRule(G, Z1) ∧
      hasHostMacFlowRule(G, Z2) ∧ (Z1 ¬= Z2)  ∧ Switch(S1) ∧
      hasFlowTable(S1, Y1) ∧ hasFlowRule(Y1, Z1) ∧ Switch(S2)
      ∧ hasFlowTable(S2, Y2) ∧ hasFlowRule(Y2, Z2) ∧
      hasMatch(Z1, A1) ∧ hasMatchField(A1, F1) ∧
      MatchField.fieldName(F1, "in_port") ∧
      MatchField.fieldValue(F1, N) ∧ hasMatch(Z2, A2) ∧
      hasMatchField(A2, F2) ∧ MatchField.fieldName(F2,
      "in_port") ∧ MatchField.fieldValue(F2, N) ∧ overlaps(Z1,
      Z2) → (∃I)(HostMacFlowRuleOverlap(I) ∧
      hasHostMacFlowRuleOverlap(M, I) ∧
      hasOverlappingFlowRule(I, Z1) ∧
      hasOverlappingFlowRule(I, Z2)))

(∃Z, ∃A, ∃F1, ∃G, ∃F2, ∃N, ∃S1, ∃S2, ∃Y, ∃R1, ∃R2)(FlowRule(Z)
      ∧ hasMatch(Z, A) ∧ hasMatchField(A, F1) ∧
      MatchField.fieldName(F1, "eth_src") ∧
      MatchField.fieldValue(F1, G) ∧ hasMatchField(M, F2) ∧
      MatchField.fieldName(F2, "in_port") ∧
```

```
MatchField.fieldValue(F2, N) ∧ Switch(S1) ∧
hasFlowTable(S, Y) ∧ hasFlowRule(Y, Z) ∧ hasPort(S1, R1)
∧ Port.portNumber(R1, N) ∧ ¬(hasConnectionTo(L, R1) ∧
Link(L) ∧ hasConnectionFrom(L, R2) ∧ hasPort(S2, R2) ∧
Switch(S2))  → HostMacAddress(G) ∧ HostMacFlowRule(Z) ∧
hasHostMacFlowRule(G, Z) ∧ hasSwitch(Z, S1) ∧
hasInPort(Z, N))
```

Informally: For a specific MAC address, identify a pair of flow-rules that were installed in the flow-table of the same switch, that match packets based on the same criteria – that is, the same fields and the same values for those fields. The relevant fields are:

- *Ethernet source MAC address*
- *Input switch port.*

The second logic statement creates relationships between a MAC address and the flow-rules on a switch that match packets based on that MAC address, for use in inferring the HostFlowRuleGroup defined earlier.

Logic for identifying groups of ARP messages that are within `ARPTrustLimit` seconds of each other:

```
(∀G, ∃U1, ∃M, ∃Q1, ∃Q2, ∃R, ∃N1, ∃S, ∃U2, ∃N2, ∃G, ∃U3,
      ∃U4)(HostMacAddress(G) ∧ (U1 = {M : (hasHostMacARP(G,
      M)}) ∧ (M ∈ U1) ∧ Message.hostIPAddress(M, Q1) ∧
      Message.sourceIP(M, Q2) ∧ Message.sourcePort (M, N2) ∧
      Message.receivedSwitchPort(M, N1) ∧ (U2 = {E :
      (HostArpTemporalAdjacency(E))})) ∧ (G = Graph(U1, U2)) ∧
      (U3 = ConnectedComponents(G)) ∧ (U4 ∈ C) ∧
      getSwitchFromIPandTcpPort(Q2, N2, S) →
      (∃W)(HostArpGroup(W) ∧ HostArpGroup.hostIPAddress(W, Q1)
      ∧ hasSwitch(W, S) ∧ hasInPort(Q, N1) ∧
      hasHostMacArpGroup(G, W) ∧ hasHostMacArpSet(W, U4)))

(∀W, ∃U, ∃T1, ∃T2)(HostArpGroup(W) ∧ hasHostMacArpSet(W, U) ∧
      earliestInstant(U, T1) ∧ latestInstant(U, T2) →
      (∃G)(HostArpGroup.startpoint(T1) ∧
      (HostArpGroup.endpoint(T2 + ArpTrustLimit) ∧
      hasHostMacArpGroup(G, W) ∧ ¬hasIntermediateHostARP(G,
      T1, T2 + ARPTrustLimit) ∨ HostArpGroup.endpoint(T2))))
```

Informally: An ARP-group is a coherent group of ARP messages – i.e. where all messages in the group are from the same MAC and IP addresses, are received on the same switch port, where no ARP message from that MAC address was received on a different switch port for the duration of the timespan covered by the messages in the group, and there is no more than an operator-configurable 'n'-second gap (labelled the ARPTrustLimit) between each successive pair of ARP messages.

The second logic statement infers the start and endpoints for the ARP-Group.

The two logic statements above depend on the functions `getSwitchFromIPandTcpPort()` (defined earlier), `HostArpTemporalAdjacency()` and `hasIntermediateHostARP()`:

$$(\exists G, \exists M1, \exists M2, \neg\exists M3, \exists Q1, \exists Q2, \exists N, \exists S)(\text{HostMacAddress}(G) \wedge \text{hasHostMacArp}(G, M1) \wedge \text{Message.hostIPAddress}(M1, Q1) \wedge \text{Message.sourceIP}(M1, Q2) \wedge \text{Message.sourcePort}(M1, N) \wedge \text{Message.receivedSwitchPort}(M1, S) \wedge \text{hasHostMacArp}(G, M2) \wedge \text{Message.hostIPAddress}(M2, Q1) \wedge \text{Message.sourceIP}(M2, Q2) \wedge \text{Message.sourcePort}(M2, N) \wedge \text{Message.receivedSwitchPort}(M2, S) \wedge \text{isTemporallyAdjacent}(M1, M2) \wedge \text{hasHostMacArp}(G, M3) \wedge \text{earlierInstant}(M1, M3) \wedge \text{earlierInstant}(M3, M2) \rightarrow (\exists E)(\mathbf{HostArpTemporalAdjacency}(E) \wedge \text{hasHostArpTemporalAdjacency}(G, E) \wedge \text{hasArpMessage}(E, M1) \wedge \text{hasArpMessage}(E, M2)))$$

Informally: A HostArpTemporalAdjacency is inferred if two ARP messages for the same host are received on the same switchport within a specified interval of time.

$$(\exists G, \exists T1, \exists T2, \exists M, \exists T3)(\text{hasHostMacArp}(G, M) \wedge \text{Message}(M) \wedge \text{Message.existsAt}(M, T1) \wedge (T2 < T1) \wedge (T1 < T3) \rightarrow \mathbf{hasIntermediateHostARP}(G, T2, T3))$$

Informally: Return an ARP message for a specific host if it has a timestamp within an interval for which the start and endpoints are supplied.

`HostArpTemporalAdjacency()` depends on the function `isTemporallyAdjacent()`:

$$(\exists M1, \exists T1, \exists M2, \exists T2)(\text{Message}(M1) \wedge \text{Message.existsAt}(M1, T1) \wedge \text{Message}(M2) \wedge \text{Message.existsAt}(M2, T2) \wedge (M1 \neg= M2) \wedge (T2 - T1 < \text{ARPTrustLimit}) \rightarrow \mathbf{isTemporallyAdjacent}(M1, M2))$$

Informally: Are two message timestamps within `ARPTrustLimit` *number of seconds of each other?*

As can be construed from the number of logic statements required, Hosts require the most processing to infer, since they must be inferred indirectly from different sources of evidence of their activity.

5.5.2.6 Inference dependencies

The inference of any DCN element depends on the message ontology having been populated with message instances. Furthermore, the inference of some DCN

elements depends on the prior inference of other elements, as outlined above and as detailed in Table 5.4.

Table 5.4: Inference Dependencies

DCN Element	Depends On
Switch	Control-plane messages of any type, from switch to controller
Controller	Switch; control-plane messages of any type, from controller to switch
Port	Switch; control-plane messages (of type *MultipartReply*)
Inter-switch link	Port; control-plane messages (of type *PacketIn*)
Flow-rule	Switch; control-plane messages (of type *FlowMod*, *FlowRemoved*)
Flow-table	Switch; flow-rule
Host	Switch; flow-rule; control-plane messages (of type *PacketIn*, *MultipartReply*)
Host-link	Port; host

5.5.3 Representing Change Over Time in a DCN History

The DCN model requires the extra dimension of time, to represent how long a DCN element such as a switch was present in a network, the lifespan of each link connecting that switch to other switches, the duration for which each flow-rule is present in the flow-table of the switch, and so forth. Krieger's 4D approach [137] to representing change over time has been selected and employed in this work for the DCN History ontology, following the design pattern described in [135]. This is believed to be the first application of the 4D approach to a model of an SDN. Applying Krieger's 4D approach allows the re-use in the DCN History ontology of concepts from the atemporal Network Markup Language ontology.

When representing an instance of a switch, for example, some properties of the switch are constant, others change. 4D splits the representation into two classes:

- a perdurant class, with properties that are invariant, and
- a manifestation class, which can have multiple temporal instances each of which may have different values for its properties.

A property of an OpenFlow switch that does not change is its DataPath Identifier (DPID), hence that property belongs to the switch perdurant. Since the IP address of

a switch can change, and the TCP or UDP port from which the switch connects to a controller can change, those are properties of switch manifestations, as is the time interval for which the manifestation existed with unchanged values for those properties.

Temporal entities can be nested: A switch manifestation contains a flow-table. A flow-table contains flow-rules, of which each flow-rule has perdurant and manifestation parts. The priority and match-criteria of a flow-rule cannot change, but the other properties, e.g. instructions, timeouts and counters, can. The instructions and timeouts of an existing rule can be modified by a controller via a *FlowMod* message. The 4D representation of a switch, with nested 4D representation of a flow-rule, is illustrated in Figure 5.7. Omitted from the figure are controller- and host-related entities, properties and relationships, and abstract concepts that were added to the DCN History ontology to support queries (as described in Chapter 6).

The logic statements presented in section 5.5.2 define manifestations. For example, if a `Switch(S)` is inferred from a log messages, then `S` is an instance of a switch manifestation. From that switch instance, and any others that have the same value for `Switch.DPID(S, D)`, a switch perdurant is inferred with the following logic:

$$(\forall S, \exists D)(Switch(S) \wedge Switch.DPID(S, D) \rightarrow (\exists P)(\mathbf{SwitchPD}(P) \wedge SwitchPD.DPID(P, D) \wedge hasManifestation(P, S)))$$

$$(\exists P, \exists U, \exists S)(SwitchPD(P) \wedge (U = \{S : (hasManifestation(P, S)\}) \rightarrow (\exists T1, \exists T2)(earliestStartpoint(U, T1) \wedge latestEndpoint(U, T2) \wedge SwitchPD.startpoint(P, T1) \wedge SwitchPD.endpoint(P, T2)))$$

Informally: A switch perdurant is identified by the existence of a set of switch manifestations with the same DPID; the startpoint of the perdurant is the startpoint of the earliest manifestation; the endpoint of the perdurant is the endpoint of the latest manifestation.

The same pattern of logic is used to infer perdurants from instances of manifestations for `ControllerPD`, `HostPD`, and `FlowRulePD`:

$$(\forall C, \exists Q)\ (Controller(C) \wedge Controller.ipAddress(C, Q) \rightarrow (\exists P)(\mathbf{ControllerPD}(P) \wedge ControllerPD.ipAddress(P, A) \wedge hasManifestation(P, C)))$$

(∃P, ∃U, ∃S)(ControllerPD(P) ∧ (U = {C : (hasManifestation (P, C)}) → (∃T1, ∃T2)(earliestStartpoint(U, T1) ∧ latestEndpoint(U, T2) ∧ ControllerPD.startpoint(P, T1) ∧ ControllerPD.endpoint(P, T2)))

(∀H, ∃G)(Host(H) ∧ Host.macAddress(G) → (∃P)(**HostPD**(P) ∧ HostPD.macAddress(P, G) ∧ hasManifestation(P, H)))

(∃P, ∃U, ∃H)(HostPD(P) ∧ (U = {H : (hasManifestation (P, H)}) → (∃T1, ∃T2)(earliestStartpoint(U, T1) ∧ latestEndpoint(U, T2) ∧ HostPD.startpoint(P, T1) ∧ HostPD.endpoint(P, T2)))

(∀Z, ∃Z, ∃Y, ∃S, ∃D, ∃N, ∃M)(FlowRule(Z) ∧ hasFlowRule(Y, Z) ∧ hasFlowTable(S, Y) ∧ FlowRule.priority(Z, N) ∧ FlowRule.match(Z, A) → (∃P)(**FlowRulePD**(P) ∧ FlowRulePD.onSwitch(P, S) ∧ FlowRulePD.inFlowTable(P, Y) ∧ FlowRulePD.priority(P, N) ∧ FlowRulePD.match(P, A) ∧ hasManifestation(P, Z)))

(∃P, ∃U, ∃S)(FlowRulePD(P) ∧ (U = {Z : (hasManifestation(P, Z)}) → (∃T1, ∃T2)(earliestStartpoint(U, T1) ∧ latestEndpoint(U, T2) ∧ FlowRulePD.startpoint(P, T1) ∧ FlowRulePD.endpoint(P, T2)))

5.5.4 Implementation

The workflow that was followed for implementation is shown in Figure 5.8. The tools and process used at each numbered step in the workflow are as follows:

Step 1. Creation of new ontologies with Protégé [138]. This ontology authoring tool is widely used in the ontology development community, is stable, and is being actively developed to support evolving standards. Authoring an ontology is a significant undertaking that entails the definition of:

1. Relevant classes and a class hierarchy,
2. properties of objects - i.e. relationships between instances of classes,
3. data properties - specifying what information can be associated with an instance of an object, and
4. datatypes - so that the values pointed to by the data properties can be stored and processed.

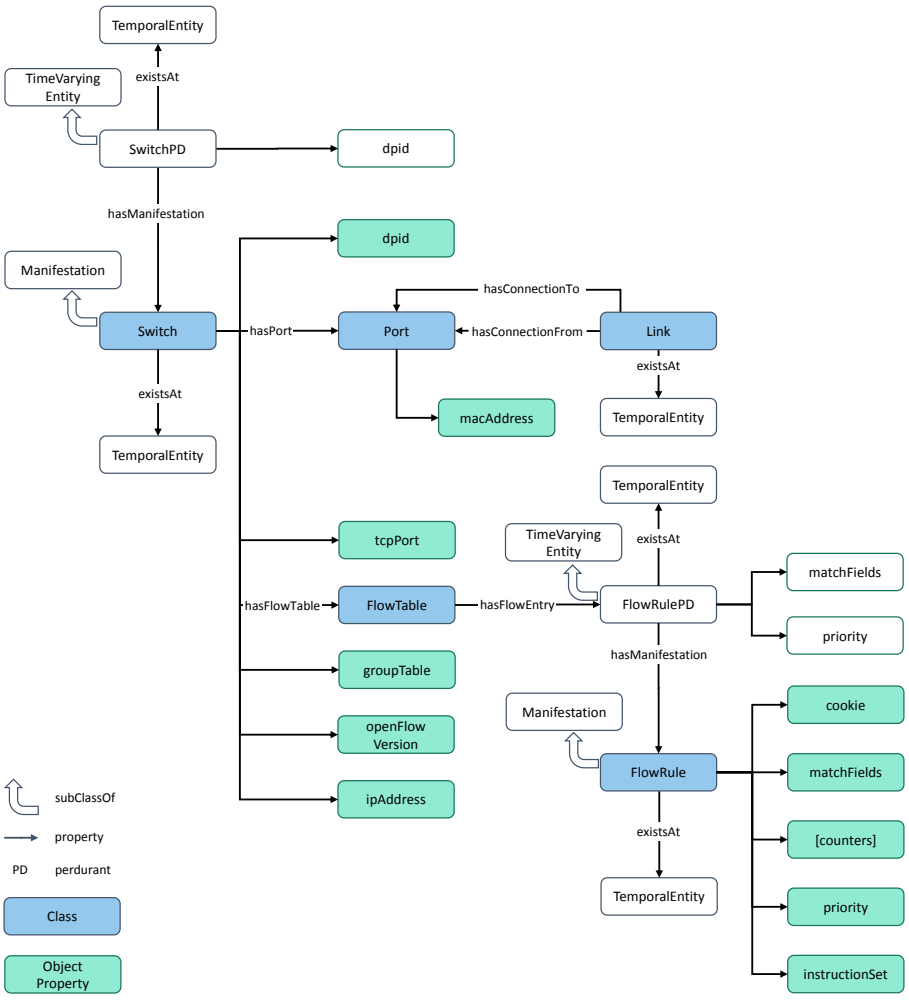


Figure 5.7: DCN History Ontology (Partial)

Step 2. Export of the authored ontologies from Protégé as OWL (Web Ontology Language) files expressed in Turtle syntax, followed by import of the Turtle files into AllegroGraph [139]. AllegroGraph is a graph database, used in this work for its triple-store capabilities. While Protégé can act as a triple-store using in-memory storage, or combined with a backend database to store and access large amounts of data, AllegroGraph is designed to be high performance and massively scalable –

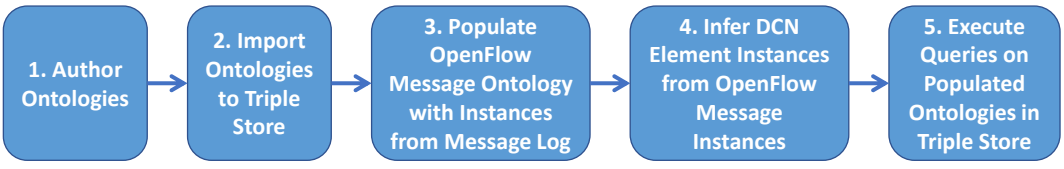


Figure 5.8: Workflow Diagram

properties essential for handling the quantity of data extracted and inferred from the control-plane message log of a multitenant DCN.

Step 3. Population of the message ontology in AllegroGraph with instances based on data read from the OpenFlow message log stored in ElasticSearch [93] by LogSnap. The population logic was coded in Python since both AllegroGraph and ElasticSearch provide Python APIs. ElasticSearch stores full OpenFlow message details, from which key data is extracted and written to AllegroGraph to populate the ontology with the triples to represent instances of message types and properties defined in the OpenFlow Message ontology.

Step 4. Inference of new temporal instances of network elements from OpenFlow messages. Inference modules were written in AllegroCL, a Common Lisp implementation that natively interfaces with AllegroGraph, and Allegro Prolog, a Prolog implementation embedded in AllegroCL. Where possible, the inference logic was encoded as Prolog rules as it lends itself to direct translation of logic rules. However Allegro Prolog is not a full Prolog implementation so additional Lisp code provided the required inferencing functionality. Each type of DCN element required a separate inference module to be written, although code was re-used between modules were possible.

Step 5. Application of verification queries to populated ontologies. The queries were encoded in Prolog and SPARQL [126], and executed from within Lisp modules. The Lisp modules or SPARQL queries were called from Python scripts if the results were to be charted. SPARQL is a W3C-standard semantic query language supported by AllegroGraph and accessible via Lisp and Python. Experience has shown that SPARQL allowed the writing of more expressive queries than Prolog, although the SPARQL queries were not readily composable unlike those coded in Prolog. Where appropriate, results were visualised with Plotly [126], a Python charting package.

Steps 1 to 5 can be repeated when necessary:

- If an ontology is changed or added to (Step 1), then the subsequent 4 steps should be repeated to produce new results based on the updated ontology.

- If a new or updated message log is available, then Step 3 onwards should be re-done.
- If the inference modules are modified, the restart point is Step 4.
- Step 5 can be repeated indefinitely, and if new queries are created based on existing ontology concepts and relationships, previous steps do not need to be re-done.

5.5.4.1 Inferring DCN Elements – Implementation Issues Overcome

In 5.5.2.2, it was stated that the lifespan of a controller instance could be inferred from a set of sessions between switches and the controller that overlap in time. The task of identifying sets of overlapping sessions was recognised as lending itself to being formulated as a graph problem. The benefit of this implementation decision was that it provided a functional approach that can be re-used for the inference of other temporal relationships.

The nodes of the graph correspond to switch-controller sessions. An edge connects two nodes if the two corresponding sessions overlap in time and the identity of the controller is the same for the two sessions (matching IP addresses and port numbers). A group of nodes is a 'connected-component' if all nodes in the group have at least one path to every other node in the group. Each connected-component of the graph contains the nodes corresponding to the overlapping switch-controller sessions that represent the lifespan of the controller instance. The start point of the controller instance is the earliest session start point in the set, and the end point of the controller instance is the latest session end point in the set. The 'graph' library available via the Quicklisp [140] library manager provides the required 'connected-components' function.

Figure 5.4, shown earlier in this chapter, illustrates how each identified connected-components group of switch-initiated sessions with a controller relates to, and can be used to infer, a controller instance.

The logic for associating flow-rules specific flow-tables on a switch was presented earlier. For simplicity, the implementation currently assumes one flow-table per switch. Furthermore, it is assumed for the implementation that controllers will request to be notified when a flow-rule is removed, as can be specified in the *FlowMod* request sent to first add or to modify the flow-rule.

5.6 Evaluation

The objective of the evaluation was to verify that the inferred model of a DCN topology matches the original network and tracks the changes in the original network accurately. This was assessed through experiments using different DCN topologies:

- A traditional multi-tier topology (Figure 5.9), as used historically in data-centres [27]. This consisted of:
    - 3 layers of switches – 2 aggregation switches, 4 edge switches and 6 top-of-rack switches, and
    - 20 hosts connected to each top-of-rack switch.
- A leaf-spine topology (Figure 5.12), as used in various forms in modern multi-tenant data-centres [35]. This consisted of:
    - leaf switches and 6 spine switches, connected in a folded-clos arrangement, and
    - 20 hosts connected to each leaf switch.

Each network topology was constructed in an emulated environment using MaxiNet [141], which extends the Mininet [96] emulator to support larger networks. Experiments were also performed with simple-tree and fat-tree topologies. Those are not discussed further here, however results were similar to the topologies that are discussed.

The message logs are from LogSnap experiments where light traffic was generated (using the DCT2Gen / TrafficGen utility described in Chapter 3) on the relevant DCN

for two minutes. Traffic was required in order to cause flow-rules to be added to flow-tables, but the traffic level was kept light to make verification unambiguous. Logs from exercises with heavy traffic levels were used in later, non-verification experiments. The light traffic profile specified 10MB TCP transfers between pairs of hosts scheduled to start one per second, mostly rack-local (75%) with some inter-rack traffic (25%) to make use of longer paths. The 75/25 split reflects observed Cloud DCN traffic distributions [113]. Each host participated in at least one TCP transfer.

For each message log individually, the ontologies were populated with message instances – step 3 in the workflow (Figure 5.8) – and the rest of the steps then followed. A Python script was run on the populated ontologies to extract details of the inferred switch, host and link instances, and to chart them. Figure 5.10 shows a composite inferred topology for the multi-tier DCN, Figure 5.11 shows the equivalent for the spine-leaf DCN. Each chart is 'composite' because it is the result of a query for all DCN elements inferred within the time range covered by the message log, even if these were not present for the full duration, or even at the same time. For a DCN operator, the time range can be narrowed to view the topology as it was over shorter windows of time.

In addition to the message logs, periodic dumps were taken of flow-table contents from the DCN switches on the original networks. By querying the populated ontologies for the flow-rules that were inferred to be present at the time of each periodic dump, and comparing the two sets of flow-rules, it was verified that both matched.

5.6.1 Expressing DCN History Queries for Verification

The three verification queries presented in Table I are expressed in second-order logic below, and in terms of the ontological concepts discussed in section 5.5.2.

Query #1 'What switches / hosts / controllers were active in the network?' is expressed as:

$$(\exists S, \exists C, \exists H)(SwitchPD(S) \vee ControllerPD(C) \vee HostPD(H))$$

Query #2 'For each inferred switch, show the durations for which the switch was active in the network' is expressed as:

```
(∃S, ∃D, ∃T1, ∃T2)(Switch(S) ∨ Switch.dpid(D) ∨ ,
∃T1Switch.startpoint(T1) ∨ Switch.endpoint(T2))
```

For this query, the switch DPID is retrieved in addition to the start and end points, since the DPID is the real-world identifier of the physical switch that is logically represented in the ontology.

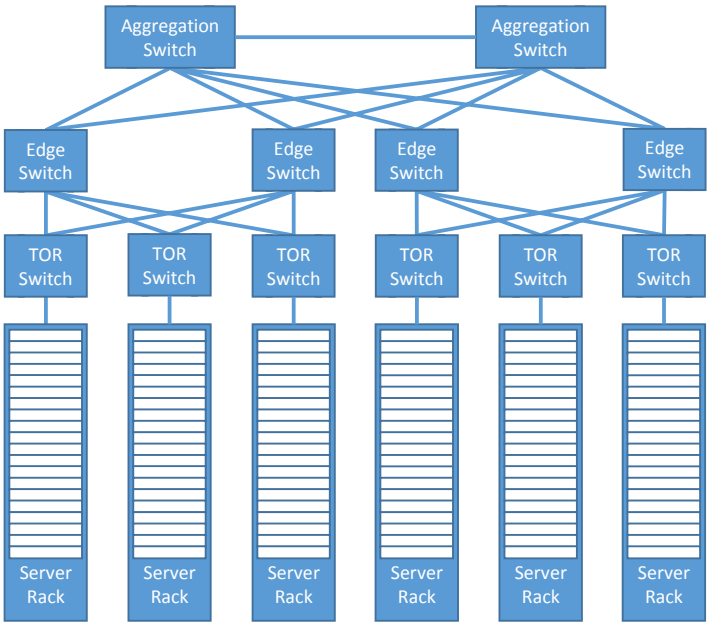


Figure 5.9: Logical Topology for Original Multi-Tier DCN

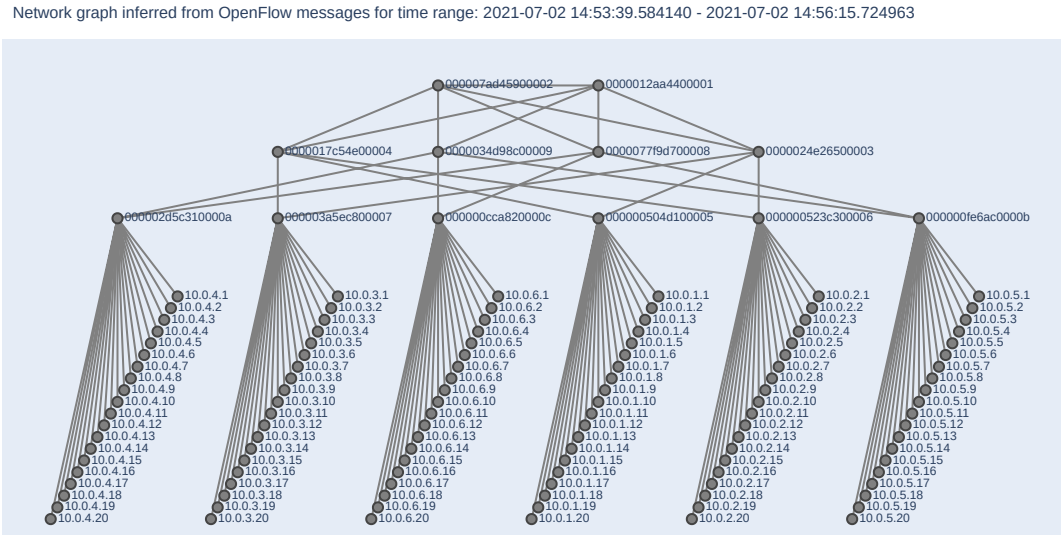


Figure 5.10: Composite Inferred Topology for Multi-Tier DCN

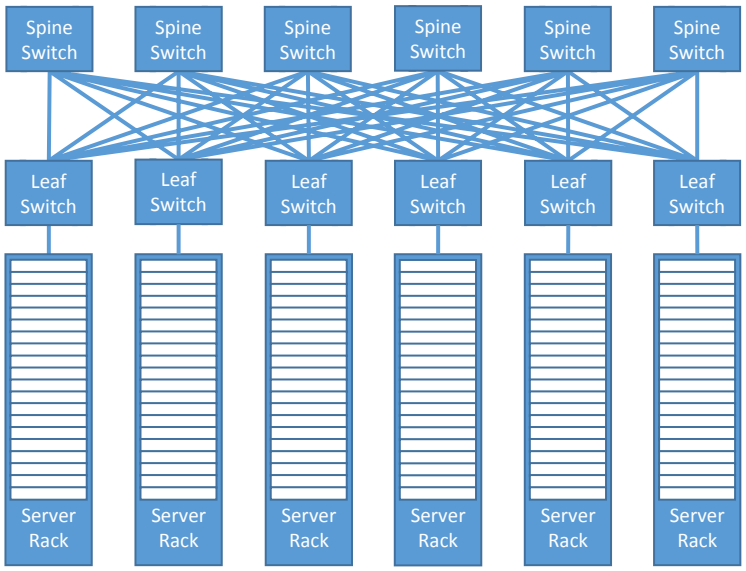


Figure 5.12: Logical Topology for Original Spine-Leaf DCN

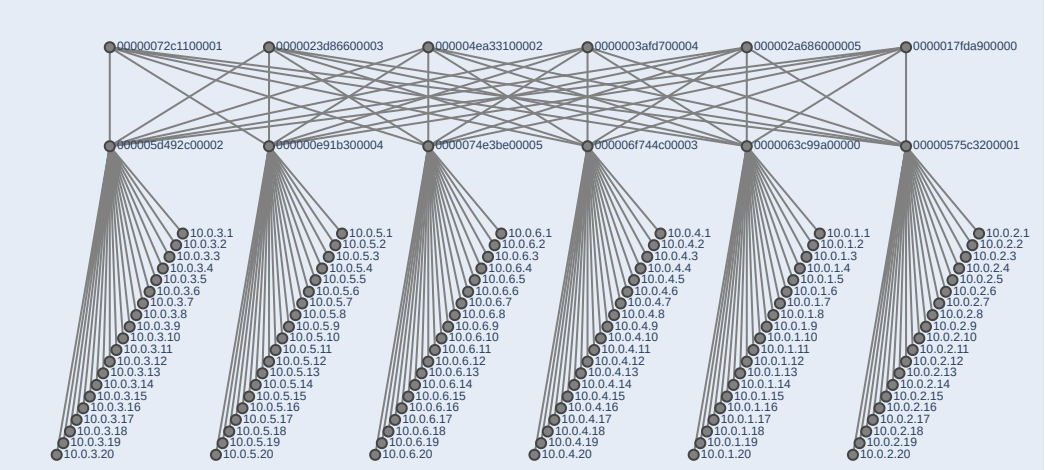


Figure 5.11: Composite Inferred Topology for Spine-Leaf DCN

Query #3 'For each inferred host, show the durations for which flow-rules were installed in switch flow-tables' is expressed as:

```
(∃H, ∃Q, ∃A, ∃S, ∃Y, ∃Z, ∃T1, ∃T2, ∃T3, ∃T4)(Host(H) ∧
Host.ipAddress(H,Q) ∧ Host.macAddress(H,A) ∧
Host.startpoint(H,T1) ∧ Host.endpoint(H,T2) ∧ Switch(S) ∧
hasFlowTable(S,Y) ∧ hasFlowRule(Y,Z) ∧ hasMatch(Z, M) ∧
Match.macAddress(M, A) ∧ FlowRule.startpoint(Z, T3) ∧
FlowRule.endpoint(Z, T4))
```

For query #3, the host IP address is retrieved because that is the host identifier that a DCN operator is most likely to use, and switch DPIDs to identify the switches on which relevant flow-rules were configured.

The three formally-expressed queries were encoded for application against the populated ontological model.

dcnhistory:controller1	dcnhistory:host119	dcnhistory:host31	dcnhistory:host52	dcnhistory:host73	dcnhistory:host94
dcnhistory:host1	dcnhistory:host12	dcnhistory:host32	dcnhistory:host53	dcnhistory:host74	dcnhistory:host95
dcnhistory:host10	dcnhistory:host120	dcnhistory:host33	dcnhistory:host54	dcnhistory:host75	dcnhistory:host96
dcnhistory:host100	dcnhistory:host13	dcnhistory:host34	dcnhistory:host55	dcnhistory:host76	dcnhistory:host97
dcnhistory:host101	dcnhistory:host14	dcnhistory:host35	dcnhistory:host56	dcnhistory:host77	dcnhistory:host98
dcnhistory:host102	dcnhistory:host15	dcnhistory:host36	dcnhistory:host57	dcnhistory:host78	dcnhistory:host99
dcnhistory:host103	dcnhistory:host16	dcnhistory:host37	dcnhistory:host58	dcnhistory:host79	dcnhistory:switch1
dcnhistory:host104	dcnhistory:host17	dcnhistory:host38	dcnhistory:host59	dcnhistory:host8	dcnhistory:switch10
dcnhistory:host105	dcnhistory:host18	dcnhistory:host39	dcnhistory:host6	dcnhistory:host80	dcnhistory:switch11
dcnhistory:host106	dcnhistory:host19	dcnhistory:host4	dcnhistory:host60	dcnhistory:host81	dcnhistory:switch12
dcnhistory:host107	dcnhistory:host2	dcnhistory:host40	dcnhistory:host61	dcnhistory:host82	dcnhistory:switch2
dcnhistory:host108	dcnhistory:host20	dcnhistory:host41	dcnhistory:host62	dcnhistory:host83	dcnhistory:switch3
dcnhistory:host109	dcnhistory:host21	dcnhistory:host42	dcnhistory:host63	dcnhistory:host84	dcnhistory:switch4
dcnhistory:host11	dcnhistory:host22	dcnhistory:host43	dcnhistory:host64	dcnhistory:host85	dcnhistory:switch5
dcnhistory:host110	dcnhistory:host23	dcnhistory:host44	dcnhistory:host65	dcnhistory:host86	dcnhistory:switch6
dcnhistory:host111	dcnhistory:host24	dcnhistory:host45	dcnhistory:host66	dcnhistory:host87	dcnhistory:switch7
dcnhistory:host112	dcnhistory:host25	dcnhistory:host46	dcnhistory:host67	dcnhistory:host88	dcnhistory:switch8
dcnhistory:host113	dcnhistory:host26	dcnhistory:host47	dcnhistory:host68	dcnhistory:host89	dcnhistory:switch9
dcnhistory:host114	dcnhistory:host27	dcnhistory:host48	dcnhistory:host69	dcnhistory:host9	
dcnhistory:host115	dcnhistory:host28	dcnhistory:host49	dcnhistory:host7	dcnhistory:host90	
dcnhistory:host116	dcnhistory:host29	dcnhistory:host5	dcnhistory:host70	dcnhistory:host91	
dcnhistory:host117	dcnhistory:host3	dcnhistory:host50	dcnhistory:host71	dcnhistory:host92	
dcnhistory:host118	dcnhistory:host30	dcnhistory:host51	dcnhistory:host72	dcnhistory:host93	

Listing 5.1: Inferred Active Switches / Hosts / Controllers for Spine-Leaf DCN Topology

Query #1 was applied to a triple store containing an OpenFlow Message ontology populated based on a message log captured from the leaf-spine topology, and the output is shown in Listing 5.1. Each inferred element is named according to its type and numbered according to the order in which each instance of that type was inferred. The prefix ‘dcnhistory:’ represents the URI (Universal Resource Identifier) of the DCN History ontology. The results show that 1 SDN controller, 120 hosts and 12 switches were inferred. Identical results were obtained on the multi-tier topology, as expected since the same numbers of controllers, hosts and switches were present in that topology. The results verify that these active DCN elements were successfully inferred, and match observations on the original network.

The output of Query #2 provides detail regarding the lifespan of inferred switches. Query #2 output for the multi-tier DCN topology is shown in Table 5.5. In terms of

the DCN History ontology, each DPID in the first column represents a single switch perdurant. For each perdurant, a single switch manifestation was detected, the lifespan of which is represented as time deltas from the startpoint of the first inferred switch manifestation (which is for the switch with DPID 000002d5c310000a). Query #2 output for the spine-leaf DCN topology is shown in tabular form in Table 5.6.

Figure 5.13 charts the lifespans for the inferred switch instances for the multi-tier DCN topology – the data in Table 5.5. Figure 5.14 charts the equivalent for the spine-leaf DCN. The chart format was produced automatically with the query output as a visual aid to interpreting the results. For example, it can be clearly seen

Table 5.5: Lifespan of Inferred Switches for Multi-Tier DCN Topology

Dpid	start-delta	end-delta
000003a5ec800007	00:00:07.472109	00:02:39.463248
000000fe6ac0000b	00:00:07.469785	00:02:39.419909
000007ad45900002	00:00:03.991582	00:02:38.537373
000000504d100005	00:00:03.990203	00:02:38.494152
0000017c54e00004	00:00:03.987998	00:02:38.689693
0000077f9d700008	00:00:03.981904	00:02:38.444722
0000024e26500003	00:00:03.980221	00:02:38.418217
0000012aa4400001	00:00:03.977042	00:02:38.387741
000000523c300006	00:00:03.975524	00:02:38.361932
000000cca820000c	00:00:00.013891	00:02:36.867776
0000034d98c00009	00:00:00.004485	00:02:36.840698
000002d5c310000a	00:00:00	00:02:36.821114

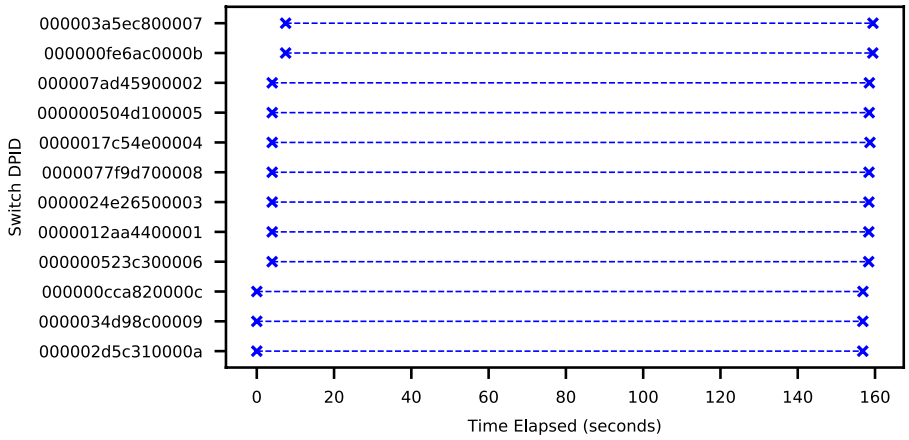


Figure 5.13: Inferred Switch Instance Lifespans for Multi-Tier DCN Topology

Table 5.6: Lifespan of Inferred Switches for Spine-Leaf DCN Topology

dpid	start-delta	end-delta
00000072c1100001	00:00:06.929260	00:02:39.077462
0000063c99a00000	00:00:06.923770	00:02:39.031460
0000017fda900000	00:00:06.921409	00:02:39.010429
000000e91b300004	00:00:06.917377	00:02:38.984109
0000023d86600003	00:00:02.949286	00:02:37.871078
000004ea33100002	00:00:02.942012	00:02:37.839192
00000575c3200001	00:00:02.937575	00:02:37.808328
000005d492c00002	00:00:02.935611	00:02:37.790135
0000074e3be00005	00:00:00.000503	00:02:36.759437
000002a686000005	00:00:00.000331	00:02:36.725874
000006f744c00003	00:00:00.000173	00:02:36.695941
0000003afd700004	00:00:00	00:02:36.723600

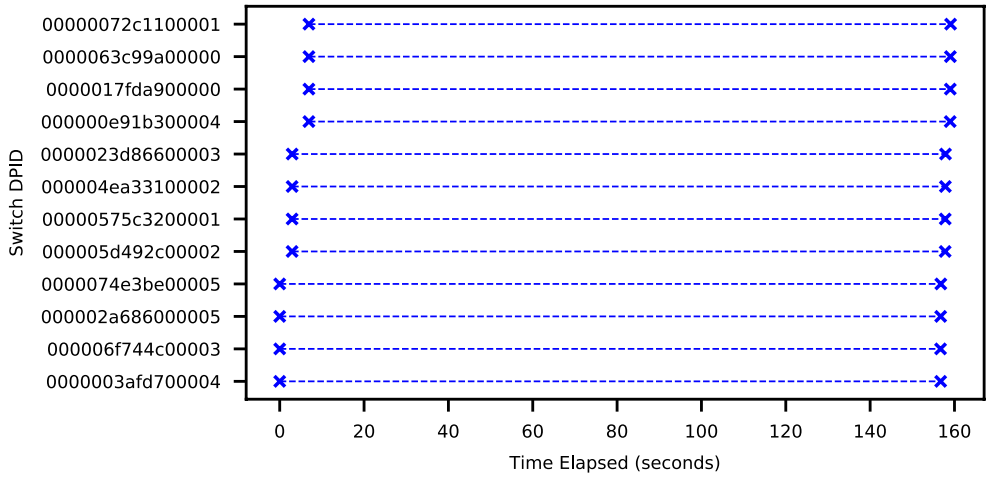


Figure 5.14: Inferred Switch Instance Lifespans for Spine-Leaf DCN Topology

in both charts that the switches did not all connect to their respective controller at exactly the same time, and that each switch had a single instance with an uninterrupted lifespan. This corresponds with observations on the original DCNs from which the message logs were captured.

The output of Query #3 allows comparison between the observed activity of hosts and the durations for which flow-rules that supported that activity were present in switch flow-tables. Figure 5.15 shows the inferred lifespan of the host with IP address 10.0.2.13, charted against the lifespan of the flow-rules that were configured on switches in the DCN and that matched packets sent by that host. The message log on which the inferences are based is from a Multi-Tier DCN topology.

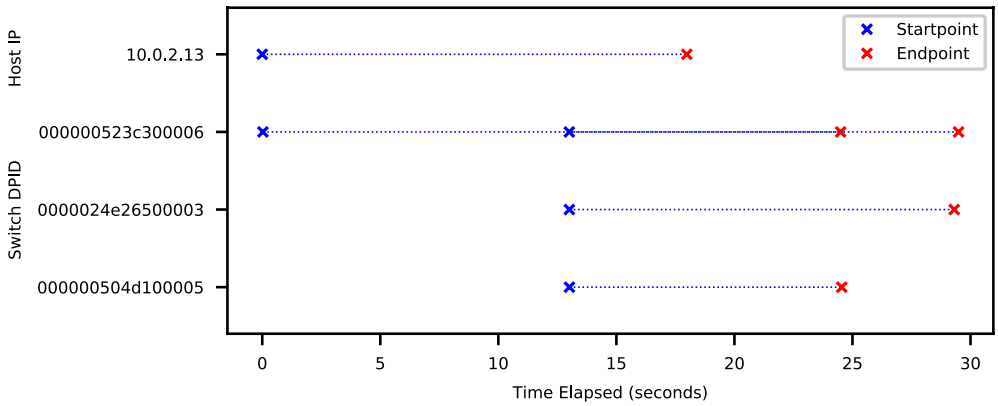


Figure 5.15: Host Lifespan with Lifespan of Flow-Rules Matching Host as Source of Packets (Multi-Tree Topology)

It can be seen that two flow-rules were configured on the switch with DPID '000000523c300006'. The first flow-rule was installed when the host initially communicated with another host. It is not shown which host initiated the communication, and the second host is not identified because the query did not request that information. The fact that no rules were installed on other switches in conjunction with this rule is an indication that the two communicating hosts were connected to the same switch (intra-rack communication).

The second flow-rule configured on the switch with DPID '000000523c300006' was installed at approximately the same time as a rule on two other switches. This indicates that host 10.0.2.13 participated in another communication session and that this was with a host connected to a different switch (inter-rack communication).

The four rules were removed sometime after host 10.0.2.13 was no longer actively communicating. This behaviour is expected: an SDN controller will normally explicitly remove a flow-rule after receiving statistics reports that indicate the rule has not matched packets for a period of time.

Figure 5.16 shows the equivalent chart generated from the output of the same query, but applied to DCN elements inferred from a message log captured on a Leaf-Spine topology DCN. The chart is very similar to that from a Multi-Tier DCN because the test traffic was generated using the same traffic profile described earlier in this section.

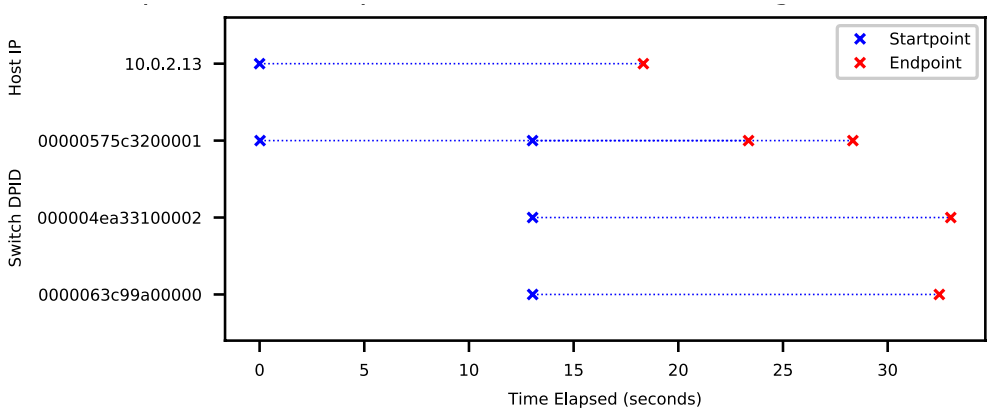


Figure 5.16: Host Lifespan with Lifespan of Flow-Rules Matching Host as Source of Packets (Spine-Leaf Topology)

In fact, the traffic profile that was used in the experiments that produced the message logs for both topologies specified three flows to which host 10.0.2.13 was a party. The first flow was from host 10.0.2.3 to 10.0.2.13, and the second was from host 10.0.2.13 to 10.0.2.3 starting 8 seconds after the first flow. Both flows were carried over TCP connections, therefore flow-rules were installed to carry packets in both directions for the first flow: data packets in one direction, and acknowledgements in the opposite direction. The flow-rules were still present to carry the second flow.

The third flow was from host 10.0.1.13 to 10.0.2.13, and was timed to start 13 seconds after the first flow. The charts confirm that this is what happened.

5.6.2 Discussion of Evaluation, Queries and Results

The initial evaluation confirmed that all targeted elements of the original DCN were successfully inferred from the control-plane message log, and used to populate an ontological model of the network. The elements were controllers, switches, hosts, inter-switch links, host-links, flow-tables and flow-rules. The inferred instances of the elements matched observations on the original network. Timepoints and time-ranges of the inferred instances relate directly to the timestamps of the logged OpenFlow messages from which they were inferred. The accuracy of the message logging process was evaluated in Chapter 3. The approach taken has assessed the correctness of inferencing within the limitations of the experimental environment as described.

Three queries, ranging from simple to complex, were selected to demonstrate the process of query development, and the extraction of useful and insightful information from the populated ontologies produced for this work. Moreover, the query results are further verification that the ontological approach results in a temporal and topological model of the network from which information can be extracted that is accurate when compared with the network it models. The range of queries was selected to evaluate a critical set of network operations. The queries were expressed informally in English, translated to second-order logic, and then encoded in the appropriate programming or query language. The results of applying the queries to populated ontologies were presented. The temporal dimension to the extracted information is of particular note.

5.7 Conclusion

The challenge addressed in this chapter was to describe a method to create a temporal and topological model of a DCN. The model was required to represent key elements of a software-defined DCN, and lend itself to being populated with historical data extracted from a control-plane message log to represent the state of a specific DCN and how that state changed over time.

An ontological approach was selected to create the model. Two ontologies were designed and authored: an OpenFlow Message ontology, and a DCN History ontology. The OpenFlow Message ontology was populated from a message log, and instances of DCN elements were then inferred from the message instances to populate the DCN History ontology. These new ontologies are believed to be the first temporal ontologies designed and used to represent a software-defined DCN, and contain the first application of Krieger's 4D approach to representing change over time in a model of a software-defined network.

A set of verification tests were devised and implemented to firstly confirm the accuracy of the inferred DCN elements, and to secondly demonstrate that sophisticated queries could be applied to the populated model to return useful and

insightful information to the DCN operator. The results of the tests were listed or charted as appropriate.

The DCN model that was created can be re-populated using log files of control-plane messages captured from different networks or at different times. Furthermore the model lends itself to the development of more complex queries intended to gain deeper insight into the network, as is discussed and demonstrated in Chapter 6.

6 Resolution of Sophisticated Operator Queries on a DCN

6.1 Introduction

Chapter 5 presented the ontological approach selected for creating a topological and temporal model of a Data Centre Network (DCN). The purpose of the model is to support the resolution of operator queries. Chapter 5 illustrated the results of fundamental queries used to verify the correctness of the model. In this chapter, more sophisticated queries are considered, intended to extract information providing deeper insights for the DCN operator. The methodology for encoding those queries is described, and the results shown of applying the queries against the ontological model populated from log files of captured control-plane data.

Table 6.1 is a list of queries, selected because the answers can provide useful insight into the operation of a DCN. The queries are intentionally of differing levels of sophistication, providing a variety of challenges to their resolution and to the task of devising a methodology for encoding and resolving other queries. The sample queries are discussed individually in section 6.2, and used to illustrate points made in relation to the methodology in later sections.

Table 6.1: List of Sample Queries

#	Query
1	For what hosts were flow-rules installed in switch flow-tables, but for which no packets were observed in the DCN?
2	What (if any) hosts migrated from one part of the network to another, and what were the details (when, from where, to where)?
3	What VPLS tunnels were configured on the network, and what were the details?
4	What path would a specified packet follow through the network at a particular time?

6.2 Analysis of sample queries

6.2.1 Query 1 - For what hosts were flow-rules installed in switch tables, but for which no packets were observed in the DCN?

An answer to Query 1 reveals situations where network policy has resulted in flow-rules being installed in switch flow-tables to handle flows involving a specific host, but packets did not match against those rules. There are several reasons why this might be the case, some benign, others more concerning:

- A benign reason is that the rules were installed to establish a backup path for packets of a flow in case the primary path fails.
- A concerning reason is if the host is not present on the network. This is concerning because a flow-rule has been unnecessarily installed in a switch flow-table, contributing to increased flow-table occupancy, which reduces the performance of flow-table management [83]. Moreover, since flow table capacity is limited, installing unnecessary rules increases the likelihood of flow-table overflow. Flow-table overflow can result in the controller being unable to install new flow-rules on the switch.

In the latter case, a DCN operator can use the results of Query 1 to investigate further. In a multi-tenant data centre environment, the flow-rules in question may be present due to a tenant self-service configuration of network policy, and the DCN operator could debate the requirement for that configuration with the tenant in question.

Query 1 is challenging to answer, because the information to answer the query is generally not easily accessible to the DCN operator. It could be obtained by online polling of switches to see whether installed flow-rules matched any packets before they expired or were removed. The work on anomaly detection (including locating unmatched rules) in [142] depends on analysis of entire flow-table contents that have periodically been 'dumped' from switches. Polling or dumping incurs the overhead of CPU time on the switch to prepare the information and traffic on the

network to transfer it. However, the information is in a set of control-plane messages captured with LogSnap, lending itself to extraction from the message log into the populated ontologies to be queried.

Using the ontological model created and populated as described in Chapter 5, Query 1 was quickly constructed as a search for inferred flow-rules whose statistics show that they did not match any packets, and a comparison between the match criteria of those inferred flow-rules and the identities of inferred hosts.

6.2.2 Query 2 - What (if any) hosts migrated from one part of the network to another, and what were the details (when, from where, to where)?

Query 2 is relevant in two scenarios in a DCN:

1. Where virtual hosts can migrate from one physical server to another, a feature that is used for balancing resource usage between servers, or to free up a server so that it can be taken out of service for maintenance, for example.
2. A physical server could be disconnected from one DCN switch by a technician and reconnected to another.

In either case, a DCN operator will be interested to know where, when, and how often migrations occurred, and can then follow up on why they happened.

The information to answer the query may be available in VM management software logs, if those are kept, or in manual records of network maintenance work. Neither is guaranteed to be available, and searching either set of records will be time consuming. The control-plane message logs used in this work provide a single source for the information, and it will be seen that a carefully constructed query extracts the information from a populated ontology for a network operator.

6.2.3 Query 3 - What VPLS tunnels were configured on the network, and what were the details?

VPLS (Virtual Private LAN service) is a facility provided in DCNs (and also used across Wide Area Networks) where multipoint-to-multipoint tunnels are established to emulate private LAN connectivity between multiple points. A VPLS isolates the traffic of a data centre tenant from other tenants, avoiding potential problems due to addressing conflicts between virtual hosts belong to different tenants, for example. In a software-defined DCN, VPLS tunnels are configured programmatically. An OpenFlow controller installs rules in switches that label packets at entry points to tunnels, remove labels at exit points, and forward packets within the tunnels based on their labels.

The policy defined by the DCN operator, and implemented by the Software-Defined Networking (SDN) controller, may simply specify the entry and exit points for a VPLS instance that should be present, or may include desired times at which it should be present. OpenFlow message logs are a record of the requests from the controller to install the flow rules in switch flow tables to implement the VPLS tunnels according to the policy, and the statistics from switches proving what tunnels were used to forward traffic. It would be challenging to manually find the requests and responses, and to interpret the details to verify that the policy was in fact implemented as required, or possibly that it was partially or incorrectly implemented. The ontological approach used here makes the task more achievable with far less effort, as demonstrated in section 6.4.2.

Query 3 identifies the lifespan of VPLS tunnels, where the tunnel entry and exit points were, and the switches that were part of each tunnel. A DCN operator can use this information to verify that VPLS tunnels were present when they were required and, with knowledge of the network policy, should be able to relate the tunnels to the tenant for whose traffic they were configured.

6.2.4 Query 4 - What path would a specified packet follow through the network at a particular time?

Query 4 reveals the path through the network on which a packet injected in the network at a specific time would have been forwarded to the destination chosen for it by switches according to their configured rules. The packet needs to be described – i.e., values must be provided for header fields in the packet. A destination MAC address suggests the intended destination, however the actual destination discovered by following the sequence of matching installed flow-rules may be different. Where the path leads to the intended destination, it may still not be the expected path – for example, possibly following a non-optimal route.

The ability to identify the path a packet would have followed at a specified point in time is important for compliance, and to show that packets containing a data-centre tenant's data could not have strayed outside the boundaries of that tenant's resources other than to destinations permitted by the tenant. If a packet somehow does end up where it should not be, knowing the path allows the DCN operator to investigate further to find the reason that an unexpected path was configured.

Answering Query 4 requires detailed knowledge of what flow-rules were configured on every DCN switch at the time specified in the query, and what links were operational in the network at that time. Strictly speaking, it only requires knowledge of the flow-rules configured on the DCN switches along the path to the eventual destination, and the state of the links in that path – however the path is not known in advance. In any case, the complete set of information is not normally recorded, however it is present in the ontological model and is extracted with some additional inferencing and querying, as shown below.

6.3 Research Challenges

The overall challenge addressed in this chapter is the provision of a methodology for encoding queries to apply them against the populated ontological model – to extract useful and accurate information. This is a challenge because the questions that DCN operators have are about what happened in the physical networks that

they built and operate. The terms in which they express their questions are not limited to the concepts and relationships that have, so far, been defined in the message or history ontologies.

Developing and encoding a complex query about the past history of a DCN requires careful expression of temporal parameters, and may require new concepts to be defined in the ontologies. Clearly, when a new concept needs to be defined, it must be in terms of existing ontological concepts, however the relationship between new and existing concepts may not be readily apparent. New concepts should be constructed in such a way as to lend themselves to being the basis for further concepts defined later, as well as to being useful as targets for other new queries. Moreover, each addition to the ontologies should be validated.

As outlined in Chapter 5, other researchers have used ontological approaches [79, 120, 143, 144] for SDN-related work. By their nature, ontologies are extensible – a fact that is often cited in support of taking an ontological approach.

However, an ontology is usually only part of the output of the researchers' work, and something that is missing from their output is a description of, or methodology for, how to apply such an extension in the context of adding functionality to their work.

The queries listed in Table 6.1, and discussed through this chapter, have informed the process of addressing the research challenge of providing a methodology for extension of the range of queries available to DCN operators, and are used to illustrate the methodology presented.

6.4 A Methodology for DCN Ontological Concept and Query Design

To address the research challenge, a set of methods is required that describes how to:

1. Add new concepts to the ontologies, to answer new queries.
2. Implement the logic to infer instances of the new concepts.

3. Add new queries to provide insight for operators.
4. Validate instances of new concepts.

Clearly, the process is query-driven: once it is recognised that some useful set of information needs to be extracted from the ontology, that requirement is framed as a query. With careful phrasing of the query to make use of existing concepts and relationships where available, it is then apparent what (if any) new concepts need to be added to the ontologies to allow the query to be resolved. When new concepts are defined, instances of those concepts must be inferred and validated, and the query encoded to extract the relevant information from the populated ontology. The process is presented diagrammatically in Figure 6.1, and expanded upon below.

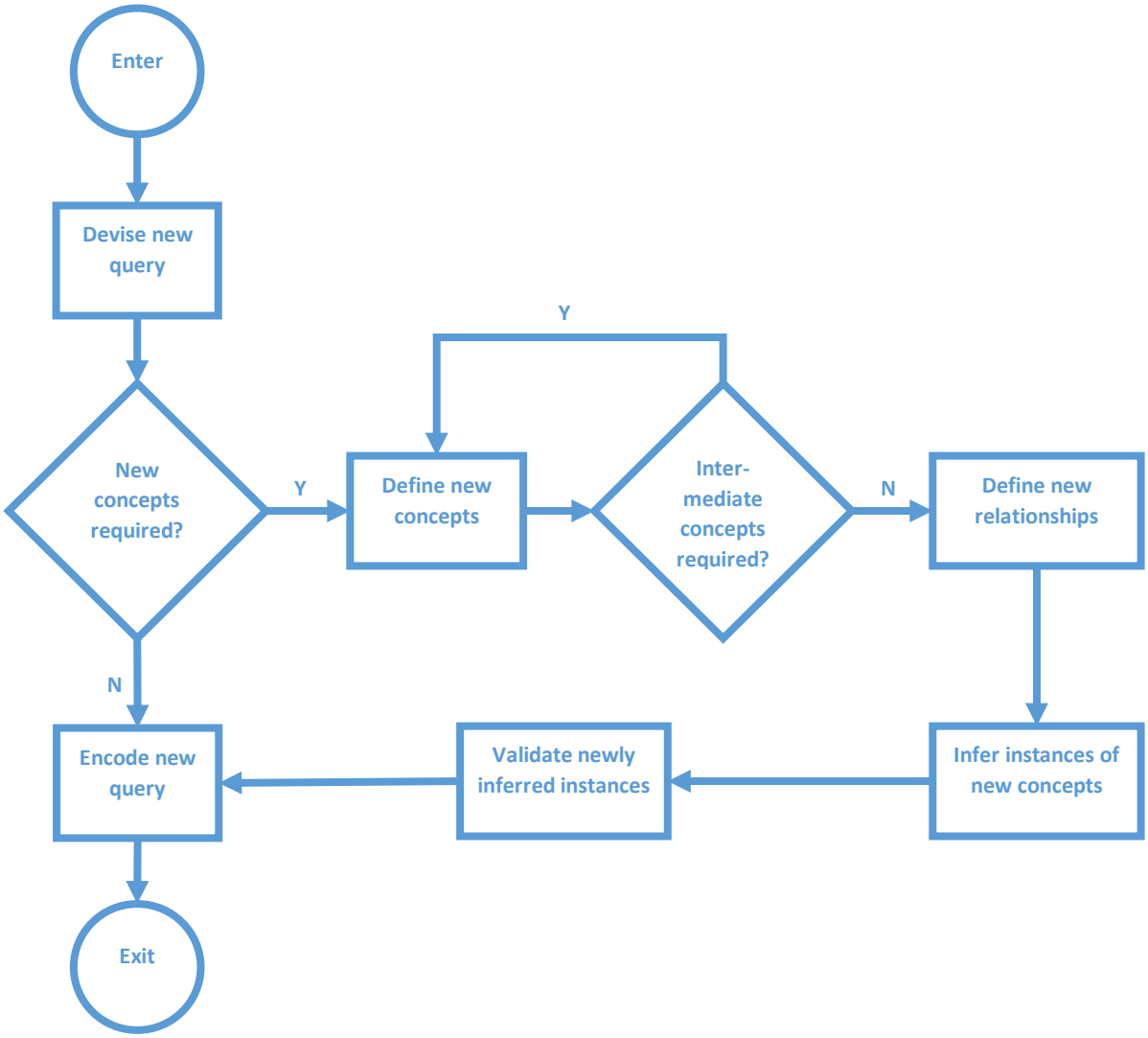


Figure 6.1: Query and Concept Creation Flowchart

6.4.1 Identifying and Adding New Concepts to the Ontologies

Consider Query 1 from Table 6.1. Required concepts are *Host* and *FlowRule*, both defined in the DCN History ontology presented in Chapter 5. Concepts of *Switch*, *FlowTable*, *Packet* and *DCN* are mentioned in the query, and the first two are defined in the DCN History ontology, however none of these four concepts are required to obtain an answer for the following reasons:

- A packet concept is not required because the answer can be found by looking at the statistical properties of flow-rules, which will record whether the flow-rule matched any packets.
- The control-plane message history is from a single DCN, which is not named in the ontology, and does not currently need to be distinguishable from any other DCN. If that situation changed, a DCN concept could easily be introduced.
- The query is concerned with hosts for which unmatched flow-rules were present in the DCN, regardless of the switch or flow-table those flow-rules were associated with.

If further qualification was required, the query could be restated to include the qualification(s) – e.g. 'For what hosts were flow-rules installed in switch flow-tables, but for which no packets were observed in the DCN, and in what switches and flow-tables were those flow-rules installed?'.

Query 2 requires just one new concept – *MigratedHost* – of which *Host* becomes a subclass. With this added concept, some host instances may be inferred (through the appropriate logic) to be migrated hosts.

Query 3 requires a new *TunnelNet* concept. A tunnel network is comprised of a sequence of links between switches, and the flow-rules that match against packets and forward them across those links and on to their ultimate destination.

A new concept may require other new, intermediate concepts to be defined first. That fact may be recognised this stage of the process, or it may not become apparent until the logic is being defined to infer instances of the new concept. For example, to assist with the inference of *TunnelNets* for Query 4, an intermediate concept of *TunnelRule* was defined. A *TunnelRule* is a *FlowRule* that has match criteria or actions that refer to MPLS labels.

For Query 4, *Path* and *PacketSpecification* are concepts that were considered for addition to the DCN history ontology. The *Path* concept has been added to support the query. However, it has as yet not been seen as beneficial to add a *PacketSpecification* concept: a packet is specified simply by listing header field names and values, and for the purposes of the query it is enough to compare that list with packet match criteria in flow-rules.

When defining a new concept, the relationships between that concept and existing concepts should also be expressed and encoded in the ontology to enrich the model, to support future queries and inferences, and to allow correctness checks. For example, the *Path* concept identified for Query 4 is an ordered sequence of *Nodes*, in which the order is represented by a *nextNode* relationship between a pair of nodes, and there is a *hasNode* relationship between a *path* instance and relevant nodes. A *Node* may be a *Host*, *Switch* or *Controller*. In practice, intermediate nodes are switches, the first node may be a *Host* or *Switch*, and the last node may be a *Host*, *Switch,* or *Controller*. The *Node*, *Host*, *Switch*, and *Controller* concepts are already defined on the ontologies. *hasNode* and *nextNode* are new relationships defined in order to answer this query, but can also be used in the future for other queries.

6.4.2 Inferring Instances of New Concepts

Instances of new concepts are inferred via sets of logical statements that are encoded in a programming and/or query language. For each of queries 2, 3 and 4, the logical statements are now presented and explained. Query 1 is not discussed in this section, as it does not require any new concepts.

The logic symbols used in the logic statements are shown in Table 6.2, and are the same as those used in Chapter 5. Variable identifiers are list in Table 6.3. Note that some instances of concepts and data properties represented by variable identifiers are changed from Chapter 5, as there is a requirement for different instances and properties to be represented in this chapter.

Table 6.2: Meanings of relevant logic symbols

Symbol	Meaning	Symbol	Meaning
∃	There exists	¬	Not
∀	For all	≺	Precedes (temporal)
∧	And	→	Implies
∨	Or		

Table 6.3: Logic variables used to represent instances of concepts and data properties

Variable	Represents	Variable	Represents
A	Action	O	Concept instance of unknown type
B	Ordered List	P	Perdurant (Switch, Controller, Host, FlowRule)
C	Controller (and FlowPathController)	Q	Tuple of concept instances of unknown type
D	FlowPath	R	Switchport
F	Match Field	S	Switch (and FlowPathSwitch)
H	Host (and FlowPathHost)	U	Set (of Tunnel-rule, TunnelLink)
J	Graph	V	Value of unknown type (e.g. string, integer…)
L	Link (and TunnelLink)	W	Packet Specification
M	Match	Y	Flow-table
N	Integer numeric value (MPLS label, Switchport number, Priority)	Z	Flow-rule (and Tunnel-rule)

6.4.2.1 Inferring Instances for Query 2: Migrated Hosts

The statement below defines the logic for identifying an instance of a *MigratedHost*:

```
(∀P, ∃H1, ∃H2, ∃L1, ∃L2, ∃R1, ∃R2, ∃S1, ∃S2)(HostPD(P) ∧
      Host(H1) ∧ hasConnectionFrom(L1, H1) ∧ Host(H2) ∧
      hasConnectionFrom(L2, H2) ∧ (L1 ¬= L2) ∧ Switch(S1) ∧
      hasPort(S1, R1) ∧ hasConnectionTo(L1, R1) ∧ Switch(S2) ∧
      hasPort(S2, R2) ∧ hasConnectionTo(L2, R2) ∧ (S1 ¬= S2)
      → MigratedHost(P))
```

Informally: Any host perdurant that had manifestations connected to different switches is a migrated host.

Clearly, the existence of a *MigratedHost* instance depends on the existence of instances of other concepts (*Hosts*, *Links, Switches*), and on the properties of those other instances meeting certain criteria. This illustrates the general approach used to infer instances of new, more sophisticated concepts from existing instances of simpler concepts. In this example, the criteria to be met is relatively straightforward – was a host recorded as having been connected to more than one switch? In other situations, the criteria to be met is more complex, and therefore more challenging to express logically, as demonstrated for the other sample queries below.

There is no explicit reference to timepoints or intervals in the logic statement above. There is an implicit dependence on the lifespan of a *Link* instance (and consequently the *hasConnectionFrom* relationships) being limited to the overlap in the lifespans of the *Host* and *Switch* instances.

In a DCN, a host may be connected via a cable to one port of a switch initially, and later the cable disconnected and re-connected to a different port on the same switch. The host has not migrated, since it has not been connected to a different switch, and the event will not cause the host to be included in the results of a *MigratedHost* query. Nevertheless, a DCN operator could be interested in such events. It would be simple to write the logic to query for instances of these events, following the approach used for *MigratedHost.*

6.4.2.2 Inferring Instances for Query 3: TunnelNets and Related Concepts

Compared to Query 2, inferring *TunnelNets* presents more of a challenge, and requires the approach of inferring intermediate concepts and relationships first, according to this sequence:

1. Search all switch flow-tables for flow-rules (*TunnelRules*) that match packets containing MPLS labels or that add an MPLS label to a matched packet.
2. Find all links (*TunnelLinks*) to which a *TunnelRule* forwards matched packets, and from which another *TunnelRule* accepts matched packets, subject to the condition that the label added or matched by the first rule must be the same as the label matched by the second rule.

The logic for inferring *TunnelRules* is encoded as follows:

```
(∃Z, ∃M, ∃N, ∃A)(FlowRule(Z) ∧ ((hasMatch(Z, M) ∧
      hasMatchField(M, F) ∧ MatchField.fieldName(F,
      "mpls_label") ∧ MatchField.fieldValue(F, N)) ∨
      (hasAction(Z, A) ∧ Action.actionType(A, "push_mpls")))
      → TunnelRule(Z))
```

Informally: If a FlowRule matches packets with MPLS labels, and/or adds an MPLS label to a matched packet, then it is a TunnelRule

For the task of inferring *TunnelLinks*, two logic functions have been defined separately to improve readability – *getPortFromMatch*, to return the input port (if any) specified in a match criteria structure, and *getPortFromAction*, to return the output port (if any) for an action structure. The functions can be thought of as macros, and the logic for each could be substituted for the function calls in the calling rule, which infers *TunnelLink*s.

```
(∃M, ∃N, ∃Z, ∃Y, ∃S, ∃R)(Match(M) ∧ Match.in_port(N) ∧
      hasMatch(Z, M) ∧ hasFlowRule(Y, Z) ∧ hasFlowTable(S, Y)
      ∧ hasPort(S, R) ∧ Port.portNumber(R, N) →
      getPortFromMatch(M, R))
```

Informally: Given a match structure, find the switchport from which packets are accepted by the match

```
(∃A, ∃N, ∃Z, ∃Y, ∃S, ∃R)(Action(A) ∧ Action.port(A ,N) ∧
      (hasAction(Z, A) ∧ hasFlowRule(Y, Z) ∧ hasFlowTable(S,
      Y) ∧ hasPort(S, R) ∧ Port.portNumber(R, N) →
      getPortFromAction(A,R))
```

Informally: Given an action structure, find the output switchport on which matched packets will be forwarded

```
(∃Z1, ∃Z2, ∃M1, ∃M2, ∃F1, ∃F2, ∃N, ∃A, ∃R1, ∃R2,
      ∃L)(TunnelRule(Z1) ∧ ((hasMatch(Z1, M1) ∧
      hasMatchField(M1,F1) ∧ MatchField.fieldName(F1,
      "mpls_label") ∧ MatchField.fieldValue(F1, N)) ∨
      (hasAction(Z1, A) ∧ Action.mpls_label(A, N))) ∧
      getPortFromAction(A, R1) ∧ Link(L) ∧
      hasConnectionFrom(L, R1) ∧ TunnelRule(Z2) ∧ hasMatch(Z2,
      M2) ∧ hasMatchField(M2, F2) ∧ MatchField.fieldName(F2,
      "mpls_label") ∧ MatchField.fieldValue(F2, N) ∧
      getPortFromMatch(M2, R2) ∧ hasConnectionTo(L, R2) ∧
      overlaps(Z1, Z2) ∧ overlaps(Z1, L) ∧ overlaps(Z2, L) →
      TunnelLink(L) ∧ fromTunnelRule(L, Z1) ∧ toTunnelRule(L,
      Z2))
```

Informally: If Z1 is a TunnelRule that forwards match packets on Link K, and Z2 is a TunnelRule that matches packets received from Link K, then (assuming the two TunnelRules use the same label, and their lifespans overlap both with each other and with the link that connects the two switches on which the tunnelrules are installed) K is a TunnelLink.

Once the *TunnelRules* and *TunnelLinks* have been inferred, *TunnelNets* can be inferred by building a graph with *TunnelRules* as vertices and *TunnelLinks* as edges, and then extracting the sets of connected-components from the graph. Each set of connected-components is a *TunnelNet* instance. The inference rule is below. This graph-based approach was previously used in Chapter 5, for example to infer controller instances from switch instances.

```
(∃U1, ∃U2, ∀R, ∃E, ∀L, ∃J, ∃U3, ∀U4)((U1 = {Z :
      TunnelRule(Z)}) ∧ (U2 = {L : TunnelLink(L)}) ∧ (J =
      Graph(U1, U2)) ∧ (U3 = ConnectedComponents(J)) ∧ (U4 ∈
      U3) → TunnelNet(U4))
```

Informally: Given a set N of TunnelRules, and a set E of TunnelLinks, forming a graph G with nodes N and edges E, and finding the set C of connected components of the graph G, then each connected component is a TunnelNet.

6.4.2.3 Inferring Instances for Query 4: FlowPaths and Related Concepts

The logic to infer a *FlowPath* is more complex than the examples seen so far. The path depends on network links in the DCN, on the actions of relevant flow rules in switch flow-tables, and on the priorities of those flow rules. Intermediate concepts and relationships are inferred according to this sequence:

1. Find the highest priority flow-rule (*FlowPathRule*) that will match the specified packet on each switch (*FlowPathSwitch*)
2. Identify hosts (*FlowPathHost*) from which packets will be matched by *FlowPathRules* on the switches to which those hosts are connected, and hosts connected to ports through which *FlowPathRules* will forward those packets
3. Identify where a *FlowPathRule* will forward to a controller (*FlowPathController*)
4. Record the relationships between *FlowPathHosts* and *FlowPathSwitches*:

a. if a packet sent by *FlowPathHost* will be forwarded by a *FlowPathSwitch* (*toFlowPathSwitch*), or

b. if a packet will be forwarded by a *FlowPathSwitch* to a *FlowPathHost*, *FlowPathController*, or another *FlowPathSwitch* *(fromFlowPathSwitch)*

5. Identify ordered pairs of nodes (*FlowPathPair*), where a node is a *FlowPathSwitch*, a *FlowPathHost*, or a *FlowPathController*, and where the first node in the pair will forward a matched packet to the second node.

Logically, instances of the intermediate concepts are identified using the rules below. Firstly, inferring *FlowPathRules* and *FlowPathSwitches*, and the *hasFlowPathRule* relationship between instances of the two:

$$(\exists Z, \exists Q, \forall F1, \exists F2, \exists V1, \exists V2)(\text{FlowRule}(Z) \wedge \text{hasMatch}(Z, M) \wedge \text{hasMatchField}(M, F1) \wedge \text{MatchField.fieldName}(F1, V1) \wedge \text{MatchField.fieldValue}(F1, V2) \wedge \text{PacketSpec}(W) \wedge \text{hasMatchField}(W, F2) \wedge \text{MatchField.fieldName}(F2, V1) \wedge \text{MatchField.fieldValue}(F2, V2)) \rightarrow \text{matchesPacket}(Z, W)$$

Informally: Does a flowrule match a packet specification? Everything in the flow-rule match criteria has to match the PacketSpec, except for ingress port.

$$(\exists W, \exists S, \exists Z1, \exists N1, \neg\exists Z2, \exists N2)(\text{PacketSpec}(W) \wedge \text{Switch}(S) \wedge \text{FlowRule}(Z1) \wedge \text{hasFlowRule}(S, Z1) \wedge \text{matchesPacket}(Z1, W) \wedge \text{FlowRule.priority}(Z1, N1) \wedge \text{FlowRule}(A) \wedge \text{hasFlowRule}(S, Z2) \wedge \text{matchesPacket}(Z2, W) \wedge \text{FlowRule.priority}(Z2, N2) \wedge (N2 > N1) \wedge \text{existsDuringInterval}(W, Z)) \rightarrow (\text{FlowPathRule}(Z) \wedge \text{FlowPathSwitch}(S) \wedge \text{hasFlowPathRule}(S, Z))$$

Informally: for a specified packet, the highest priority flowrule (manifestation) on a switch that can match the packet (if such a flowrule exists at the time specified), then that flowrule is a flowpathrule and the switch manifestation in whose flowtable the rule appears is a flowpathswitch.

Then *FlowPathHosts* are inferred, and the *toFlowPathSwitch* and *fromFlowPathSwitch* relationships recorded, depending on whether each host is a potential source or destination for matched packets:

```
(∃W, ∃H, ∃S, ∃Z, ∃L, ∃R, ∃N1, ¬∃N2, ∃M)(PacketSpec(W) ∧
      Host(H) ∧ existsDuringInterval(W, H) ∧ FlowPathSwitch(S)
      ∧ hasFlowPathRule(S, Z) ∧ hasLink(H, L) ∧ hasLink(S, L)
      ∧ hasConnectionFrom(L, R) ∧ hasPort(S, R) ∧
      Port.portNumber(P, N1) ∧ hasMatch(Z, M) ∧
      (Match.in_port(M, N1) ∨ Match.in_port(M, N2) →
      (FlowPathHost(H) ∧ toFlowPathSwitch (H, S))

(∃W, ∃H, ∃S, ∃Z, ∃L, ∃R, ∃N, ∃A)(PacketSpec(W) ∧ Host(H) ∧
      existsDuringInterval(W, Z) ∧ FlowPathSwitch(S) ∧
      hasFlowPathRule(S, Z) ∧ hasLink(H, L) ∧ hasLink(S, L) ∧
      hasConnectionTo(L, R) ∧ hasPort(S, R) ∧
      Port.portNumber(R, N)∧ hasAction(Z, A) ∧
      Action.actionType(A,"output") ∧ Action.port(A, N)) →
      (FlowPathHost(H) ∧ fromFlowPathSwitch (S, H))
```

Informally: Either a host that is connected to a port on a FlowPathSwitch, where the FlowPathRule on that switch will match packets received on that port; or a host connected to a port on a FlowPathSwitch where the FlowPathRule on that switch will forward matched packets through that port.

Next, any relevant *FlowPathController* is inferred, re-using the *fromFlowPathSwitch* relationship, for *FlowPathRules* that forward to a controller:

```
(∃C, ∃S, ∃Z, ∃A)(Controller(C) ∧ FlowPathSwitch(S) ∧
      hasController(S, C) ∧ hasFlowPathRule(S, Z) ∧
      hasAction(Z, A) ∧ Action.actionType(A, "output") ∧
      Action.port(A, "4294967293")) → (FlowPathController(C)
      ∧ fromFlowPathSwitch(S, C))
```

Informally: a controller to which a FlowPathRule will forward matched packets. (Note: output port for controller specified in flowrules as "4294967293", and matched packets are forwarded to the controller to which the switch is connected).

The fourth step is to find pairs of *FlowPathSwitches* that have a link between them, and where that link will be used to forward a matched packet between them, to register a *fromFlowPathSwitch* relationship between each such pair:

```
(∃S1, ∃S2, ∃Z1, ∃Z2, ∃L, ∃R1, ∃R2, ∃N1, ∃N2, ¬∃N3,
      ∃A)(FlowPathSwitch(S1) ∧ hasFlowPathRule(S1,Z1) ∧
      hasLink(S1, L) ∧ hasConnectionFrom(L, R1) ∧ hasPort(S1,
      R1) ∧ Port.portNumber(R1, N1) ∧ hasAction(Z1, A) ∧
      Action.actionType(A, "output") ∧ Action.port(A, N1) ∧
      (FlowPathSwitch(S2) ∧ hasFlowPathRule(S2,Z2) ∧
      hasLink(S2, L) ∧ hasConnectionTo(L, R2) ∧ hasPort(S2,
      R2) ∧ Port.portNumber(R2, N2) ∧ hasMatch(Z2, M) ∧
      (Match.in_port(M, N2) ∨ Match.in_port(M, N3)) →
      fromFlowPathSwitch (S1, S2)
```

Informally: if one FlowPathSwitch has a FlowPathRule that forwards packets through a link from which another FlowPathSwitch has a FlowPathRule that will match and accept packets, then there is a fromFlowPathSwitch relationship between the two switches.

Lastly, *FlowPathPairs* are inferred – tuples of the form (*FlowPathHost*, *FlowPathSwitch*), (*FlowPathSwitch*, *FlowPathSwitch*), (*FlowPathSwitch*, *FlowPathController*) and (*FlowPathSwitch*, *FlowPathHost*) where there is a *fromFlowPathSwitch* or *toFlowPathSwitch* relationship between the elements of each tuple:

$$(\exists S1, \exists S2)(\text{FlowPathSwitch}(S1) \wedge \text{FlowPathSwitch}(S2) \wedge \text{fromFlowPathSwitch}\ (S1, S2)) \rightarrow \text{FlowPathPair}(S1, S2)$$

$$(\exists H, \exists S)(\text{FlowPathHost}(H) \wedge \text{FlowPathSwitch}(S) \wedge \text{toFlowPathSwitch}(H, S)) \rightarrow \text{FlowPathPair}(H, S)$$

$$(\exists S, \exists C)(\text{FlowPathSwitch}(S) \wedge \text{FlowPathController}(C) \wedge \text{fromFlowPathSwitch}\ (S, C)) \rightarrow \text{FlowPathPair}(S, C)$$

$$(\exists S, \exists H)(\text{FlowPathSwitch}(S) \wedge \text{FlowPathHost}(H) \wedge \text{fromFlowPathSwitch}\ (S, H)) \rightarrow \text{FlowPathPair}(S, H)$$

Informally: a FlowPathPair is one of: a FlowPathSwitch that forwards to another FlowPathSwitch; a FlowPathHost that is connected to a FlowPathSwitch; a FlowPathSwitch that forwards to a FlowPathHost; or a FlowPathSwitch that forwards to a FlowPathController.

Having identified all possible path elements, inferring a *FlowPath*, given a starting node X, is a matter of finding the *FlowPathPair* (X, Y) (i.e. the first element is the starting node), if such a pair exists, and then iteratively finding the next pair (Y, Z) (i.e. the first element of the next pair is the second element of the previous pair), until there is no further pair found that meets the criteria. The set of pairs found forms the path that the matched packet would have followed through the DCN from the starting node. This definition is more challenging to encode logically due to the inherent iteration:

$$(\exists B, \forall Q_i, \exists M_i, \exists N_i)((B = [Q_1, Q_2, \ldots Q_i\ldots]) \wedge (Q_i = (O1_i, O2_i)) \wedge \text{FlowPathPair}(O1_i, O2_i) \wedge (((Q_{i+1} = (O1_{i+1}, O2_{i+1})) \wedge (O1_{i+1} = O2_i)) \vee \neg\exists Q_{i+1})) \wedge ((O2_i \neg= O1_j) \wedge (i > j)) \wedge \text{StartNode}(O1_1)) \rightarrow (\exists D)(\text{FlowPath}(D) \wedge \text{hasNode}(D, O1_i) \wedge \text{hasNode}(D, O2_i) \wedge (\text{nextNode}(O2_i, O1_{i+1}) \vee \neg\exists O1_{i+1})))$$

Informally: a FlowPath is a sequence of one or more FlowPathPairs, where the first element of the first pair is the specific starting node, and the first element of each successive pair is also the second element of the previous pair.

The inferred *FlowPath* is retrieved for examination by the DCN operator, who will be interested in whether the actual destination is the intended destination, for example, and whether the path taken to the destination is the expected or best path.

6.4.3 Validating Instances of New Concepts

Section 6.4.2 presented examples of logic statements for inferring instances of new concepts. Inferred instances of ontological concepts can be validated using SHACL [130] shape definitions (with SPARQL for advanced checks), or by using alternative tools such as SPIN [127] or ShEx [145] to check for inconsistencies or illogical results. Listing 6.1 provides examples of SHACL shape definitions for validating the shape of a Path instance (which should have at least one node, a node marked as the first node on the path, and a node marked as the last node on the path) and the shape of a PathSwitch (which should be related to a switch and a specific flowrule, and should only appear once on the path of which it is a component). The shape definitions should be created at the same time as the ontological definitions of the new concepts, and later used to validate inferred instances.

6.4.4 Encoding Queries

As stated earlier in section 6.4, when all necessary concepts are defined, and instances of those concepts and their relationships to both existing and new concepts have been inferred, the query may be encoded to extract the relevant information from the populated ontology. Encoding a query is an implementation task, described in section 6.5 below.

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix dcnhistory:
<http://www.purl.org/osearbhain/ontologies/2019/5/dcn-history/> .

dcnhistory:Shapes {

 dcnhistory:PathShape
  a sh:NodeShape ;
  sh:targetClass dcnhistory:Path ;
  sh:property [
    sh:path dcnhistory:hasNode ;
    sh:minCount 1 ;
  ] ;
  sh:property [
    sh:path dcnhistory:firstNode ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:path dcnhistory:lastNode ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .

 dcnhistory:PathSwitchShape
  a sh:NodeShape ;
  sh:targetClass dcnhistory:PathSwitch ;
  sh:property [
    sh:path dcnhistory:hasSwitch ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:path dcnhistory:hasFlowRule ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:sparql [
    sh:message "{$this} part of loop on path" ;
    sh:select """
      prefix dcnhistory:
<http://www.purl.org/osearbhain/ontologies/2019/5/dcn-history/>
      select $this
      where {
        $this dcnhistory:nextNode/dcnhistory:nextNode* $this ;
      } """ ;
    #sh:maxCount 0 ;
  ] .

}
```

A *Path* has at least one *node*, and has exactly one *firstNode* and exactly one *lastNode*.

A *PathSwitch* relates to exactly one *Switch* and exactly one *FlowRule*, and may only appear once on a path.

Listing 6.1: SHACL shape definition example

6.5 Implementation Approach

The implementation work to answer the queries described above follows the workflow presented in Chapter 5 (section 5.6) – although Step 4 may be better described now as the inference of DCN concept instances from OpenFlow message instances and *previously inferred instances*, rather than from message instances alone. Step 5 is still accurately described as query execution.

The same languages and tools as listed for Steps 4 and 5 in Chapter 5 were used to implement queries 1 to 4 – Lisp, Python, Prolog and SPARQL for inferencing and querying, with AllegroGraph as the triple-store. The 'graph' library provided by the Quicklisp library manager was again used, as the 'connected-components' function was employed to identify *TunnelNet* instances for Query 3.

For Query 4, an instance of the *PacketSpecification* concept could have a comprehensive set of properties consisting of all possible header field names so that a value can be specified for each header field for a particular packet. However, it is sufficient to only specify values for header fields that will be matched by flow-rules. A SPARQL query was constructed to extract from the ontological mode a full list of header fields that are matched against by installed flow-rules, and it is enough to provide values for those fields in a query for a *FlowPath*. For example, in addition to default rules, the controller may possibly only install rules that match on source and destination MAC address, and EtherType value. A *FlowPath* query with values for each of those fields will suffice to find the resulting path. In light of this approach, the PacketSpecification concept (as mentioned in section 6.4.1) has not been formally defined in the ontologies.

6.6 Methods for Evaluation

There are two aspects to be evaluated – the correctness of the inferences that have been made, and the accuracy of the results of queries. Several methods have been used in this work:

1. **Inferencing**: Making inferences and running queries against a log of control-plane messages from a network where events have been triggered and

monitored in a controlled manner, with the timing of all relevant events recorded separately to the message log. The information gathered is invaluable when verifying results generated by the implementation of newly developed inferencing logic and queries.

2. **Correlation**: Recording the connections between inferred object instances and key pieces of evidence for their existence as relationships in the triple-store. If done consistently, this allows the inferencing steps to be traced backward through multiple layers of inferred instances of objects to the specific control plane messages that form the foundation on which the results are based. This is useful for debugging when developing new queries and logic, but also to provide the confirmatory evidence when a query returns unexpected results.

3. **Validation**: Applying validation tests to inferred instances, using the SHACL shape definitions (or alternatives) defined earlier. This method is useful for the same purposes as the methods above, but additionally, if validation tests are included as part of the workflow each time new object instances are inferred, they can reveal previously undiscovered bugs in the logic or implementation.

For the sample queries presented in this chapter, except for Query 1, all three methods have been used. Query 1 is the exception because no new object instances needed to be inferred, so no new instances required validation.

No results are shown for Query 1, because the DCN topologies from which control-plane messages were captured used reactive rather than proactive installation of forwarding rules (other than for tunnelling). The query was run on ontologies populated from those logs, but the set of results was empty – i.e. no rules were configured that did not match packets for their respective source host.

6.6.1 Example of Results for Query 2

For Query 2 (identifying migrated hosts), an experiment was performed on an emulated DCN in the testbed described in Chapter 3. In this experiment, a host was

migrated, and the control plane messages were captured from the network to record the event. In fact, the event was simulated on a network created using MaxiNet by detaching a host from the switch to which it was attached and changing the identity (IP address and MAC address) of another host to match that of the first one (This approach was used as MaxiNet and Mininet do not provide functions to support host mobility). The message log was used to populate the message ontology in the triple store, against which the inferencing code was then run – inferring basic DCN elements first, and then inferring migrated hosts. The results were compared with the known parameters of the original experiment and found to match.

A chart of sample results for Query 2 is shown in Figure 6.2. Labels on the Y-axis identify the IP address of a host, and the DPID of the switch to which the host was connected. The duration of the connection is represented in the chart. For each migrated host, the chart shows that the host was initially connected to a specific switch and was migrated to a different switch at a point in the experiment. The results match the observed events on the original DCN from which the message log was captured, with some exceptions:

- Some hosts were migrated but do not appear on the chart, because they did not generate traffic before and after their migration.
- The host with IP address 10.0.2.5 is anomalous, because it appears to migrate to a second switch, and then migrate back to the first switch. This happened because of the method by which the migration was performed in MaxiNet, as described above. In the case of the host with IP address 10.0.2.5, the operation to detach the original host failed, and it resumed actively sending packets afterwards, thereby appearing to have migrated back to its original location.

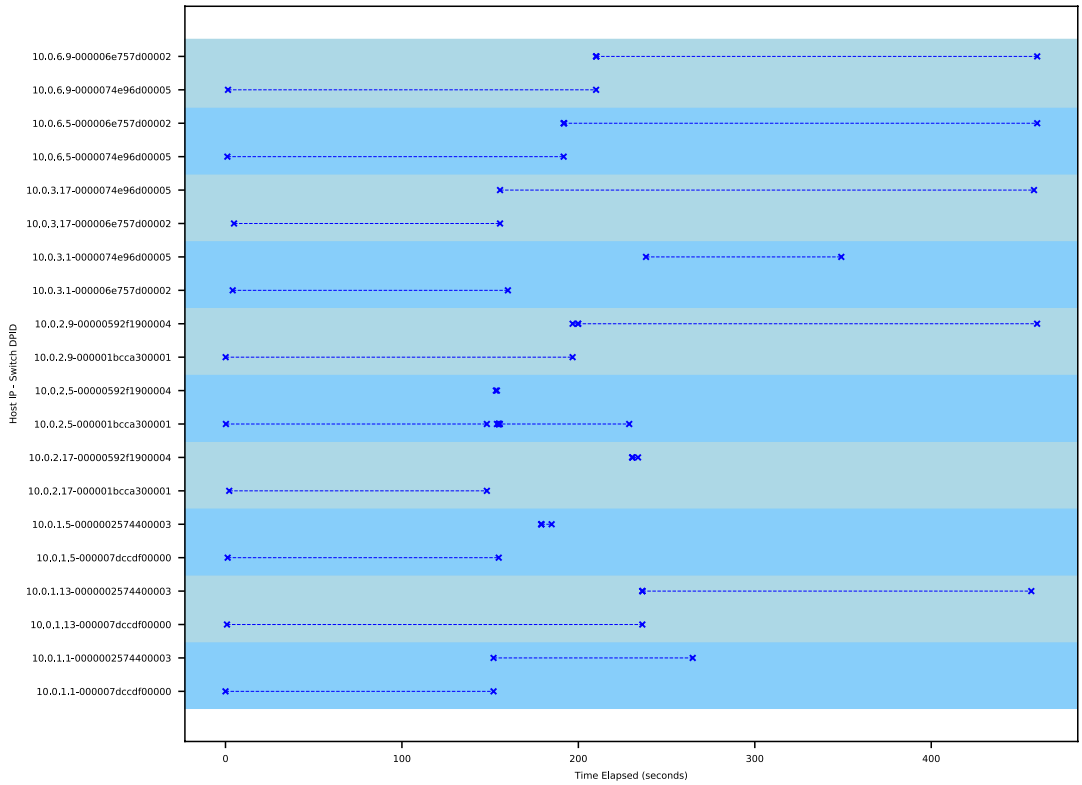


Figure 6.2: Migrated Hosts

6.6.2 Example of Results for Query 3

For Query 3 (inferring VPLS tunnels), triples are written to the datastore relating each *TunnelNet* to the *FlowRules* that form the tunnel. Each *FlowRule*, in turn, has triples relating it to control-plane messages that formed the evidence for inferring the *FlowRule*.

Listing 6.2 shows partial output obtained by running Query 3 on an ontology populated from a log of messages captured from a DCN where VPLS tunnels were configured. The details of one VPLS tunnel network are shown: *tunnelNet5* en-tunnels packets (by adding an MPLS label) received on ports 10 and 11 of *switch6*, forwards tunnelled packets to *switch12*, from where they are relayed to *switch10*, which de-tunnels them (removes the MPLS label) before outputting them on port 22. The tunnel was present for approximately 47 seconds.

The output from Query 3 is of use for validating that tunnels were configured, and that the durations for which they were configured correspond with tunnel-service establishment and teardown requests from tenants.

```
tunnelnet   startpoint    endpoint      tunnelrule    switch    inPort  outPort

tunnelNet5  11:17:10.228 11:17:57.533 tunnelRule21 switch6     10
tunnelNet5  11:17:10.228 11:17:57.533 tunnelRule22 switch6     11
tunnelNet5  11:17:10.228 11:17:57.533 tunnelRule1  switch12
tunnelNet5  11:17:10.228 11:17:57.533 tunnelRule13 switch10           22
```

Listing 6.2: Example of partial output for Query 3

6.6.3 Example of Results for Query 4

The purpose of query 4 is to return the inferred path along which a specified packet would have been forwarded if injected into the DCN at a particular point in time. The packet was specified in terms of header field names and values shown in Table 6.4. The *eth_src* value is the MAC address of *host45*, whose IP address is 10.0.3.11. The *eth_dst* value is the MAC address of *host45*, whose IP address is 10.0.4.11. The *eth_type* value of 2048 indicates that the Ethernet frame encapsulates an IP datagram. *host45* was specified as the injection point from which the inferred path should start.

Table 6.4: Header Field Values for Specified Packet

Header Field Name	Header Field Value
eth_src	12:3a:10:fa:de:f7
eth_dst	76:1d:94:ef:d5:63
eth_type	2048

The output of query 4 is shown in several formats. Listing 6.3 (a) shows the inferred *FlowPath* as an ordered n-tuple; Listing 6.3 (b) shows the *FlowPathPairs*, not in order. Figure 6.3 charts the inferred path on a composite inferred topology diagram, as presented in Chapter 5.

The inferred elements were validated using SHACL shape definitions, as described in section 6.4.3. The pySHACL tool [146] was used to perform the validation, resulting in a validation report of ‘Conforms: True’, indicating that the inferred elements passed the tests.

```
(a) (pathHost1.host43 switch10.flowrule80 switch11.flowrule141
     switch6.flowrule129 pathHost2.host45)

(b) ('switch10', 'switch11')
    ('switch6', 'host45')
    ('switch11', 'switch6')
    ('host43', 'switch10')
```

Listing 6.3: Output Inferred Packet Path for Specified Packet

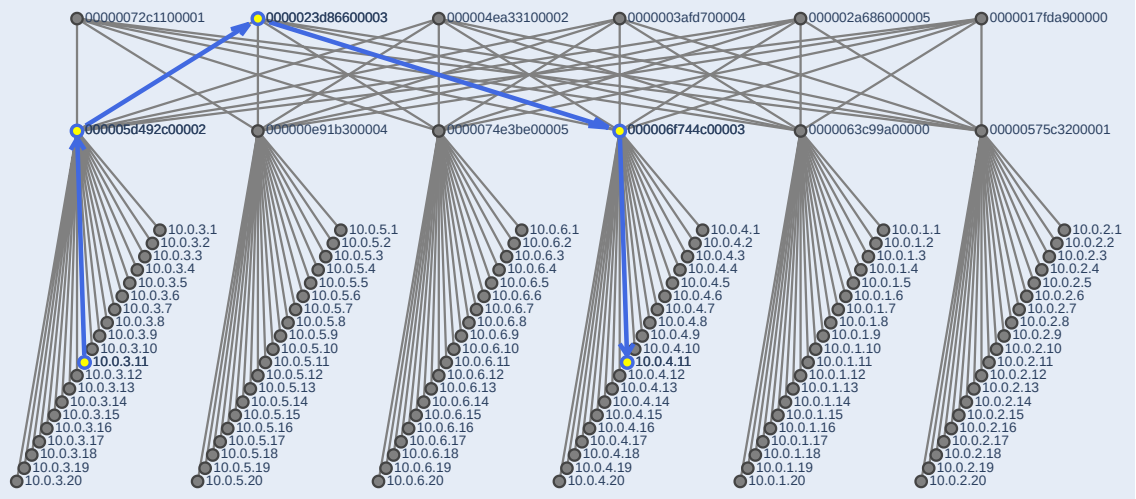


Figure 6.3: Inferred Path Charted for Specified Packet

6.6.4 Discussion of Results

The results of each of the queries provides deeper insight into the inner-workings of the DCN, making it less of a black box for the operator. More importantly, the results enables the operator to state concrete, objective facts about the DCN's past behaviour:

- Whether switch flow-tables contained rules for inactive hosts;
- Which hosts were migrated, when, and where;
- What tunnels were configured and, again, when and where;
- Exactly what sequence of switches would a certain packet cross based on the flow-rules that were configured at a chosen time, and at what destination would that packet concluded its journey.

The individual results obtained here are reassuring evidence that the DCNs whose histories were the target of the queries were operating and behaving as expected, the exception being the migrated host that didn't successfully migrate (and, as pointed out, that was an issue with the simulation environment). That reassurance may itself be useful to a DCN operator.

6.7 Conclusion

In this chapter, a sample set of DCN operator queries were presented. The queries target ontologies populated from control-plane message logs, as described in Chapter 5. The message logs are from experiments performed on simulated DCNs under the control of an OpenFlow SDN controller but could be from any DCN for which a message ontology has been prepared. The queries are of varying levels of complexity, and have been used to illustrate a methodology developed for designing and developing ontological concepts for the purposes of answering queries. Such a methodology has not been presented before in the context of software-defined DCNs.

The populated ontologies, like the underlying message logs from which they were populated, are a rich source of data. A key contribution of this chapter has been demonstrating that the ontologies provide a solid, logical foundation on which to build concepts, both simple and complex, to make the data accessible to inquisitive DCN operators. Some queries have been implemented, to demonstrate the methodology. In the context of a network management model such as FCAPS[147], the queries presented clearly can be used to validate configurations, as part of Configuration Management. However, they can also yield information useful for completing tasks under other categories. For example, the results of queries analysing flow-table contents (such as Query 1) could inform the process of streamlining policy rules, to improve performance under the category of Performance Management. Investigation of paths (with the results of Query 4) may identify a path that terminated somewhere it should not have – critical information for Security Management. Many more insightful queries, related to all FCAPS categories, can be devised and implemented by following the methodology.

In practice, the methodology requires programmatic and ontological skills, as well as domain knowledge about DCNs and SDN. A DCN operator will have some level of domain knowledge, but most likely will require assistance from experts with the other skills. Options for reducing the dependence on programmers and ontologists are discussed as future work, in Chapter 8.

7 Reducing the Latency of SDN Controller and Hardware Switch Interactions

7.1 Introduction

Previous chapters presented methods to provide better insight for Data Centre Network (DCN) operators into the past history of their networks. During the research work, an opportunity was identified to improve the accuracy of the historical message logs upon which those methods are based. Work published by other researchers describes shortcomings in hardware Software-Defined Networking (SDN) switches affecting their control-plane performance. Knowledge of common DCN network architectures suggested a way to mitigate the performance issue, while also improving the veracity of OpenFlow control-plane messages and therefore that of the recorded network state.

7.2 Motivation

Most DCN switch manufacturers ship devices that support the OpenFlow protocol. However, their implementations differ, and suffer from some problems. Researchers have shown [82, 83, 148] that OpenFlow rule modification times vary from device to device, and the delays can be significant, particularly if rule modification requests are received at a high rate, if the rules have different priorities, and if flow tables have high occupancy levels.

Furthermore, as those researchers discovered, some hardware switches prematurely indicate a rule has been installed in a flow table, and the rule may not affect the forwarding of packets in the data plane for some length of time. Such delays and uncertainty are unacceptable in a multi-tenant DCN for two reasons:

a) distributed application performance is degraded, and

b) security is compromised if the destination of packets cannot be fully controlled.

In contrast to the situation with hardware switches, software switches (for example, Open vSwitch [149], a widely used open-source software switch) can

handle much higher flow-modification request rates, without the uncertainty as to whether a rule has taken effect. As part of this work, experiments were performed to verify the assertion that a software switch can outperform a hardware switch in handling flow-modification requests – the experiments and results are discussed in section 7.6.1.

At first glance, it appears counter-intuitive that a software switch could be faster than a hardware switch, however the assertion applies specifically to control-plane operations. Clearly, hardware switches can forward packets at the data plane at a much higher rate than software switches. The primary reason for the difference in control-plane performance is where the flow-tables are stored. On a software switch, the flow-tables are in RAM, and while updating the data structure is fast, looking up the data structure to find a matching rule for packet forwarding is relatively slow. On a hardware switch, the flow-tables are stored in Ternary Content Addressable Memory (TCAM) where possible, which provides fast lookup, but is slow to update. This is a simplified explanation, and not all switches use TCAM in exactly the same manner or have the same amount of TCAM. Some higher-end switches are not affected, and those switches that are affected can have different performance profiles.

For proactively installed flow rules, the hardware switch's limitation is not necessarily an issue – so long as the rules are fully installed ahead of the arrival of any packets that will match against those rules. While that may be reasonable to enforce in an enterprise DCN, in a multi-tenant DCN it could cause unacceptable delays to service provisioning for new tenants or existing tenants who change their resource requirements. Furthermore, in any DCN there will likely be some flows for which flow rules need to be reactively installed, even if most rules are installed proactively.

7.3 Research Challenges

The key research question to be answered is:

- How can the control-plane delay for flow-table updates of hardware OpenFlow switches be reduced or otherwise compensated for in a DCN?

Clearly it does not make sense to replace all hardware switches in the DCN with software switches. On the other hand, it is common to have both hardware and software switches in a network. In a DCN in particular, while physical servers are interconnected via hardware switches, the hypervisor or server operating system normally incorporates a software switch. Virtual machines running on the server communicate with each other through the software switch. The path for those VMs to communicate with non-local entities (e.g. VMs running on other servers in the data centre) goes through the software switch. It has been observed [113, 150] that much of the traffic in a DCN is east-west, i.e. between servers. For east-west traffic, the first and last switches to handle packets of a particular flow are likely to be software switches in the hypervisors of host servers, but could also be software switch processes running in a non-hypervisor operating system.

7.4 Design

The insight into the characteristics of switches and the structure and operation of a DCN inspired an idea that software switches could be used to mask the control-plane limitations of hardware switches. The idea was formulated to temporarily tunnel new flows that must be handled reactively by adding flow rules to software switches to direct packets through pre-configured tunnels on the hardware switches. The approach is illustrated in Figure 7.1, and outlined as follows:

1) Pre-configured tunnels connect software switches across the hardware switches in the DCN. A mesh of tunnels between all servers is established – for clarity, however, the figure shows just two servers with a single tunnel between their software switches.

2) When a new flow arrives (or a current flow needs to be modified), high priority rules are installed on software switches to forward the flow across

the appropriate tunnel. Then normal priority rules are installed on software and hardware switches to be ready to carry the flow on a non-tunnelled path. Flow rules are matched in order of priority, so while the high priority rules are present, all matching packets will be tunnelled.

3) When the non-tunnelled path has been verified, the high priority rules are removed so that subsequent packets of the flow follow the non-tunnelled path.

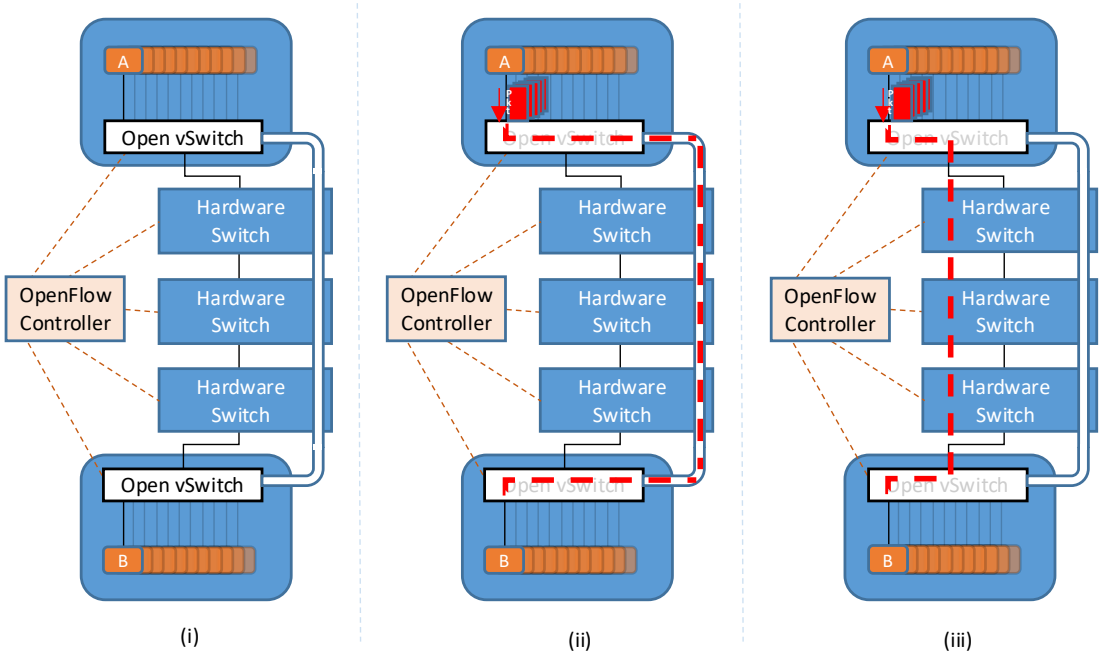


Figure 7.1: Temporary tunnelling of new flow

(i) tunnels are established between pairs of software switches before traffic flow
(i) a new flow is initially forwarded through tunnels, while flow-rules are updated on hardware switches
(i) tunnelled flow migrated to non-tunnelled path, once the latter has been verified to be configured

While flows could be left to complete on the tunnelled connections (essentially a virtual network solution), there are reasons not to:

a) It would result in an imbalance in traffic across the DCN, the architecture of which normally includes multiple paths between servers;

b) The ability to have granular treatment of flows at intermediate switches (for example for QoS or security purposes) is impacted;

c) Tunnelling imposes a per-packet overhead of marking packets by, for example, adding fields, thereby consuming bandwidth and processing power, and adding latency.

Two additional scenarios for which temporary tunnelling provides compelling advantages:

1) Quickly resolving congestion by reliably and predictably migrating flows to alternative paths.

2) Handling proactive flow-rule installation for flows for which packets may arrive very soon (as in a multi-tenant DCN).

This approach works for several tunnelled paths between any pair of software switches for redundancy in case of switch or link failure, and for traffic load distribution. The number of tunnels may increase or decrease dynamically according to requirements but should be kept small to limit the quantity of extra flow-rules required in hardware switch tables for tunnels, especially in a large DCN network. However, all paths through the network are available for non-tunnelled flows.

Flows that require granular treatment at intermediate switches need not be tunnelled at all, or may be prioritized for migration to non-tunnelled paths over those flows that do not require such treatment.

The approach incurs the overhead of tunnelling to some extent. For flows of short duration, all packets may end up being tunnelled. For long flows consisting of packets with large payloads (e.g. jumbo frames), the overhead of tunnelling is not overly significant.

7.5 Implementation

The controller implementation was written in Python, and based on Pox. A TCP connection comprises two uni-directional flows, requiring flow rules to be configured for both. As illustrated in Figure 7.2, a *PacketIn* message from a Pox *packet_in* event for a new TCP connection is the trigger for the controller to

configure the tunnel rules on the software switches for the two flows. Flow-rule addition requests are sent for the inbound flow first, and then the outbound flow, to reduce the likelihood of *packet_in* events for the same flow being generated by switches other than the first one on the path.

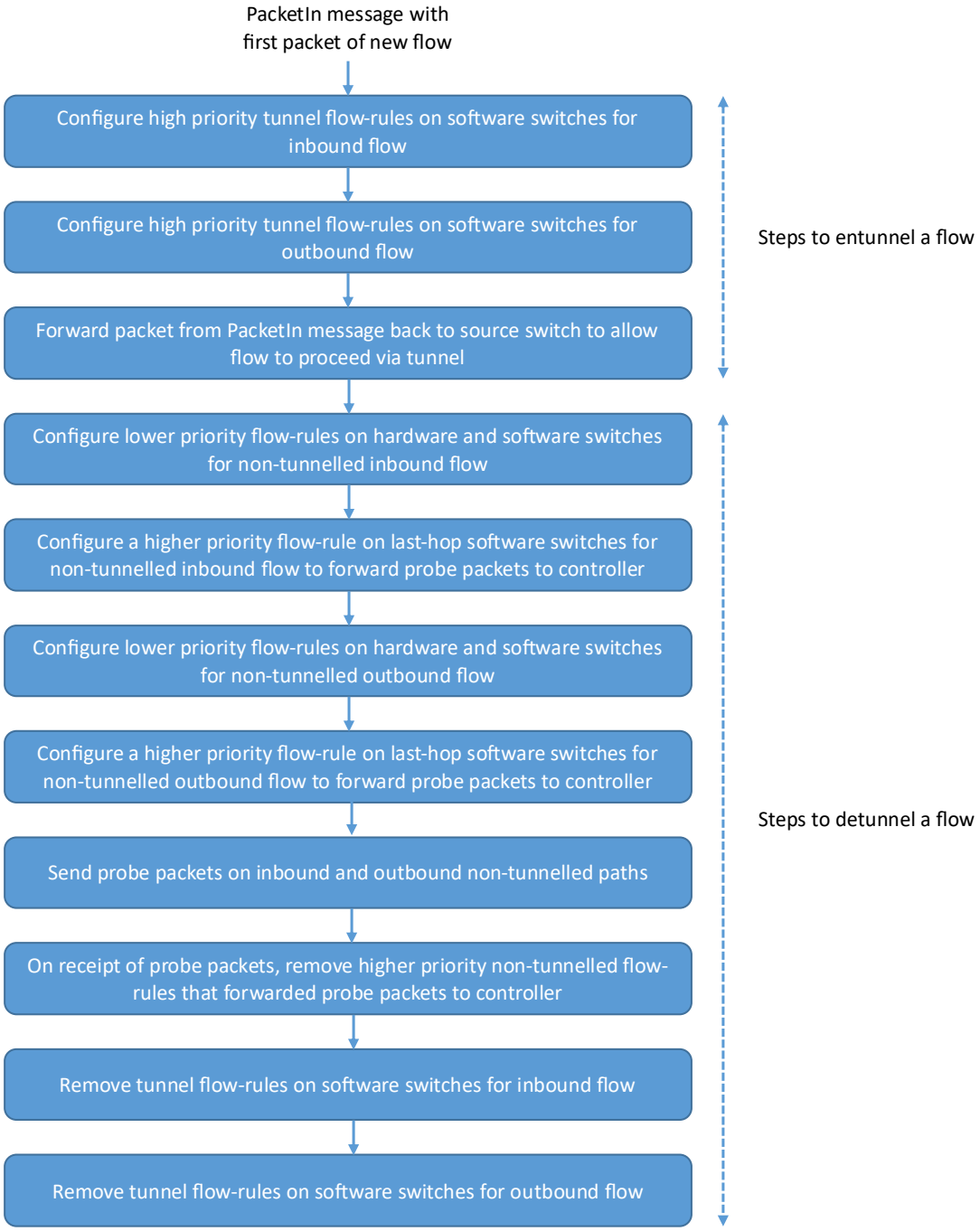


Figure 7.2: Steps taken by a controller to handle a new flow

The controller keeps track of all active flows, using the information to ignore duplicate *packet_in* events, to migrate flows to non-tunnelled paths as necessary, and to verify a non-tunnelled path before migrating a flow to that path.

Once the rules are configured for the tunnelled paths, the next step is to configure rules on the software and hardware switches for the non-tunnelled path, assuming the flows have not yet completed. The implementation uses one path through the DCN for tunnelled traffic, and a different path, if available, for non-tunnelled traffic.

The controller verifies the non-tunnelled path with probe packets. When a flow-rule is added to the last switch on a non-tunnelled path, a higher priority rule with the same match criteria is added to redirect matched packets to the controller. The controller sends probe packets through the first switch on the path, and when it successfully receives a probe packet back from the last switch, the higher priority rule is removed so that subsequent packets of the flow will follow the verified non-tunnelled path.

MPLS labels are used for tunnelling. MPLS was chosen because the overhead it introduces is low compared to other tunnelling methods, it is supported by all versions of OpenFlow, and requires no special NIC or switch support. When the switches initially connect to the controller, the hardware switches are configured with flow rules to forward packets between ports through which software switches are reachable, based on MPLS label values. When the controller learns of a new flow, it configures rules on the software switches to label packets of that flow before forwarding to a hardware switch, and remove labels from packets of the flow received from a hardware switch.

The design mandated a mesh of tunnels between servers. However, a set of tunnels that terminate at the same point can share the same MPLS label. The two DCN topologies illustrated in Chapter 5, a spine-leaf topology (Figure 5.12) and a multi-tier topology (Figure 5.9), the number of tunnel rules is evaluated below.

For a spine-leaf DCN with n total servers, of which k servers connect to each leaf switch, a leaf switch requires n rules: $n - k$ rules to forward packets towards all

remote servers, and *k* rules to deliver packets from all remote servers to the directly connected servers. Packets are forwarded by leaf switches such that for each spine switch the labelled packets that it receives all have as their destination a server connected to one leaf switch, requiring *k* rules on the spine switch for forwarding.

In the multi-tier DCN, for *n* total servers (evenly distributed across the server racks), of which *k* servers connect to each top-of-rack (ToR) switch, and where there are *p* edge switches per cluster and *q* aggregation switches: each ToR switch requires *n* rules (*n* – *k* rules to forward packets towards all remote servers, and *k* rules to deliver packets from all remote servers to the directly connected servers); at the edge layer, the forwarding task is split between the edge switches in the cluster, therefore each edge switch requires *n*/*p* rules; and at the aggregation layer the forwarding task is similarly split, resulting in each aggregation switch requiring *n*/*q* rules.

More rules may be desired to provide redundancy, or more balanced use of available paths. Optimisations are possible to reduce the number of rules required at higher layers of the DCN topology through wildcard matching of MPLS labels.

7.6 Evaluation

The first step in terms of evaluation was to provide the evidence, in terms of experimental results, to prove that a software switch could be faster than a hardware switch at the control-plane. The next step was to illustrate the viability and benefit of the temporary tunnelling approach, again supported by the results of experiments. Both steps are described below.

7.6.1 Problem Verification

Experiments were performed to verify the assertion that a software switch can outperform a hardware switch when modifying its flow tables in response to control-plane flow-modification requests. As mentioned in section 7.2, the short-comings of hardware switches were investigated and presented in [82, 83]. The researchers who produced those papers modelled the control-plane behaviour of

some of the switches they investigated and developed code to emulate that behaviour. In using that emulator for this work, the first step was to ensure that the results obtained by the original research team could be reproduced. This was achieved successfully once the environment was configured correctly. An important piece of that configuration was to ensure that Nagle's algorithm was turned off for the TCP socket through which the SDN controller communicated with the switch. Nagle's algorithm is an optimisation to TCP that reduces the number of packets sent by waiting for previously transmitted data to be acknowledged before sending more. Until it was turned off, this behaviour added latency of up to 40ms to messages sent from the controller to the switch.

The second step was to run experiments to allow comparison between an emulated hardware switch and a software switch. These experiments were performed in Mininet, with Open vSwitch, and using Pox as the OpenFlow controller, as illustrated in Figure 7.3.

A Mininet script created the network, with a software switch (an Open vSwitch instance) and two virtual hosts (*H1*, *H2*) connected to ports (*sw-eth1* and *sw-eth2*) on the switch. The hardware switch emulator operated as a proxy between the POX controller and Open vSwitch. The emulator was present for experiments that reproduced the original researchers' results and bypassed for experiments to discover the comparative behaviour of a software switch for the same tests.

Following the methodology used by the researchers in [82], an experiment followed this sequence:

1. Prepopulate the switch flow-table with *R* default-priority, non-overlapping rules each forwarding matching UDP datagrams from port *sw-eth1* to *sw-eth2*. The only difference between the rules was the destination UDP port number, which had values in the range of 1 to *R*.

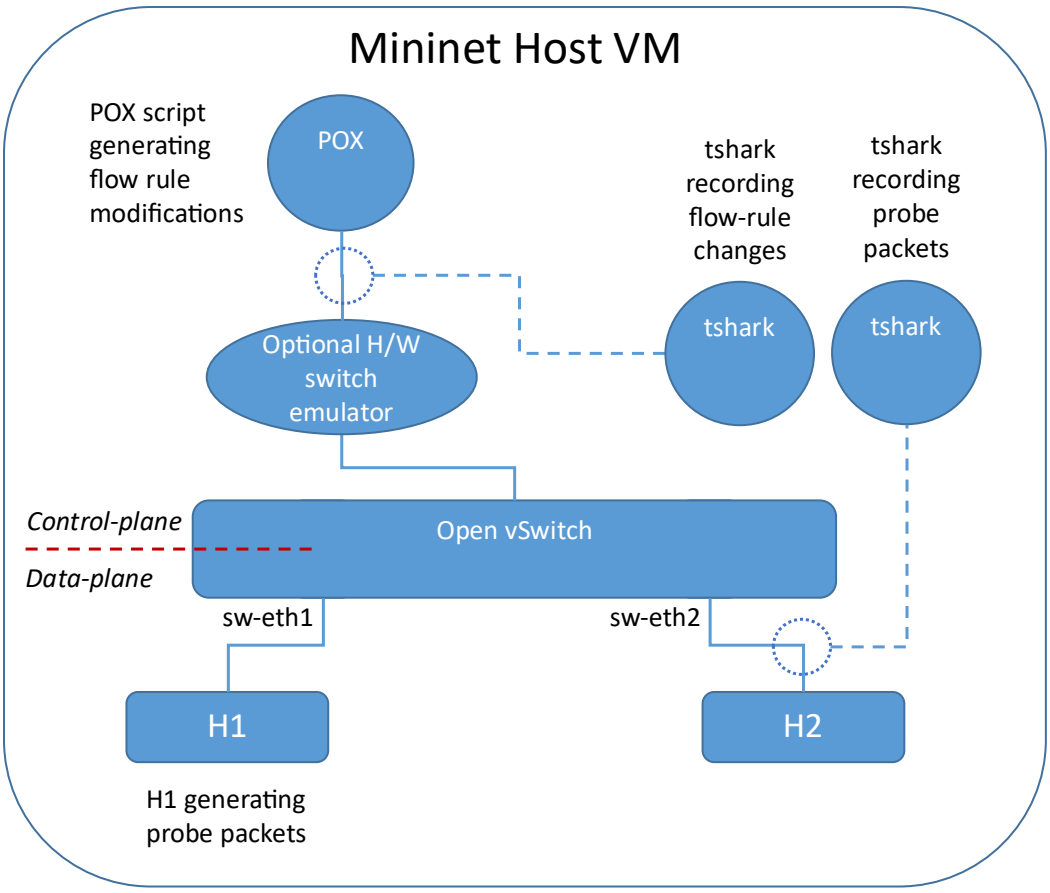


Figure 7.3: Virtual testbed for timing flow-rule updates

2. Send *B* batches of OpenFlow request messages to the switch. A batch *i* consisted of:

    a. a controller-to-switch request to delete an existing flow rule (rule number $R - i$),

    b. a request to add a new flow rule (rule number *i*), and

    c. a barrier request to prompt the switch to confirm when it had previous requests completed.

The purpose of prepopulation was to avoid the trivial case of an empty or nearly empty switch flow-table. Transmission of a batch started when a barrier reply was received for the second previous batch, the aim being to keep the switch busy with two 'batches in-flight' while avoiding queue overflow due to too many flow-rule

modification requests. Note that removing a flow-rule every time a new one is added maintains the total number of flow-rules in the table at a constant level, so there was no risk of a flow-table overflowing.

After prepopulation but before batches were sent, probe packets were generated at a rate of 1000 packets per second to match all new flow rules being added by the batches. Probe packets were sent from H1 to H2, and would only successfully reach their destination once a matching rule was configured. During the experiment, tshark instances captured barrier-reply messages arriving at the controller from the control-plane and probe packets arriving at H2 on the data-plane.

The results for experiments with R = 300 and B = 50 are shown in Figure 7.4 and summarised numerically in Table 7.1.

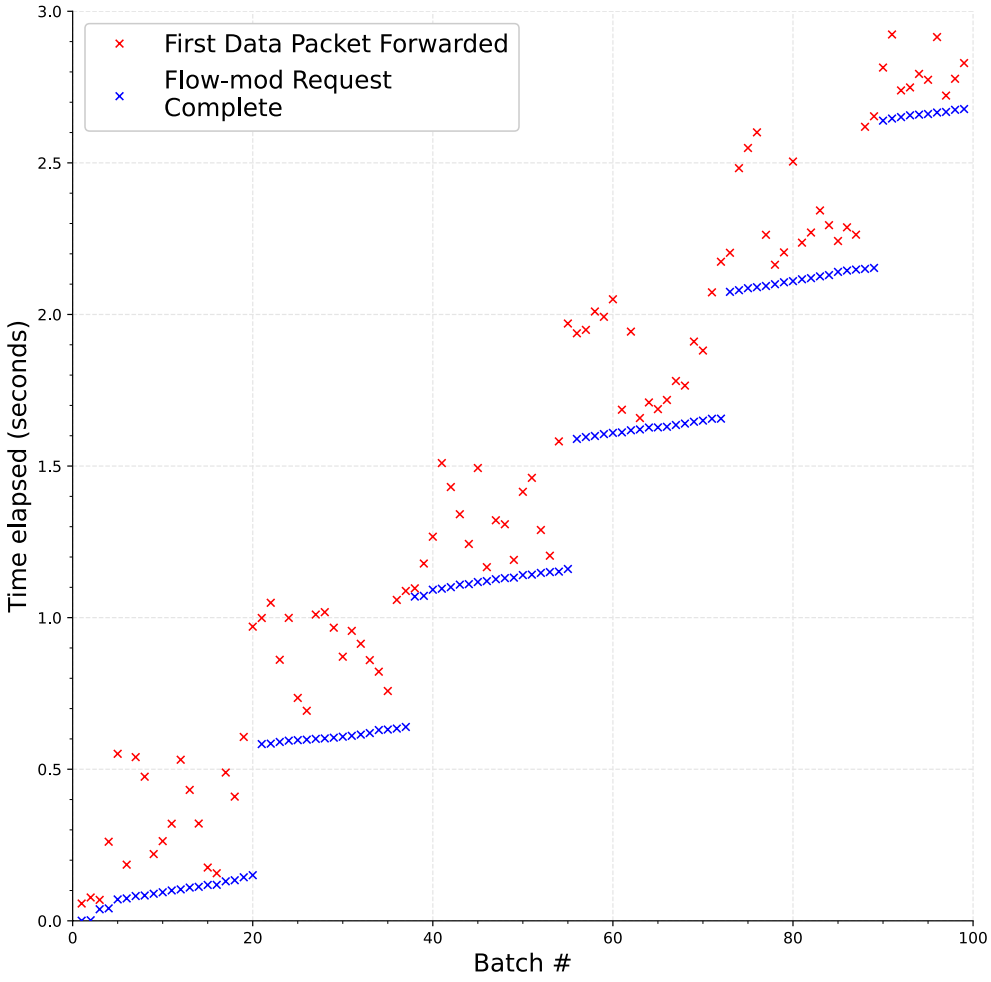
Table 7.1: Flow-rule Installation Statistics

Switch Type	Mean Delay (ms)	Std. Deviation (ms)
Emulated Hardware	218.3	132
Software	24	18.8

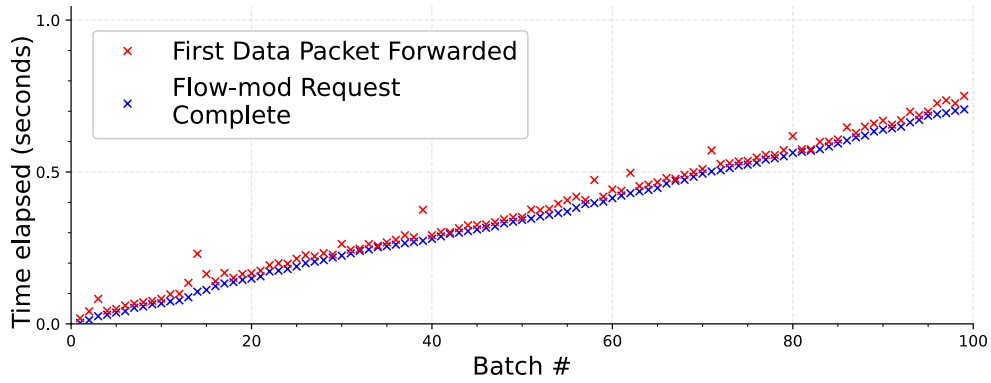
The 'time elapsed' shown on the Y-axes in Figure 7.4 is the time from the start of the experiment. For each individual batch, the data points charted are:

a) The time at which a *BarrierReply* message was received by the controller from the switch, ostensibly confirming that all outstanding control-plane messages have been processed. The flow-rule for which the add_flow request was sent before the *BarrierRequest* message *should* now be in the switch's flow table.

b) The time at which the first packet was received for the flow for which the flow-rule in 1) above was just added. Hence, the earliest time at which it can be *definitively* stated that the flow-rule was present in the switch's flow table.



(a) Emulated H/W switch



(b) S/W switch

Figure 7.4: Flow-rule Installation Delay

Several observations can be made regarding the charts in Figure 7.4:

1) A sequence of batches is processed faster on the software switch than on the emulated hardware switch – at a better than 3:1 ratio. Note that the Y-axis values for the software switch do not extend beyond 1 second because all batches had completed before that time.

2) The delay between a *BarrierReply* and the arrival of the first data packet forwarded following the newly added flow rule is significantly greater on the emulated hardware switch tested as compared to the delay on a software switch.

3) There is a substantial periodic delay observed with the specific type of hardware switch being emulated (a Pica8 P-3290), seen in Figure 7.4 (a) roughly every 18th batch, matching the results in [82, 83] for the hardware switch in question.

The research papers referred to in observation 3 above contain a detailed discussion of the control-plane behaviour of the type of hardware switch used here for comparison, and that of other types of hardware switches. The authors explain that the Pica8 switch maintains a software and hardware flow-table. It switches between updating the software flow-table (and responding to *Flow-Mod* messages) and batch-updating the hardware flow-table at regular intervals. Furthermore, flow-rules are not installed in the hardware flow-table in the order in which they were originally configured in the software flow-table.

For a specific model of hardware switch, the hardware configuration is fixed, and so control-plane performance would be expected to be broadly consistent for all instances of that switch model. With a software switch, the server hardware that it runs on varies from data centre to data centre, or even within the same data centre. For the results shown here, the software switch was tested on a virtual machine running on commodity hardware, therefore the results will be easily matched or exceeded in a production DCN.

7.6.2 Experimental Testbed

To assess the effectiveness of the temporary tunnelling approach, access was obtained to a single DCN hardware switch to use in a physical testbed (illustrated in Figure 7.5). The switch, an Edgecore AS5712-54X running PicOS 2.6.3, was connected to a Cisco C200 M2 server and Dell OptiPlex 760, both running Ubuntu Linux Server 14.04, with Open vSwitch 2.7.3 installed. The server and PC each had a management interface through which the SDN controller exchanged control-plane traffic with Open vSwitch, and a Mellanox 10G fibre interface connected to the hardware switch to carry data-plane traffic.

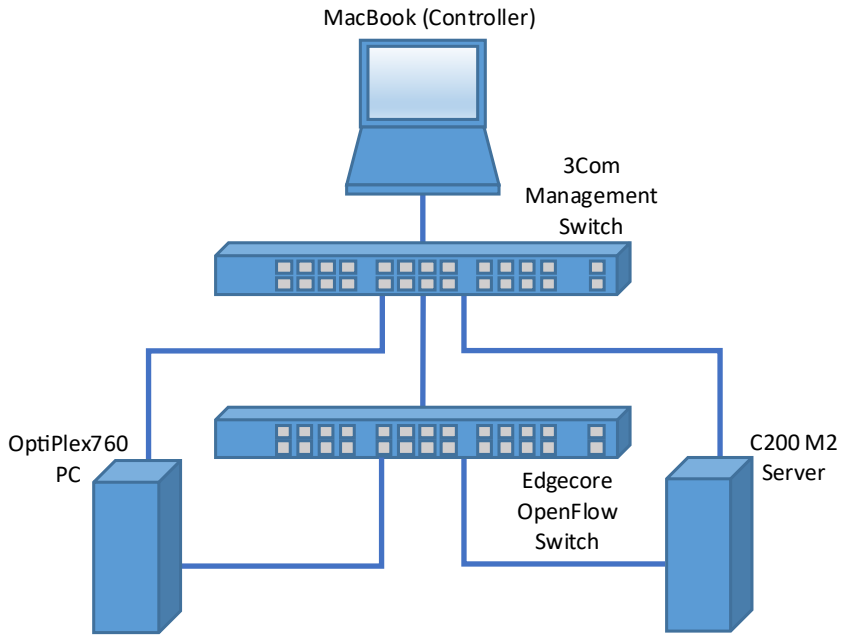


Figure 7.5: Physical Testbed

Debian 8 Linux VMs were hosted on the server and PC with VirtualBox virtualization software, and were run in headless mode for experiments. Each VM had a control network interface used for automated control during experiments, and a data network interface through which TCP connections could be initiated with other VMs. The control interface was linked to the physical management interface on the host machine, as experiments were managed from the computer on which the DCN controller was located. The data network interface was connected to an Open

vSwitch bridge through a Linux tap interface. The bridge was in turn connected to the physical fibre network interface on the host machine.

The SDN controller ran in an Ubuntu VM on an Apple MacBook Pro. The MacBook was connected to an 8-port 100Mb/s 3Com switch, to which the management interfaces of the server, PC and hardware switch were also connected.

7.6.3 Experimental Testing and Results

Experiments were automated via a Python script that interacted with all components through the management switch. The script prepared the Debian VMs (adding static ARP entries for each other VM, starting packet captures), started the SDN controller and packet capturing on the MacBook, and started the client/server TCP connection generator. At the end of the experiment, the same script collected the packet captures and metrics recorded at the controller, then processed and analysed the data and generated the charts presented here.

The controller is coded with knowledge of the DCN topology. When all switches have connected to the controller, it configures a mesh of tunnel rules on all the hardware switches. The mesh requires *n* rules on each leaf switch in a leaf-spine topology, and *k* rules on each spine switch, where *n* is the total number of servers connected by the network and *k* is the number of servers connected to each leaf switch, as detailed in section 7.5. In the experiments described here, the single hardware switch contained two tunnel rules.

Each experiment ran for 20 seconds, long enough to verify whether the approach could maintain deterministic results for a given flow arrival rate beyond the initial start-up. Flow tables were cleared before the experiment started. Experiments were run with a range of TCP connection arrival rates. Note that one TCP connection consists of two unidirectional flows, and is handled as follows: When the first packet of a TCP connection arrives at a software switch, since there are no matching rules it is forwarded to the controller. The controller then configures rules on the relevant switches on both the outgoing and return paths to allow the connection to be established as quickly as possible.

To evaluate the approach, experiments were run where:

1) flows were initially tunnelled, and later migrated to non-tunnelled paths.
2) flows were not tunnelled at all.
3) flows were tunnelled but never migrated from the tunnels.

Figure 7.6 shows TCP connection setup delay with rates of 30, 70 and 90 new connections per second, for the three scenarios listed above. To measure connection setup delay, the time is recorded at which the client sends a TCP SYN packet, and the time that it receives a SYN/ACK in return. While this measure disregards the time required for the final ACK of the TCP 3-way handshake, it captures the time taken for the flow-rules for outbound and inbound paths to be configured.

In the three charts in Figure 7.6, the TCP connection delay rises dramatically (note that the y-axis is plotted on a log scale) for non-tunnelled operation – i.e. when flow-rules must be added to the hardware switch for each arriving connection. Even for a relatively low arrival rate of 30 connections per second, the delay incurred as the hardware switch queues up requests to add flow rules causes the TCP connection setup delay to rise exponentially. As expected, the situation only deteriorates with higher arrival rates of 70 and 90 TCP connections per second.

In contrast, the performance for ‘tunnelled without migration’ is maintained at a reasonably stable delay of around 10ms at an arrival rate of up to 70 TCP connections per second. With this approach, only the software switches are required to add flow rules before the SYN packet can reach the server, and the SYN/ACK response can get to the client. In Figure 7.6(c), there is an anomaly in the results for ‘tunnelled without migration’ near the end of the experiment, coming up to TCP connection #1400. The raw data indicates that this was a short-lived anomaly, and the connection delay appears to be returning to the levels observed before the anomaly, but it is too close to the end of the experiment to be definitive.

A likely reason for the anomaly is another process running on the server or PC causing the controller to be briefly starved of resources.

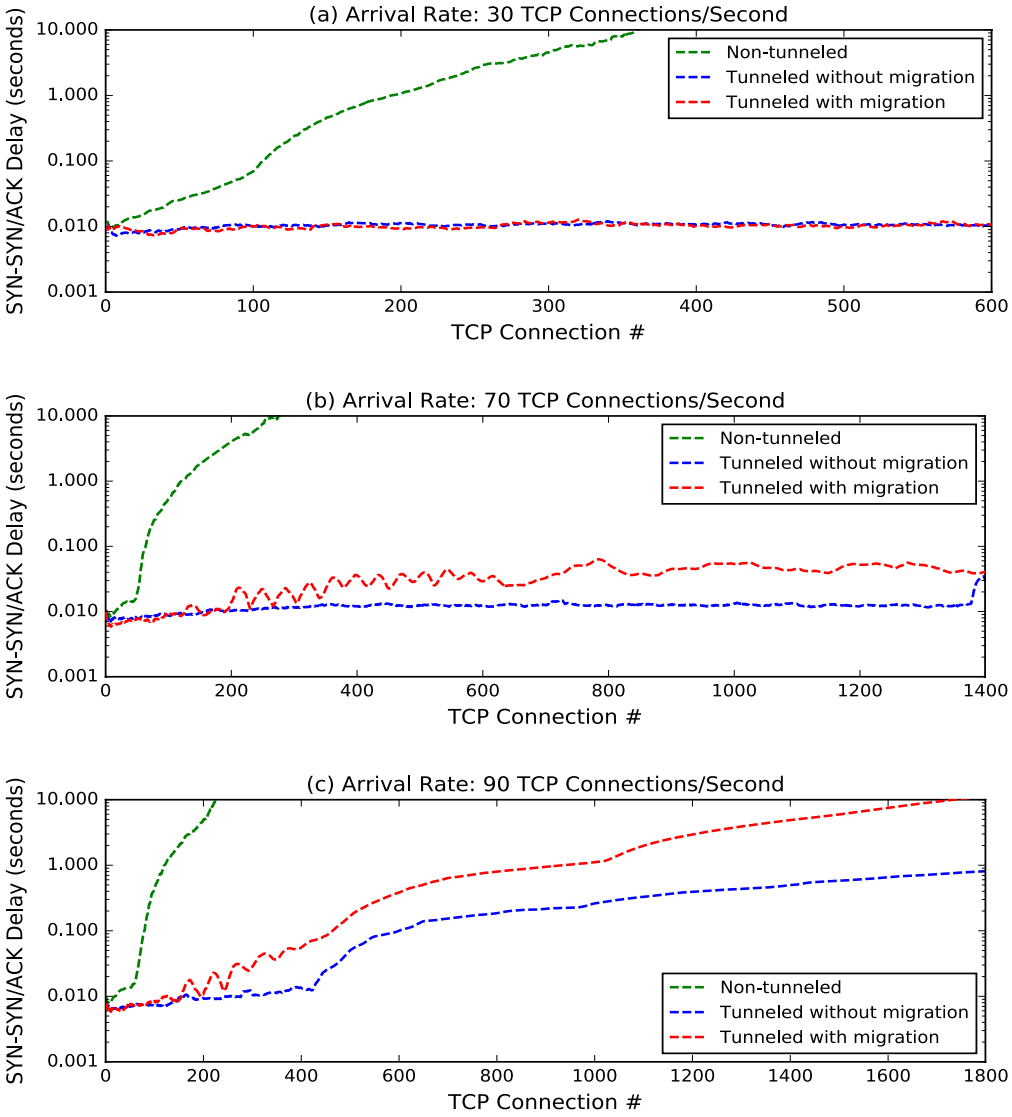


Figure 7.6: Rolling mean TCP connection setup delay at specific flow arrival rates

The 'tunnelled with migration' approach performs well at an arrival rate of 30 TCP connections per second, with delay comparable to 'tunnelled without migration' in Figure 7.6(a). The cost of migrating flows from tunnelled to non-tunnelled paths starts to weigh in Figure 7.6(b), but note that the TCP connection setup latency is maintained at less than 100ms for the full experiment for 'tunnelled with migration'. While migrating flows incurs a cost for TCP connection setup, clearly

that cost can be balanced out over the lifetime of a connection through utilisation of redundant paths and by the elimination of tunnelling overhead after a flow is migrated.

While both mice flows and elephant flows (as short- and long-lived flows are often labelled respectively) benefit from the short setup latency achieved by tunnelling, it is elephant flows that best mitigate the overhead of migration to de-tunnelled paths. As future work, methods devised by other researchers [151, 152] could be employed to identify elephant flows for de-tunnelling, leaving mice flows to finish on the tunnelled paths.

The results in Figure 7.6(c) for an arrival rate of 90 connections per second, with TCP connection setup delay rising without an upper limit for all scenarios, show that the capabilities of the implementation are exceeded. Having profiled the controller code, this is because as new flows arrive, the data structure tracking new and currently tunnelled flows grows, and takes longer to search to make sure that a packet received as part of *PacketIn* message doesn't belong to a flow that is already being actively managed for tunnelling purposes. The reason for tracking flows is so that repeated *PacketIn* messages for the same flow do not result in multiple *Flow_Mod* requests being sent to switches, thereby worsening setup latency. The search time could be reduced by optimising the code, and by using a language other than Python. Clearly, there will still be a point at which the number of tracked flows limits the TCP connection arrival rate that can be supported. However, the point to note is that the demonstrated performance is more than double the limit some hardware switches would otherwise impose on a DCN.

For comparison, the authors of [113] found a per-switch arrival rate of up to 60 flows per second (equivalent to 30 TCP connections per second) in DCNs they analysed – the implementation results presented above show that rate has been surpassed. More recently, a figure of up to 500 flows per second at a single DCN switch has been quoted in [150] by a hyperscale data-centre operator. As Open

vSwitch is quoted as supporting a flow-rule update rate of 42,000 per second [153], a target of 500 flows per second is certainly achievable.

7.7 Conclusions

The work described in this chapter set out to address a weakness of some hardware SDN switches that affects their control-plane performance. Many hardware switches fall behind when receiving flow-rule modification messages at a high rate. Moreover, when a switch responds to say it has implemented the requested modification, the change may not take effect at the data-plane for some time. The performance issue is of particular concern in a multi-tenant DCN environment where switch flow-tables require frequent updates to reflect tenant churn and changing tenant requirements.

An approach to address that weakness was devised, based on the insight that software switches can be immune to the control-plane issue, and the knowledge that software switches are normally the first and last nodes in a path between tenant VMs communicating within a data centre. The claim of immunity for software switches was verified, by testing Open vSwitch under the same conditions as a range of hardware switches had been previously tested by other researchers. This is believed to have been the first comparative test of OpenFlow control-plane performance between software and hardware switches.

The approach devised was to temporarily tunnel flows between software switches, across hardware switches that were preconfigured with tunnel forwarding rules. This provides fast-start for a flow, while allowing time for flow-specific rules to be configured on hardware switches to establish the desired path. Following path verification with probe packets, the flow is 'detunnelled' – migrated to the non-tunnelled path which is used until the flow finishes. The approach was implemented and tested, and the results show that, using temporary tunnelling, the implementation supported a flow arrival rate better than twice that of not tunnelling flows at all.

8 Conclusions and Future Work

8.1 Summary

The focus of this thesis is leveraging the use of Software-Defined Networking (SDN) in a Data Centre Network (DCN) for improved network management. DCNs are complex environments, and the requirements placed on a DCN in a multi-tenant cloud data centre can only be met through the application of SDN. Those requirements include responding in a timely manner to changes tenants make through their dashboard that result in changes to the DCN configuration, ensuring reachability between tenant results, isolation between the resources of different tenants, maintaining availability in case of congestion or failure within the DCN. SDN enables the agility needed to respond to the requirements, but also brings its own challenges to the operator's task of managing the network.

A major challenge for a DCN operator is how to analyse, query, and simply keep track of the changing configuration of their network. In a multi-tenant data centre, the network configuration changes constantly, even if the overall policy is consistent, due to tenants arriving, leaving, and changing their requirements. It is essential for the operator to be able to review past configurations for the purposes of, for example, troubleshooting, auditing, and resource planning. That requirement has been answered in this thesis with models (an emulated copy, and an ontological model) of their DCN that an operator can engage with. In the case of the emulated copy of their DCN, the engagement can be through the use of the tools the operator uses on the physical network to test reachability, path, flow-table contents – using for example, ping, traceroute, ovs-vsctl. The ontological model can be queried to extract key information about historical events, configuration, and behaviour of the DCN.

A further challenge is the control-plane latency of some hardware OpenFlow SDN switches, and the impact that latency can have on the accuracy of the message logs used to create the models referred to above. Queues of flow-rule modification

requests can build up quickly, increasing flow-start times to levels unacceptable to the DCN operator and tenants. Moreover, the response from a switch stating that a flow-rule modification has been completed may be sent before the change has taken effect at the data-plane. Several factors contribute to the delay – mostly the time it takes to update the TCAM (Ternary Content Addressable Memory) tables where rules are normally stored, but also underpowered CPUs on hardware switches and switch operating system design choices. Software switches do not suffer the same control-plane delays, as they maintain flow-tables in RAM on servers where adequate CPU resources can be assigned. Hypervisors on servers hosting VMs in a DCN incorporate a software switch to provide connectivity for VMs to each other and to external communication partners through the server's NIC. This knowledge inspired the idea of temporarily tunnelling flows that require reactive treatment, forwarding packet of a flow across tunnels between software switches until the hardware switches have processed the relevant flow-rule modification requests and until the non-tunnelled path has been verified. The idea was implemented and demonstrated to be effective.

8.2 Key Contributions

These key contributions were made as additions to the body of research on the application of SDN in DCNs:

1. A survey was made of literature published in the area of SDN for DCNs, catalogued from the point of view of DCN management. The survey contributed a taxonomy of prior work and identified key research opportunities. Several of those opportunities were selected as targets for advancement.

2. A methodology was presented that achieves an accurate reproduction of a DCN, in terms of its topology and state, for any time in its recorded history. The DCN is reproduced in emulated form, using information sourced only from a log of control-plane messages that have been passively captured from the network for an duration of weeks, months, or years. Prior

approaches were not passive, did not scale to quantity of data collected for a full historical record, and could not reproduce network state without a target network being provided. The methodology formed the basis for a system architecture that was realised in a software-defined DCN testbed.

3. An optimisation technique was applied to the methodology mentioned in 2. above, to ensure a DCN could be reproduced within a bounded time interval. This is achieved through the creation of snapshots at intervals in the DCN history covered by the control-plane message log. A snapshot contains the topology and state information for a specific timepoint, and is augmented with message log data as necessary to reproduce the state for any timepoint chosen by the DCN operator.

4. Methods to create a temporal and topological model of a DCN were presented. The purpose of the model is to provide a target for DCN operator queries about past events and behaviour of their network. An ontological approach was used, as concepts in an ontology can have a temporal dimension, with attributes including timepoints and time ranges. Having populated the new ontologies with data from a log of control-plane messages, sample queries against the ontologies illustrated their utility to a DCN operator.

5. An approach was set out for devising new, more complex queries to apply against the DCN model referred to in 4. above. New queries are prompted by a DCN operator's need for insight into historical events and behaviours. Addressing complex queries requires new concepts to be added to the ontologies, constructed on top of existing concepts, and inferred from instances of those concepts and/or the results of other queries. The process of adding new complex queries was documented and demonstrated, with examples shown of new concepts being added to the ontologies, enriching the ontologies, and providing targets for further queries.

6. To address control-plane latency issues with hardware OpenFlow DCN switches, a novel scheme was proposed and demonstrated that takes advantage of the placement and much better control-plane performance of software switches. By establishing temporary tunnels between software switches across the hardware switches in the DCN, the delay in updating hardware switch flow-tables is masked. Temporarily en-tunnelled flows may be de-tunnelled as soon as the flow-tables are verified to have been updated on the hardware switches.

Together, the novel contributions listed above have realised the overall goal of the thesis: to efficiently capture and store control-plane messages on a DCN, and to use those messages to build accurate models of the network from which they were logged, to support sophisticated network management query facilities for DCN operators.

8.3 Future Work

The contributions of this work, along with the system implementation, provide a firm foundation for future research and opportunities for additional novel contributions to the state-of-the-art. This is exemplified by the following paragraphs.

8.3.1 Developing a DSL for Querying and Inferencing

The process described in Chapter 6 for adding a new query has a number of steps, chiefly: 1) define new concepts and relationships, 2) encode the new query. In the current implementation, a query is encoded in SPARQL or Prolog if it is straightforward, or a combination of SPARQL, Prolog and Lisp if it is complex. The experience of encoding queries has been that blocks of code are re-used, leading the idea that query encoding would lend itself use of a domain-specific language (DSL).

Use of DSL would speed up the development of new queries by reducing the encoding effort – for example, a re-usable block of code would be represented by a primitive of the language. It would also reduce the likelihood of implementation

bugs since less code would be hand-written. Moreover, a well-designed DSL should reduce the number of logic errors, since checks and safeguards can be built into the language to catch such errors. It may be feasible for non-programmers (for example, DCN operators themselves) to write queries using the DSL.

A DSL needs a grammar and a vocabulary. Developing a new language is a challenging task, usually performed by a language engineer. However, use of an ontology can shortcut the process of DSL development, as the ontology provides a vocabulary (concepts map to nouns, relationships to verbs) and automated tools exist [154] to generate a grammar from an ontology. While saving significant effort, the automated tools will only provide a rudimentary DSL that will require manual verification and refinement with additional syntax rules. For example, the syntax that allows a query to be framed, or inferences to be made. Inferencing may need to be underpinned by the blocks of code mentioned above.

Some consideration would need to be given to a dynamic relationship between the ontology and the new DSL. That is, how the DSL is updated as new concepts and relationships are added to the ontology. Also, if the DSL itself allows new concepts and relationships to be defined, and instances of the new concepts to be inferred, the issue of whether and how those new components can be migrated back into an updated ontology.

8.3.2 Extending the Ontologies to Support Other Sources of Control-Plane or Network Management Data

SDN encompasses other control-plane protocols in addition to OpenFlow. For example, as mentioned in Chapter 3, NetConf, OVSDB and BGP have been used. The separation of the OpenFlow Message ontology from the DCN History ontology, as illustrated in Chapter 5, was intentional, as it provided the option for the future to underpin the DCN History ontology with ontologies for other control-plane protocols or other sources of information. Clearly, creating such ontologies would allow an implementation to work with relevant control-plane messages collected on a non-OpenFlow software-defined DCN. Furthermore, it may be feasible for it to operate on a DCN where more than one SDN protocol is in use – either because

different devices are controlled with different protocols, or because an SDN controller uses multiple protocols to communicate with individual devices. An advantage of having control-plane information from multiple sources is that it may mean that the history of the DCN is richer in detail, providing a more useful target for operator queries.

There will inevitably be some duplication in the multiple sources of control-plane information, and a method for identifying and dealing with duplication will need to be produced. There is also likely be contradictory data between different sources, and methods to address this will be a challenge to formulate. For example, timing information for events signalled by different control-plane protocols can be expected to differ.

An initial step for this initiative would be to devise ontologies for relevant other control-plane protocol(s). Deciding how to handle duplication and discrepancies requires an analysis of the information sources, the granularity and accuracy of data provided by those sources, and the semantics of the contained data. A decision to be made is whether to populate the DCN history ontology with inferred instances from all sources, or whether to selectively populate using only the most trusted information.

8.3.3 Answering Queries about DCN History for PISA Devices

Programmable forwarding devices, such as P4 or PISA (Protocol Independent Switch Architecture) switches, are becoming more of a focus for research, in addition to being available for commercial use. A DCN operator will want to know about the historical state and events on their network whether the switches in the network are programmable or not, so clearly it is of interest to investigate the application of the approaches devised here to programmable switches also.

P4 switches must be programmed to function. The first challenge is to define a method to model P4 programs so that they can be input to the process used to construct the DCN history. The next challenge is to devise the process of constructing the DCN history, combining the P4 program model with other

components. The other components are models of switches and the DCN topology, and temporal information regarding topology changes and the timepoints at which switches have been reprogrammed.

It has been shown [155] that Java program code can be represented in an ontology, for querying purposes. This approach may be applicable to P4 code also. An alternative approach could leverage work on compiling P4 programs to Datalog [156] rules, with the extra challenge of combining the Datalog component with temporal and topological ontology data for query purposes. P4 switches provide packet telemetry data, which should be useful for topology discovery, and flow statistics from which flow start and end time-points can be inferred and used to populate the DCN history ontology with instances.

8.3.4 Applying Machine-Learning to Control-Plane Message Logs

LogSnap (from Chapter 3) is designed to collect control-plane messages in a DCN over long periods, generating very large logs of data in the process. Information is extracted from the messages in the logs, then used in LogSnap to reproduce a DCN, and further used to populate an ontology against which queries are run. The message logs would be an ideal dataset against which to apply machine learning (ML) approaches. Some potential applications for this approach include:

- Predicting flow arrival and traffic levels in the DCN so that an SDN controller can proactively avoid congestion, or meet QoS requirements
- Identifying uncharacteristic flows, control-plane messages, control-plane messages sequences for security purposes – possibly identifying rogue SDN controllers, compromised switches, malicious SDN controller apps, or untrustworthy tenants.
- Performing network diagnostics, identifying unstable controller instances, and feeding the information back to restart those instances.

A pre-requisite is a collection of control-plane message logs, ideally from several multi-tenant data centres, where different SDN controllers and an assortment of SDN switches are in use. The first challenge is to pinpoint the relevant information to be extracted from the control-plane messages, and in what form it should be presented to the ML application. The intended goal(s) of the ML application will dictate what input information is required. The next challenge is to identify the most appropriate ML technique. A survey of ML techniques used with SDN in recent years [157] will help to inform this step. Implementation in a lab environment will be sufficient to demonstrate the benefits of the approach.

References

[1] "Cisco Global Cloud Index: Forecast and Methodology, 2011–2016." https://web.archive.org/web/20121126231230/http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns1175/Cloud_Index_White_Paper.pdf (accessed August, 2022).

[2] "Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper." https://web.archive.org/web/20180215132349/https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf (accessed August, 2022).

[3] "Pipeline of Over 300 New Hyperscale Data Centers Drives Healthy Growth Forecasts." https://www.srgresearch.com/articles/pipeline-of-over-300-new-hyperscale-data-centers-drives-healthy-growth-forecasts (accessed August, 2022).

[4] "Hyperscale Data Center Capacity Doubles in Under Four Years; the US Still Accounts for Half." https://www.srgresearch.com/articles/as-hyperscale-data-center-capacity-doubles-in-under-four-years-the-us-still-accounts-for-half-of-the-total (accessed August, 2022).

[5] *SolarWinds Orion*. Accessed: December 2022. [Online]. Available: https://www.solarwinds.com/solutions/orion

[6] A. Roy *et al.*, "Cloud Datacenter SDN Monitoring: Experiences and challenges," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 464-470.

[7] "OpenFlow Conformance Certification." https://opennetworking.org/product-certification/ (accessed August, 2022).

[8] K. Greene. "TR10: Software-Defined Networking." http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/ (accessed August, 2022).

[9] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.,* vol. 38, no. 2, pp. 69-74, 2008, doi: 10.1145/1355734.1355746.

[10] "Open Networking Foundation." https://opennetworking.org (accessed August, 2022).

[11] S. Ahearne, Y. Verbishchuk, C. Sreenan, and F. Gunning, "Software defined control of tunable optical transceivers using NETCONF and YANG," in *2018 European Conference on Networks and Communications (EuCNC)*, 2018: IEEE, pp. 1-86.

[12] D. Sanvito, D. Moro, M. Gulli, I. Filippini, A. Capone, and A. Campanella, "ONOS Intent Monitor and Reroute service: enabling plug&play routing logic," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018: IEEE, pp. 272-276.

[13] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review,* vol. 44, no. 2, pp. 87-98, 2014.

[14] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE,* vol. 103, no. 1, pp. 14-76, 2014, Art no. 6994333, doi: 10.1109/JPROC.2014.2371999.

[15] M. He, A. M. Alba, A. Basta, A. Blenk, and W. Kellerer, "Flexibility in softwarized networks: Classifications and research challenges," *IEEE Communications Surveys & Tutorials,* vol. 21, no. 3, pp. 2600-2636, 2019.
[16] L. Yang, B. Ng, W. K. Seah, L. Groves, and D. Singh, "A survey on network forwarding in Software-Defined Networking," *Journal of Network and Computer Applications,* vol. 176, 2021.
[17] S. Ahmad and A. H. Mir, "Scalability, consistency, reliability and security in SDN controllers: a survey of diverse SDN controllers," *Journal of Network and Systems Management,* vol. 29, pp. 1-59, 2021.
[18] A. L. Stancu, S. Halunga, A. Vulpe, G. Suciu, O. Fratu, and E. C. Popovici, "A comparison between several Software Defined Networking controllers," in *2015 12th international conference on telecommunication in modern satellite, cable and broadcasting services (TELSIKS)*, 2015: IEEE, pp. 223-226.
[19] L. Zhu *et al.*, "SDN controllers: A comprehensive analysis and performance evaluation study," *ACM Computing Surveys (CSUR),* vol. 53, no. 6, pp. 1-40, 2020.
[20] P. T. Congdon, P. Mohapatra, M. Farrens, and V. Akella, "Simultaneously reducing latency and power consumption in openflow switches," *IEEE/ACM Transactions On Networking,* vol. 22, no. 3, pp. 1007-1020, 2013.
[21] P. Goransson, C. Black, and T. Culver, *Software Defined Networks: A Comprehensive Approach*, 2nd ed. Morgan Kaufmann, 2016.
[22] E. L. Fernandes *et al.*, "The road to BOFUSS: The basic OpenFlow userspace software switch," *Journal of Network and Computer Applications,* vol. 165, p. 102685, 2020.
[23] D. B. Hoang and M. Pham, "On software-defined networking and the design of SDN controllers," in *2015 6th International Conference on the Network of the Future (NOF)*, 2015: IEEE, pp. 1-3.
[24] L. C. Costa *et al.*, "OpenFlow data planes performance evaluation," *Performance Evaluation,* vol. 147, p. 102194, 2021.
[25] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE communications surveys & tutorials,* vol. 16, no. 1, pp. 493-512, 2013.
[26] *OpenFlow Switch Specification Version 1.5.1, Standard*. Open Networking Foundation, 2015.
[27] Y. Liu, J. K. Muppala, M. Veeraraghavan, D. Lin, and M. Hamdi, *Data center networks: Topologies, architectures and fault-tolerance characteristics*. Springer Science & Business Media, 2013.
[28] L. Guo and P. Congdon, "White Paper-IEEE SA Industry Connections--IEEE 802 Nendica Report: Intelligent Lossless Data Center Networks," 2021.
[29] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 78-85.
[30] *Open Network Install Environment (ONIE)*. Accessed: December 2022. [Online]. Available: https://opencomputeproject.github.io/onie/index.html
[31] *Open Network Linux (ONL)*. Accessed: December 2022. [Online]. Available: https://github.com/opencomputeproject/OpenNetworkLinux
[32] *Cisco Data Center Infrastructure 2.5 Design Guide*. Cisco Systems, 2007.
[33] M. Alizadeh and T. Edsall, "On the data path performance of leaf-spine datacenter fabrics," in *2013 IEEE 21st annual symposium on high-performance interconnects*, 2013: IEEE, pp. 71-74.

[34] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review,* vol. 38, no. 4, pp. 63-74, 2008.

[35] R. R. Reyes and T. Bauschert, "Infrastructure cost comparison of intra-data centre network architectures," in *2018 IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2018: IEEE, pp. 1-7.

[36] *Cisco and Hitachi Adaptive Solutions for Converged Infrastructure with Cisco ACI Design Guide*. Cisco Systems, 2019.

[37] A. Andreyev, X. Wang, and A. Eckert, "Reinventing Facebook's data center network," *Facebook Engineering,* 2019.

[38] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009, pp. 51-62.

[39] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, 2017: IEEE, pp. 64-73.

[40] J. Sherwin and C. J. Sreenan, "Software-Defined Networking for Data Center Network Management - A Survey and Open Issues," University College Cork, Ireland, Technical Report UCC-CS-2016-24-06, 2016.

[41] K. Hyojoon and N. Feamster, "Improving network management with software defined networking," *Communications Magazine, IEEE,* vol. 51, no. 2, pp. 114-119, 2013, doi: 10.1109/MCOM.2013.6461195.

[42] M. Shamseddine, I. Elhajj, A. Chehab, A. Kayssi, and W. Itani, "NCaaS: network configuration as a service in SDN-driven cloud architectures," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 448-454.

[43] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "One big switch" abstraction in software-defined networks," in *CoNEXT 2013 - Proceedings of the 2013 ACM International Conference on Emerging Networking Experiments and Technologies*, 2013, pp. 13-24. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84893398564&partnerID=40&md5=198adea3b54a99cfcb044ee2f143cc52

[44] M. Arumugam *et al.*, "Bluebird: High-performance {SDN} for Bare-metal Cloud Services," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 355-370.

[45] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic resource allocation for software-defined measurement," in *SIGCOMM 2014 - Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, 2014, pp. 419-430, doi: 10.1145/2619239.2626291. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84907377109&partnerID=40&md5=61a23e683e79618fbafb5aa01ad1f1c9

[46] J. Zheng, Q. Li, G. Gu, J. Cao, D. K. Yau, and J. Wu, "Realtime DDoS defense using COTS SDN switches via adaptive correlation analysis," *IEEE Transactions on Information Forensics and Security,* vol. 13, no. 7, pp. 1838-1853, 2018.

[47] N. Foster, N. McKeown, J. Rexford, G. Parulkar, L. Peterson, and O. Sunay, "Using deep programmability to put network owners in control," *SIGCOMM Comput. Commun. Rev.,* vol. 50, no. 4, pp. 82–88, 2020, doi: 10.1145/3431832.3431842.

[48] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on*

the applications, technologies, architectures, and protocols for computer communication, 2020, pp. 226-239.

[49] W. Wang, P. Tammana, A. Chen, and T. S. E. Ng, "Grasp the Root Causes in the Data Plane: Diagnosing Latency Problems with SpiderMon," presented at the Proceedings of the Symposium on SDN Research, San Jose, CA, USA, 2020. [Online]. Available: https://doi.org/10.1145/3373360.3380835.

[50] J. Rasley *et al.*, "Planck: Millisecond-scale monitoring and control for commodity networks," in *SIGCOMM 2014 - Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, 2014, pp. 407-418, doi: 10.1145/2619239.2626310. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84907359489&partnerID=40&md5=a253059141e76db2439c39c3a154c74c

[51] N. Rikhtegar, M. Keshtgari, O. Bushehrian, and G. Pujolle, "BiTE: a dynamic bi-level traffic engineering model for load balancing and energy efficiency in data center networks," *Applied Intelligence,* vol. 51, no. 7, pp. 4623-4648, 2021.

[52] M. Bastam, M. Sabaei, and R. Yousefpour, "A scalable traffic engineering technique in an SDN-based data center network," *Transactions on Emerging Telecommunications Technologies,* vol. 29, no. 2, p. e3268, 2018.

[53] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," in *SIGCOMM 2014 - Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, 2014, pp. 27-38, doi: 10.1145/2619239.2626317. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84907321982&partnerID=40&md5=10a09583e46cf66edae37677e6b39594

[54] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 15-28.

[55] H. Mekky, F. Hao, S. Mukherjee, Z. L. Zhang, and T. V. Lakshman, "Application-aware data plane processing in SDN," in *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, 2014, pp. 13-18, doi: 10.1145/2620728.2620735. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84907009902&partnerID=40&md5=18d5f099f20d32fd240e1ae6048fed0d

[56] H. Hantouti, N. Benamar, T. Taleb, and A. Laghrissi, "Traffic steering for service function chaining," *IEEE Communications Surveys & Tutorials,* vol. 21, no. 1, pp. 487-507, 2018.

[57] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee, "Application-aware virtual machine migration in data centers," in *2011 Proceedings IEEE INFOCOM*, 2011: IEEE, pp. 66-70.

[58] G. Shen, Q. Li, S. Ai, Y. Jiang, M. Xu, and X. Jia, "How Powerful Switches Should be Deployed: A Precise Estimation Based on Queuing Theory," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 29 April-2 May 2019 2019, pp. 811-819, doi: 10.1109/INFOCOM.2019.8737629.

[59] S. Hu *et al.*, "Explicit path control in commodity data centers: design and applications," presented at the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, Oakland, CA, 2015.

[60] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of SDNs," in *SIGCOMM 2013 - Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2013, pp. 327-338, doi: 10.1145/2486001.2486003. [Online]. Available:

http://www.scopus.com/inward/record.url?eid=2-s2.0-84883299005&partnerID=40&md5=46e67c26a2f7f040faf692e5149a6032

[61] D. Arora, T. Benson, and J. Rexford, "ProActive routing in scalable data centers with PARIS," in *DCC 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Distributed Cloud Computing*, 2014, pp. 5-10, doi: 10.1145/2627566.2627571. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84907327718&partnerID=40&md5=88f3628d70034e18431ff1924f7d144c

[62] N. Kim and J. Kim, "Building netopen networking services over openflow-based programmable networks," in *The International Conference on Information Networking 2011 (ICOIN2011)*, 2011: IEEE, pp. 525-529.

[63] R. Cohen *et al.*, "An intent-based approach for network virtualization," in *Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management, IM 2013*, 2013, pp. 42-50. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84883484060&partnerID=40&md5=33835fe721726b09a32273e97e948440

[64] A. Alimohammadifar *et al.*, "Stealthy probing-based verification (SPV): An active approach to defending software defined networks against topology poisoning attacks," in *European Symposium on Research in Computer Security*, 2018: Springer, pp. 463-484.

[65] S. Lee, J. Kim, S. Shin, P. Porras, and V. Yegneswaran, "Athena: A Framework for Scalable Anomaly Detection in Software-Defined Networks," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 26-29 June 2017 2017, pp. 249-260, doi: 10.1109/DSN.2017.42.

[66] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: enabling record and replay troubleshooting for networks," in *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, OR, 2011, 2002210: USENIX Association, pp. 29-29.

[67] Q. Huang *et al.*, "Sketchvisor: Robust network measurement for software packet processing," in *SIGCOMM 2017 - Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 113-126, doi: 10.1145/3098822.3098831. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85029407852&doi=10.1145/3098822.3098831&partnerID=40&md5=683df2179af3a898b2c952c6643c5b56

[68] N. Handigol, B. Heller, V. Jeyakumar, D. Mazi, and N. McKeown, "I know what your packet did last hop: using packet histories to troubleshoot networks," presented at the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, Seattle, WA, 2014.

[69] W. Kim, J. Li, J. W. K. Hong, and Y. J. Suh, "OFMon: OpenFlow monitoring system in ONOS controllers," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 6-10 June 2016 2016, pp. 397-402, doi: 10.1109/NETSOFT.2016.7502474.

[70] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, 2014, doi: 10.1109/NOMS.2014.6838228. [Online]. Available: https://www.scopus.com/inward/record.url?eid=2-s2.0-84904154017&partnerID=40&md5=9393528e282ecedfde9cfd7da190aeb9

[71] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic Matrix Estimator for OpenFlow Networks," in *Passive and Active Measurement Conference (PAM 2010)*, 2010.

[72] B. Siniarski, C. Olariu, P. Perry, T. Parsons, and J. Murphy, "Real-time monitoring of SDN networks using non-invasive cloud-based logging platforms," in *Personal, Indoor, and Mobile Radio Communications (PIMRC), 2016 IEEE 27th Annual International Symposium on*, 2016: IEEE, pp. 1-6.

[73] *Floodlight Open Source OpenFlow Controller*. Accessed: December 2022. [Online]. Available: https://github.com/floodlight/floodlight

[74] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era," presented at the Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, Canada, 2018.

[75] P. Berde *et al.*, "ONOS: towards an open, distributed SDN OS," presented at the Proceedings of the third workshop on Hot topics in software defined networking, Chicago, Illinois, USA, 2014.

[76] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014: IEEE, pp. 1-6.

[77] W. Yun, X. Zhang, Z. Li, H. Liu, and M. Han, "Knowledge modeling: A survey of processes and techniques," *International Journal of Intelligent Systems,* vol. 36, no. 4, pp. 1686-1720, 2021.

[78] C. Rotsos *et al.*, "ReasoNet: Inferring Network Policies Using Ontologies," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 25-29 June 2018 2018, pp. 159-167, doi: 10.1109/NETSOFT.2018.8460050.

[79] F. Benayas, Á. Carrera, M. García-Amado, and C. A. Iglesias, "A semantic data lake framework for autonomous fault management in SDN environments," *Transactions on Emerging Telecommunications Technologies,* p. e3629, 2019, doi: 10.1002/ett.3629.

[80] M. Perry, F. Hakimpour, and A. Sheth, "Analyzing theme, space, and time: an ontology-based approach," in *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, 2006, pp. 147-154.

[81] E. Vinarskii, J. Lopez, N. Kushik, N. Yevtushenko, and D. Zeghlache, "A model checking based approach for detecting SDN races," in *IFIP International Conference on Testing Software and Systems*, 2019: Springer, pp. 194-211.

[82] M. Kuźniar, P. Perešíni, and D. Kostić. *What you need to know about SDN flow tables, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8995, pp. 347-359, 2015.

[83] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches," *Computer Networks,* vol. 136, pp. 22-36, 2018, doi: https://doi.org/10.1016/j.comnet.2018.02.014.

[84] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "Past: Scalable ethernet for data centers," in *CoNEXT 2012 - Proceedings of the 2012 ACM Conference on Emerging Networking Experiments and Technologies*, 2012, pp. 49-60, doi: 10.1145/2413176.2413183. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84871968802&partnerID=40&md5=a8341cdb6e68ec2c3cdc1c1e1d31ead6

[85] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM'11*, 2011, pp. 254-

265. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-80053141664&partnerID=40&md5=59f14b02e3cbc080139e1dddde10da57

[86] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, Fine-Grained Data Plane Monitoring," in *CoNEXT'15 - Proceedings of the 2015 ACM International Conference on Emerging Network Experiments and Technologies*, 2015.

[87] *libpcap network traffic capture library*. Accessed: December 2022. [Online]. Available: https://www.tcpdump.org

[88] *RabbitMQ Open Source Message Broker*. Accessed: December 2022. [Online]. Available: https://www.rabbitmq.com/

[89] *IEEE 1588-2008 Precision Time Protocol*, IEEE, 2008. [Online]. Available: https://www.nist.gov/el/intelligent-systems-division-73500/introduction-ieee-1588

[90] D. A. Popescu and A. W. Moore, "Measuring network conditions in data centers using the precision time protocol," *IEEE Transactions on Network and Service Management,* vol. 18, no. 3, pp. 3753-3770, 2021.

[91] *Network Time Protocol*. Accessed: December 2022. [Online]. Available: https://www.ntp.org/

[92] *Loxi OpenFlow Protocol Library*. Accessed: December 2022. [Online]. Available: https://github.com/floodlight/loxigen

[93] *Elastic Stack Open Source Distributed Log Management*. [Online]. Available: https://www.elastic.co/products/

[94] *BPF Compiler Collection*. [Online]. Available: https://www.iovisor.org/technology/bcc

[95] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl, "MaxiNet: Distributed Emulation of Software-Defined Networks," in *IFIP Networking 2014 Conference (Networking 2014)*.

[96] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM Workshop on Hot Topics in Networks, Hotnets-9*, 2010, doi: 10.1145/1868447.1868466. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84865538933&partnerID=40&md5=dcfebfa9c17fe6ff026c2b5cfed8e7ca

[97] D. Muelas, J. Ramos, and J. E. L. de Vergara, "Assessing the limits of mininet-based environments for network experimentation," *IEEE Network,* vol. 32, no. 6, pp. 168-176, 2018.

[98] "Vertiv - What is a Hyperscale Data Center?" https://www.vertiv.com/en-emea/about/news-and-insights/articles/educational-articles/what-is-a-hyperscale-data-center/ (accessed May, 2023).

[99] "Parallels - What is a hyperscale data center?" https://www.parallels.com/blogs/ras/hyperscale-data-center/ (accessed May, 2023).

[100] "BMC - What is a Hyperscale Data Center?" https://www.bmc.com/blogs/hyperscale-data-center/ (accessed May, 2023).

[101] O. A. Adeleke, N. Bastin, and D. Gurkan, "Network traffic generation: A survey and methodology," *ACM Computing Surveys (CSUR),* vol. 55, no. 2, pp. 1-23, 2022.

[102] *iPerf: The TCP/UDP bandwidth measurement tool*. Accessed: December 2022. [Online]. Available: https://iperf.fr/iperf-doc.php

[103] *netperf TCP/UDP/sockets/etc performance benchmark*. Accessed: December 2022. [Online]. Available: https://github.com/HewlettPackard/netperf

[104] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks,* vol. 56, no. 15, pp. 3531-3547, 2012.

[105] P. Wette and H. Karl, "DCT2Gen: A traffic generator for data centers," *Computer Communications,* Article vol. 80, pp. 45-58, 2016, doi: 10.1016/j.comcom.2015.12.001.
[106] P. Vizarreta *et al.*, "Assessing the maturity of sdn controllers with software reliability growth models," *IEEE Transactions on Network and Service Management,* vol. 15, no. 3, pp. 1090-1104, 2018.
[107] https://calculator.aws/ (accessed May, 2023).
[108] "ONOS OpenFlow Benchmark - Test 7 ： Flow Setup Rate at Reactive Mode." https://wiki.onosproject.org/pages/viewpage.action?pageId=11864719 (accessed December, 2022).
[109] S. Bazarbayev, M. Hiltunen, K. Joshi, W. H. Sanders, and R. Schlichting, "Content-based scheduling of virtual machines (VMs) in the cloud," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*, 2013: IEEE, pp. 93-101.
[110] F. P. Tso, G. Hamilton, K. Oikonomou, and D. P. Pezaros, "Implementing scalable, network-aware virtual machine migration for cloud data centers," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013: IEEE, pp. 557-564.
[111] A. Magalhaes, J. M. Monteiro, and A. Brayner, "Main memory database recovery: A survey," *ACM Computing Surveys (CSUR),* vol. 54, no. 2, pp. 1-36, 2021.
[112] *PostgreSQL Open Source Database*. [Online]. Available: https://www.postgresql.org/
[113] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, 2010, pp. 267-280, doi: 10.1145/1879141.1879175. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-78650919030&partnerID=40&md5=e322ed8d3f4b061bbc3e303202a49d18
[114] *Cisco Virtualized Multi-Tenant Data Center, Version 2.2 Design Guide*. Cisco Systems, 2013.
[115] J. Hastings, "Primer on ontologies," in *The gene ontology handbook*: Humana Press, New York, NY, 2017, pp. 3-13.
[116] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american,* vol. 284, no. 5, pp. 34-43, 2001.
[117] P. Hitzler, "A review of the semantic web field," *Communications of the ACM,* vol. 64, no. 2, pp. 76-83, 2021.
[118] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins, "What are ontologies, and why do we need them?," *IEEE Intelligent Systems and their applications,* vol. 14, no. 1, pp. 20-26, 1999.
[119] A. C. Khadir, H. Aliane, and A. Guessoum, "Ontology learning: Grand tour and challenges," *Computer Science Review,* vol. 39, p. 100339, 2021.
[120] A. Martinez, M. Yannuzzi, J. E. L. d. Vergara, R. Serral-Gracià, and W. Ramírez, "An Ontology-Based Information Extraction System for bridging the configuration gap in hybrid SDN environments," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 11-15 May 2015 2015, pp. 441-449, doi: 10.1109/INM.2015.7140321.
[121] K. G. Kyriakopoulos, D. J. Parish, and J. N. Whitley, "FlowStats: An ontology based network management tool," in *2015 Second International Conference on Computing Technology and Information Management (ICCTIM)*, 21-23 April 2015 2015, pp. 13-18, doi: 10.1109/ICCTIM.2015.7224586.
[122] "Resource Description Framework," 2014. Accessed: 2022-10-03. [Online]. Available: https://www.w3.org/RDF/

[123] "RDF Schema," 2014. Accessed: 2022-10-03. [Online]. Available: https://www.w3.org/TR/rdf-schema/
[124] "Web Ontology Language (OWL)," 2012. Accessed: 2022-10-03. [Online]. Available: https://www.w3.org/OWL/
[125] F. Alam, M. A. Khan, S. Ali, and S. Khusro, "The jigsaw of resource description framework (RDF) and topic maps serialization formats: A survey," *Proceedings of the Pakistan Academy of Sciences,* vol. 51, no. 2, pp. 101-114, 2014.
[126] C. Buil-Aranda *et al.*, "SPARQL 1.1. Overview. W3C Recommendation.." Accessed: 2021-04-30. [Online]. Available: https://www.w3.org/TR/sparql11-overview/
[127] H. Knublauch, J. A. Hendler, and K. Idehen, "SPIN-overview and motivation," *W3C Member Submission,* vol. 22, p. W3C, 2011.
[128] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A semantic web rule language combining OWL and RuleML," *W3C Member submission,* vol. 21, no. 79, pp. 1-31, 2004.
[129] M. J. O'Connor and A. K. Das, "SQWRL: a query language for OWL," in *OWLED*, 2009, vol. 529, no. 2009.
[130] H. Knublauch and D. Kontokostas, "Shapes constraint language (SHACL)," *W3C Candidate Recommendation,* vol. 11, no. 8, 2017.
[131] C. Atkinson and K. Kiko, "A detailed comparison of UML and OWL," 2005. Accessed: 2022-10-03. [Online]. Available: http://ub-madoc.bib.uni-mannheim.de/1898
[132] J. Zedlitz and N. Luttenberger, "Conceptual modelling in UML and OWL-2," *International Journal on Advances in Software,* vol. 7, no. 1, pp. 182-196, 2014.
[133] J. J. Miller, "Graph database applications and concepts with Neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, 2013, vol. 2324, no. 36.
[134] J. van der Ham, F. Dijkstra, R. Lapacz, and A. Brown, "The Network Markup Language (NML) A Standardized Network Topology Abstraction for Inter-domain and Cross-layer Network Applications," presented at the TNC2013, 2013.
[135] M. Katsumi and M. Fox, "A Logical Design Pattern for Representing Change Over Time in OWL," *Proceedings of the 8th Workshop on Ontology Design and Patterns (WOP 2017)*.
[136] R. N. Carvalho, K. B. Laskey, and P. C. Costa, "PR-OWL–a language for defining probabilistic ontologies," *International Journal of Approximate Reasoning,* vol. 91, pp. 56-79, 2017.
[137] H.-U. Krieger, "Where temporal description logics fail: Representing temporally-changing relationships," in *Annual Conference on Artificial Intelligence*, 2008: Springer, pp. 249-257.
[138] J. H. Gennari *et al.*, "The evolution of Protégé: an environment for knowledge-based systems development," *International Journal of Human-computer studies,* vol. 58, no. 1, pp. 89-123, 2003.
[139] *AllegroGraph*. Accessed: 2021-04-30. [Online]. Available: https://allegrograph.com/products/allegrograph/
[140] *Quicklisp beta*. Accessed: 2021-04-30. [Online]. Available: https://www.quicklisp.org/beta/
[141] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl, "MaxiNet: Distributed Emulation of Software-Defined Networks," presented at the IFIP Networking Conference (Networking 2014), 2014.
[142] R. Aryan, A. Yazidi, P. E. Engelstad, and Ø. Kure, "A general formalism for defining and detecting openflow rule anomalies," in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, 2017: IEEE, pp. 426-434.

[143] C. Rotsos *et al.*, "ReasoNet: Inferring Network Policies Using Ontologies," in *4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 159-167, doi: 10.1109/NETSOFT.2018.8460050.

[144] T. D. P. C. D. Souza, C. E. Rothenberg, M. A. S. Santos, and L. B. D. Paula, "Towards Semantic Network Models via Graph Databases for SDN Applications," in *2015 Fourth European Workshop on Software Defined Networks*, 30 Sept.-2 Oct. 2015 2015, pp. 49-54, doi: 10.1109/EWSDN.2015.60.

[145] K. Thornton *et al.*, "Using Shape Expressions (ShEx) to share RDF data models and to guide curation with rigorous validation," in *European Semantic Web Conference*, 2019: Springer, pp. 606-620.

[146] *pySHACL - A Python validator for SHACL*. [Online]. Available: https://github.com/RDFLib/pySHACL

[147] "Information technology - Elements of management information related to the OSI Network Layer," *International Standards for Business, Government and Society*: International Organization for Standardization, 1998.

[148] K. He *et al.*, "Measuring control plane latency in SDN-enabled switches," presented at the Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, Santa Clara, California, 2015.

[149] B. Pfaff *et al.*, "The design and implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, 2015, pp. 117-130. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84946124798&partnerID=40&md5=d6d042bc7722d2c23196d998e7e8ddd2

[150] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," presented at the Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, 2015.

[151] M. Hamdan *et al.*, "Flow-aware elephant flow detection for software-defined networks," *IEEE Access,* vol. 8, pp. 72585-72597, 2020.

[152] F. Tang, H. Zhang, L. T. Yang, and L. Chen, "Elephant flow detection and load-balanced routing with efficient sampling and classification," *IEEE Transactions on Cloud Computing,* vol. 9, no. 3, pp. 1022-1036, 2019.

[153] A. L. Aliyu, P. Bull, and A. Abdallah, "Performance implication and analysis of the OpenFlow SDN protocol," in *Proceedings - 31st IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2017*, 2017, pp. 391-396, doi: 10.1109/WAINA.2017.101. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85021426061&doi=10.1109%2fWAINA.2017.101&partnerID=40&md5=0609d8df2097bea3d09209f6a3373357

[154] M. J. V. Pereira, J. Fonseca, and P. R. Henriques, "Ontological approach for DSL development," *Computer Languages, Systems & Structures,* vol. 45, pp. 35-52, 2016.

[155] M. Atzeni and M. Atzori, "CodeOntology: RDF-ization of source code," in *International Semantic Web Conference*, 2017: Springer, pp. 20-28.

[156] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in P4 Networks," *Technical Report, Tech. Rep,* 2016.

[157] J. Xie *et al.*, "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges," *IEEE Communications Surveys & Tutorials,* vol. 21, no. 1, pp. 393-430, 2018.