On Consistency and Distribution in Software-defined Networking

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Thomas Kohler

aus Illertissen

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. rer. nat. Jörg Hähner
Tag der mündlichen Prüfung:	15. Juli 2020

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2020

Acknowledgments

Reflecting the whole journey of pursuing a doctoral degree now in its last phase, I feel very much obliged to acknowledge and to thank all the wonderful people involved in enabling or supporting me to do so.

First and foremost I would like to express my deepest gratefulness to my doctoral supervisor Prof. Dr. Kurt Rothermel for giving me the opportunity to conduct my research and pursue my academic career. His continuous guidance and astute feedback marked a cornerstone for our academic achievements as well as for my personal development as a researcher. I would like to thank him also very much for assembling such a great collection of remarkable colleagues in the research group I have taken special pride to be part of as well as for providing such an inspiring atmosphere and excellent conditions to conduct research.

I would also like to thank Prof. Dr. Jörg Hähner for kindly agreeing to review my thesis and to be a part of the examination commission. Having shaped earlier parts of my academic career, it is a particular pleasure and honour to meeting him again for its provisional highlight.

A very important role for my work and for the pleasure of my time in the group is to be attributed to Dr. Frank Dürr. His distinctive passion and enthusiasm for research have been inspiring and driving me since the first moments. Not only was he an outstanding mentor and of invaluable importance for my research, he furthermore marks a remarkable person. His manifold interests, his reassuring manner and his good sense of humour make him a great conversationalist, which has led to many many joyful, mostly unscheduled hours of interesting discussions.

On the same line, I would like to express my appreciation to all the other brilliant colleagues in the group. I consider it a privilege to had been working with you and would like to thank every one of you very much for making my time in the group so productive, memorable and joyful, especially Dr. Naresh Nayak and Dr. Sukanya Bhowmik for the fruitful collaboration on topics of Software-defined Networking, Christoph Dibak for the manifold discussions and for being a perfect office mate, and Dr. Christian Mayer for his

honest interest and feedback. Furthermore, I would like to thank Eva Strähle for helping me out with administrative matters of all sorts as well as Martin Brodbeck and Franz Fabian for their technical support, especially regarding the hardware testbeds.

Finally, I would like to thank my family and my friends for creating such a supportive environment, without which this journey would not have been possible. I want to especially thank my parents Erwin and Heike who have been continuously nurturing my natural curiosity and interest in science and who have consequently enabling and constantly encouraging me to pursue my academic career. Thanks go to my brother Michael for his love and care. I would also like to thank my parents-in-law Claus and Franziska for their kind hospitality in providing me an office space and provisions for the last write-up phase of this thesis. Last but not least, I want to express my sincerest gratefulness to my wife Livia and my daughter Flora Emilia for their endless patience and the sacrifices they had to endure during this significant phase of our lives. I thank Livia for her love, care, and enduring support through prosperity and adversity of the last years. Words cannot merely express how happy I am to have you in my life.

Contents

Abstract 19

Zusammenfassung 21

1 Introduction 25
1.1 Separation of Control Plane and Data Plane & Flow-Table Programming Abstraction . . . 26
1.2 Distribution and Logical Centralization of Control in Software-defined Networking . . . 29
1.3 Consistency in Software-defined Networking . . . 31
1.4 Research Focus and Contributions . . . 33
1.5 Structure of the Thesis . . . 36

2 Software-defined Networking 37
2.1 Architecture and System Model . . . 37
2.2 OpenFlow—The de-facto Standard of SDN . . . 38
2.2.1 The OpenFlow Processing Pipeline . . . 39
2.2.2 The OpenFlow Control Channel . . . 41
2.2.3 OpenFlow-specific Mechanisms for Update Consistency . . . 42
2.3 Data Plane Consistency . . . 43
2.3.1 Update Consistency . . . 43
2.3.2 Data Plane Update Mechanisms . . . 46
2.4 Design Space and Trade-offs in Network Control Distribution . . . 48
2.4.1 Evolution of SDN Controller Architectures . . . 48
2.4.2 State Consistency and Synchronization . . . 51
2.5 White-box Networking—Anatomy of Open SDN Hardware Switches . . . 54
2.6 Data Plane Programming—The Evolution of Data Plane centric SDN . . . 56
2.6.1 The P4 Language . . . 57
2.6.2 Embracing Middleboxing and Network Function Virtualization . . . 59

3 Update Consistency and Consistent Management for SDN-based Multicast 61
3.1 Update Consistency In Network Management 63
3.2 Multicast Model 66
3.3 Problem Statement and Analysis 67
3.3.1 Problem Statement: Multicast Tree Updates 67
3.3.2 Impossibility Result 68
3.3.3 Conditions for Violation of Invariants 69
3.3.4 Central Analysis Structure: The Delta Graph 71
3.3.5 Maintaining Loop-freeness 72
3.3.6 Effective Update Order 74
3.4 Flexible Approach for Multicast Tree Updates 79
3.4.1 Processing Graph Changes 80
3.4.2 Translating Graph Changes to the Data Plane 82
3.4.3 In-network Duplicate Filtering 83
3.4.4 Hybrid Update Mechanism 84
3.5 Evaluation 85
3.5.1 Stage I: Impact Analysis of Network Dynamics 85
3.5.2 Stage II: Empirical Validation 88
3.5.3 Evaluation Conclusions 93
3.6 Related Work 93
3.7 Conclusion 94

4 Full-range Distribution of Event-based Network Control 97
4.1 System Architecture 99
4.2 The SDN Message Bus: Decoupling Controllets through Events 101
4.3 Highly Flexible Control Plane Distribution 104
4.3.1 Augmented Fully Distributed Control 104
4.3.2 Local Data Plane Event Processing 106
4.3.3 Relation to Data Plane Programming and Network Function Virtualization 111
4.3.4 Challenges of Deployment on Networking Hardware 111
4.4 Implementation 114
4.4.1 ZMF: The Zero Module Framework 114
4.4.2 ZSDN: A Distributed SDN Controller 115
4.4.3 Integration Schemes for LDPEP 117

4.5 Evaluation . . . 118
4.5.1 Raw Controller Performance . . . 118
4.5.2 Scalability of Controllet Distribution . . . 120
4.5.3 Performance on White-box Networking Switch Hardware . . . 124
4.5.4 Evaluation Conclusions . . . 130
4.6 Related Work . . . 131
4.7 Conclusion . . . 132

5 P4CEP—A Data Plane Implementation of Complex Event Processing 135
5.1 Background . . . 137
5.1.1 Data Plane Programming with P4 . . . 137
5.1.2 Complex Event Processing . . . 138
5.2 System Model . . . 140
5.3 Distribution and Consistency Aspects . . . 141
5.3.1 Requirements from the Application's Perspective . . . 141
5.3.2 Control Plane: Operator Placement and Control under Global View 142
5.3.3 Data Plane: Consistent One-to-many Event Propagation and CEP Operation . . . 142
5.3.4 Stateless In-network Filtering of Events . . . 143
5.4 P4CEP: Design & Implementation . . . 144
5.4.1 Abstract Event Processing Model . . . 144
5.4.2 The P4CEP Rule Specification Language . . . 145
5.4.3 The P4CEP Workflow and Compiler . . . 149
5.4.4 The P4CEP Processing Pipeline . . . 150
5.4.5 Data Consistency in P4CEP's Stateful Processing . . . 152
5.4.6 Limitations for Stateful Processing . . . 153
5.5 Evaluation . . . 153
5.5.1 P4CEP Hardware and Software Targets . . . 154
5.5.2 A Software-framework for CEP: Apache Flink . . . 154
5.5.3 Methodology . . . 155
5.5.4 Baseline Performance . . . 156
5.5.5 Scalability Evaluation . . . 157
5.5.6 Evaluation Conclusions . . . 162
5.6 Related Work . . . 163
5.7 Conclusion . . . 164

6 Conclusion and Future Work 167
6.1 Conclusion 167
6.2 Future Work 170

Bibliography 173

List of Figures

1.1 Generic SDN System Architecture 27
1.2 Problem space of consistency in SDN with respect to distribution 32

2.1 Differences in programming models of switches with fixed/flexible pipelines 39
(a) OpenFlow; fixed parser and pipeline; fixed-function ASIC 39
(b) P4; programmable parser and pipeline; reconfigurable ASIC 39
2.2 Fixed processing pipeline for bridging and routing on the data plane abstraction of an OpenFlow switch (OF-DPA) 40
2.3 The SDN network control loop 44
2.4 An exemplary routing update leading to intermediate states and temporal violation of network invariants 45
2.5 Evolution of distribution in SDN controller architectures 49
(a) non-modular, monolithic 49
(b) modular, monolithic 49
(c) modular, distributed, partitioned 49
(d) full-range distributed control 49
2.6 Comparison of software stacks of conventional and white-box hardware switches 55
2.7 Heterogeneity and typical components of White-box switch's hardware and software stacks 55

3.1 Overview of the proposed SDN-based system architecture for update consistency aware multicast networks 65
3.2 Branch update within a multicast distribution tree 67
3.3 Traces of packets during intra-update states leading to inconsistency effects 70
(a) packet duplication 70
(b) packet drop 70
3.4 Intermediate step of G^{Δ}-construction 73
3.5 Traversal of a delta graph, leading to a correct ordering among the updates 73

3.6 A simplified example for the analysis of intermediate states during replicator move updates . . . 76
3.7 Intermediate states of replicator moves as perceived by traversing packets considering propagation delay . . . 77
(a) Upstream replicator move with $u^{+}_{r'} \lhd u^{-}_{r}$ . . . 77
(b) Upstream replicator move with $u^{-}_{r} \lhd u^{+}_{r'}$ . . . 77
(c) Downstream replicator move with $u^{-}_{r} \lhd u^{+}_{r'}$ . . . 77
3.8 Overview of the proposed multicast update procedure . . . 80
3.9 In-network duplicate filtering . . . 84
(a) Total . . . 84
(b) Partial . . . 84
3.10 Scenario Generator for distribution tree changes . . . 86
3.11 DFN X-WiN base topology with exemplary multicast distribution tree . . . 87
3.12 Analytic evaluation (Stage I): number of replicator moves and affected nodes for varying degree of dynamics . . . 89
3.13 Distribution of replicator update types . . . 90
3.14 Empirical data plane effect occurrence (Stage II): drop and duplication factors for varying strategy and degree of update message reordering . . . 92

4.1 System architecture of ZeroSDN's full-range control distribution . . . 100
4.2 Content-based routing of data plane and control plane events over the message bus . . . 102
4.3 Design space and trade-offs in network control distribution . . . 105
4.4 Schematic overview of the Autonomous Forwarding controllet, locally processing a data plane event . . . 108
4.5 Requirements for safe control plane operation with multiple LDPEP controllets ensuring data integrity and liveness . . . 113
(a) Isolation of switch-local control processes . . . 113
(b) Fine-grained control resource control (RAM and CPU) . . . 113
4.6 Comparison of traditional heavyweight and lightweight virtualization techniques . . . 114
(a) Traditional virtual machines . . . 114
(b) Unikernels . . . 114
(c) Linux Containers . . . 114
4.7 Dependency graph for essential controllets . . . 115

4.8 Evaluation setup of control plane for raw controller performance and scalability evaluation . . . 119
4.9 Raw controller performance: comparison of throughput and sequential throughput . . . 121
4.10 Scalability evaluation of modular ZSDN . . . 123
4.11 Setup for measuring the control plane processing latency on a white-box switch . . . 124
4.12 Comparison of processing latencies on a white-box hardware switch . . . 126
4.13 CDFs of switch processing latencies on a white-box hardware switch. . . . 126
4.14 Control plane packet processing latencies of varying NOSes and isolation mechanisms . . . 128
4.15 CDFs of control plane packet processing latencies of varying NOSes and isolation mechanisms . . . 129

5.1 Comparison of *ex-situ* and *in-situ* processing in CEP . . . 136
5.2 Illustration of a Sliding-window within Complex Event Processing . . . 140
5.3 The P4CEP system model . . . 141
5.4 Generated finite state machine detecting complex event patterns . . . 146
5.5 P4CEP workflow: design-time components and source files involved in building P4CEP for different targets . . . 149
5.6 Pipeline for processing of packets and events within a P4CEP-target . . . 150
5.7 Evaluation setup for measuring latency and throughput of the P4CEP targets and Apache Flink . . . 156
5.8 Performance of P4CEP for increasing window sizes on P4CEP targets and comparison with Apache Flink . . . 159
(a) End-to-end event detection latency latency l_p . . . 159
(b) Relative operator throughput B_p . . . 159

List of Tables

3.1 Definition of switches roles in a multicast distribution tree 66
3.2 Port change-set representation of a branch update 69
3.3 Table of relevant notations for Section 3.4 79

5.1 Representation of the P4CEP FSM as a P4 match-action table. 146
5.2 Summary of P4CEP scalability results on the NFP target 157

List of Listings

4.1 Excerpt of the SwitchAdapter topic-hierarchy 117

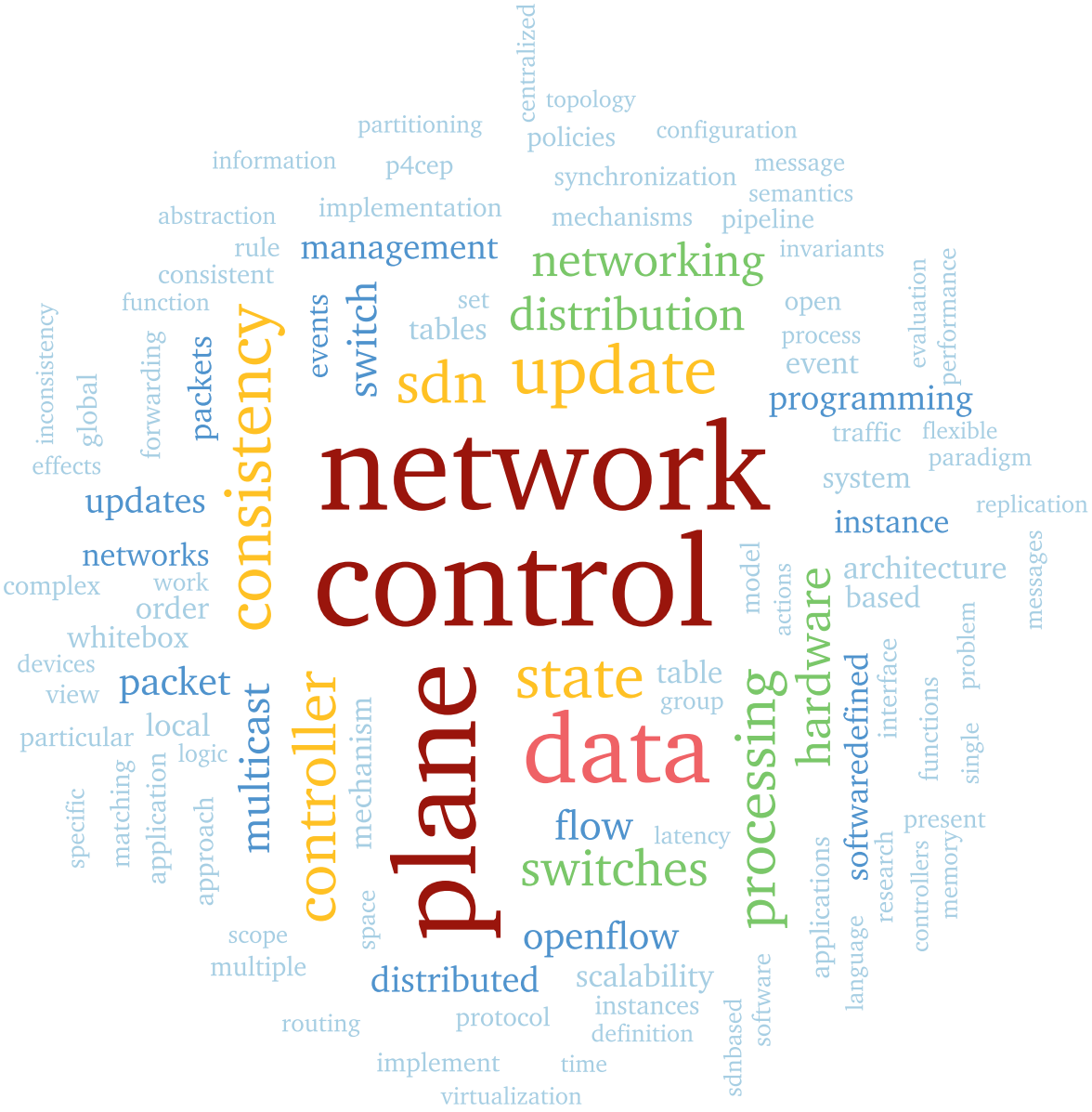
5.1 Exemplary P4CEP-rule definition of a window and a sequential pattern . . 146
5.2 P4CEP rule definition for the window scalability evaluation 158
5.3 Flink equivalent to the P4CEP rule definition for the scalability evaluation 160
5.4 P4CEP rule definition for the pattern complexity scalability evaluation . . 160
5.5 P4CEP rule definition for the predicate complexity scalability evaluation . 161
5.6 P4CEP rule definition for the pattern variables scalability evaluation . . . 161
5.7 P4CEP rule definition for the complex events scalability evaluation 162
5.8 P4CEP rule definition for the engine instance scalability evaluation 162

List of Abbreviations

10GbE	Ten-Gigabit Ethernet
ACL	Acess Control List
AFC	Autonomous Forwarding Controllet
ADD_F	Drop-prevention update strategy (add first)
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-specific Integrated Circuit
bmv2	Behavioral model version 2 (reference P4 software switch)
CAM	Content-addressable Memory
CDF	Cumulative Distribution Function
CEP	Complex Event Processing
CLI	Command Line Interface
CM_i	External Controllet
CP	Control Plane
CPE	Control Plane Event
DAG	Directed Acyclic Graph
DCN	Data Center Network
DFS	Depth-first search
DHT	Distributed Hash Table
DP	Data Plane
DPE	Data Plane Event
DTLS	Datagram Transport Layer Security
eBPF	Extended Berkeley Packet Filter
ECMP	Equal-cost Multi-path Routing
EMU	External Memory Unit
FIB	Forwarding Information Base
FPC	Flow Processing Cores

FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FT	Flow Table
GbE	Gigabit Ethernet
G^{Δ}	Delta Graph
HLP	High Level Policy
IGMP	Internet Group Management Protocol
IMU	Internal Memory Unit
IPC	Inter-process Communication
KVM	Kernel-based Virtual Machine
LAN	Local Area Network
LDPEP	Local Data Plane Event Processing
L_i	Switch-local Controllet
LLDP	Link-Layer Discovery Protocol
LPM	Longest-prefix Matching
LXC	Linux Containers
NF	Network Function
NFP	Netronome Network Flow Processor
NFV	Network Function Virtualization
NIB	Network Information Base
NIC	Network Interface Controller
NM	Network Manager
NOS	Network Operating System
NPU	Network Processing Unit
NREN	National Research and Education Network
OCP	Open Compute Project
OF	OpenFlow
OF-DPA	OpenFlow Data Plane Abstraction
ONIE	Open Network Install Environment
ONL	Open Network Linux
OS	Operating System
OSGi	Open Services Gateway initiative
OVS	Open vSwitch
P4	Programming Protocol-independent Packet Processors
$P4_{14}$, $P4_{16}$	P4 language version 14, $\sim$ version 16

pps	Packets per second
Pub/Sub	Publish/Subscribe
QoE	Quality of Experience
QoS	Quality of Service
REM_F	Duplicate-prevention update strategy (remove first)
RISC	Reduced Instruction Set Computer
RTT	Round-trip Time
SA	Switch Adapter Controllet
SAI	Switch Abstraction Interface
SDN	Software-defined Networking
SF	Simple Forwarding Controllet
S_i, FE	Switch / Forwarding Element
SM_i	State Module
SMR	State Machine Replication
SR-IOV	Single Root I/O Virtualization
TCAM	Ternary Content Addressable Memory
TLS	Transport Layer Security
µK	Micro Kernel
VM	Virtual Machine
vNF	Virtualized Network Function
VS	Virtual Synchrony
VT-x	Intel Virtualization Technology x86
WAN	Wide-area Network
ZMF	Zero Module Framework
ZMQ	ZeroMQ
ZSDN	ZeroSDN controller framework
ZSDN-AFC	ZeroSDN employing the Autonomous Forwarding Controllet
ZSDN-IPC	ZeroSDN based on Unix Domain Sockets
ZSDN-TCP	ZeroSDN based on TCP


network
control
plane
data
consistency
update
sdn
networking
distribution
management
programming
processing
hardware
state
controller
multicast
packet
packets
switch
events
switches
flow
openflow
distributed
scalability
routing
protocol
implement
time
virtualization
instances
definition
sdnbased
software
applications
language
research
controllers
memory
present
softwaredefined
interface
functions
problem
single
architecture
based
instance
replication
messages
model
actions
traffic
flexible
paradigm
system
evaluation
performance
open
process
event
invariants
pipeline
mechanisms
synchronization
message
semantics
configuration
policies
topology
centralized
partitioning
information
p4cep
implementation
abstraction
rule
consistent
function
set
tables
inconsistency
global
forwarding
effects
updates
networks
complex
work
order
whitebox
devices
view
particular
local
logic
specific
matching
application
approach
scope
multiple
space
mechanism
latency
table
group
table
scope

ABSTRACT

Software-defined Networking (SDN) is an emerging networking paradigm promising flexible programmability and simplified management. Over the last years, SDN has built up huge momentum in academia that has led to huge practical impact through the large-scale adoption of big industrial players like Google, Facebook, and Microsoft driving cloud computing, data center networks, and their interconnection in SDN-based wide-area networks. SDN is a key enabler for high dynamics in terms of network reconfiguration and innovation, allowing the deployment of new network protocols and substantially expanding the networking paradigm by moving applications into the network, both at unprecedented pace and ease. The SDN paradigm is centered around the separation of the data plane from the logically centralized but typically physically distributed control plane that programs the forwarding behaviour of the network devices in the data plane based on a global view. Especially requirements on correctness, scalability, availability, and resiliency raised through practical adoption at scale have put a strong emphasis on consistency and distribution in the SDN paradigm.

This thesis addresses various challenges regarding consistency and distribution in Software-defined Networking. More specifically, it focusses and contributes to the research areas of update consistency, flexibility in control plane distribution, and data plane implementation of a distributed application. Reconfiguring an SDN-based network inevitably requires to update the rules that determine the forwarding behaviour of the devices in its data plane. Updating these rules, which are situated on the inherently distributed data plane devices, is an asynchronous process. Hence, packets traversing the network may be processed according to a mixture of new and old rules during the update process. Consequently arising inconsistency effects can severely degrade the network performance and can break stipulated network invariants for instance on connectivity or security. We introduce a general architecture for network management under awareness of expectable update-induced inconsistency effects, which allows for an appropriate selection of an update mechanism and its parameters in order to prevent those effects. We thoroughly analyze update consistency for the case of multicast networks, show crucial particular-

ities and present mechanisms for the prevention and mitigation of multicast-specific inconsistency effects.

Observing that on the one hand SDN's separation of control has been deemed rather strict, moving any control "intelligence" from the data plane devices to remote controller entities hence increasing control latency while on the other hand the coupling between controller and data plane devices is quite tight hence hindering free distribution of control logic, we present a controller architecture enabling flexible and full-range distribution of network control. The architecture is based on decoupling through an event abstraction and a flexible dissemination scheme for those events based on the content-based publish/subscribe paradigm. This lightweight design allows to push down control logic back onto data plane devices. Thus, we expand SDN's control paradigm and enable the full range from fully decentralized control, over local control still profiting from global view up to fully centralized control. This scheme allows to trade-off scope of state data, consistency semantics and synchronization overhead, control latency, and quality of control decisions. Furthermore, our implementation covers a large set of mechanisms for improving control plane consistency and scalability, such as inherent load-balancing, fast autonomous control decision making, detection of policy conflicts, and a feedback mechanism for data plane updates.

In a last area, we focus on the implementation of a distributed application from the domain of message-oriented middleware in the data plane. We implement Complex Event Processing (CEP) on top of programmable network devices employing data plane programming, a recent big trend in SDN, or more specifically, using the P4 language. We discuss challenges entailed in the distributed data plane processing and address aspects of distribution and consistency in particular regarding consistency in stateful data plane programming, where internal state that determines how packets are processed is changed within this very processing, in turn changing the processing of subsequent packets. Since packet processing is executed in parallel on different execution units on the same device sharing the same state data, strong consistency semantics are required in order to ensure application correctness. Enabled by P4's flexible and powerful programming model, our data plane implementation of CEP yields greatly reduced latency and increased throughput. It comprises a compiler that compiles patterns for the detection of complex events specified in our rule specification language to P4 programs, consisting of a state machine and operators that process so-called windows containing historic events.

Zusammenfassung

Softwaredefinierte Netzwerke (SDN) verkörpern ein neues Netzwerkparadigma mit dem Ziel einer flexibleren Netzwerkprogrammierung und eines vereinfachten Netzwerkmanagements. Dabei hat SDN über die letzten Jahre sehr stark an Bedeutung gewonnen, sowohl in der Forschung als auch in der Industrie, wo bedeutende Firmen wie Google, Facebook und Microsoft SDN verbreitet zum Betrieb ihrer Cloudumgebungen, ihrer Rechenzentrumsnetzwerke als auch für die die Rechenzentren verbindenden Weitverkehrsnetzwerke einsetzen. SDN ist eine Schlüsselkomponente sowohl für den Umgang mit hoher Netzdynamik im Sinne von Netzrekonfiguration als auch für eine bislang unerreicht schnelle und unkomplizierte Umsetzung von Innovationen im Netzwerkbereich etwa durch neue Protokolle oder durch das Erweitern des gesamten Netzwerkparadigmas durch das Abbilden von Anwendungen im Netzwerk selbst. Das SDN-Paradigma basiert auf einer Trennung der Datenebene von der logischen zentralisierten, typischerweise aber physisch verteilten Kontrollebene, die das Weiterleitungsverhalten der Netzkomponenten in der Datenebene programmiert, wobei sie auf eine globale Netzsicht zurückgreifen kann. Insbesondere Anforderungen an Korrektheit, Skalierbarkeit, Verfügbarkeit und Resilienz, die sich durch den Einsatz in der Praxis im großen Maßstab ergeben, messen Konsistenz und Verteilung im SDN-Paradigma eine hohe Bedeutung bei.

Die vorliegende Arbeit behandelt verschiedene Herausforderungen bezüglich Konsistenz und Verteilung in softwaredefinierten Netzwerken und bezieht sich dabei insbesondere auf die Forschungsbereiche Updatekonsistenz, Flexibilität bei der Verteilung der Kontrollebene, sowie der Implementierung einer verteilten Anwendung in der Datenebene und den damit einhergehenden Konsistenzgesichtspunkten. Die Rekonfiguration eines SDN-basierten Netzwerks erfordert unausweichlich eine Aktualisierung der Regeln die das Weiterleitungsverhalten der Netzkomponenten in der Datenebene steuern. Das Aktualisieren dieser sich auf den inhärent verteilten Datenebenenkomponenten befindlichen Weiterleitungsregeln stellt einen asynchronen Prozess dar. Dies kann dazu führen, dass Pakete in der Datenebene nach einer Mischung aus alten und neuen Regeln verarbeitet werden. Die dadurch auftretenden Inkonsistenzeffekte können die Netzwerkperformanz

erheblich einschränken und verletzen oft festgelegte Netzeigenschaften, etwa damit festgelegte Verbindungs- oder Sicherheitseigenschaften. In der vorliegenden Arbeit wird daher eine generische Architektur für Netzwerkmanagement eingeführt die sich dadurch auszeichnet Kenntnis über zu erwartende Inkonsistenzeffekte zu haben und diese dadurch in Kontrollentscheidungen miteinbeziehen zu können. Somit wird die Auswahl eines angemessenen Updatemechanismus mitsamt zugehöriger Parameter ermöglicht, um das Auftreten dieser Inkonsistenzeffekte zu verhindern. Des Weiteren wird eine vollständige Analyse von Updatekonsistenz für Multicastnetzwerke durchgeführt. Dabei werden signifikante multicastspezifische Eigenheiten aufgezeigt, sowie Mechanismen zur Verhinderung bzw. Abmilderung von multicastspezifischen Inkonsistenzeffekten präsentiert.

Motiviert durch die Beobachtung, dass die Trennung von Kontrolle in SDN im Allgemeinen sehr strikt umgesetzt wird, wodurch sämtliche Kontrollfunktionalität von der Datenebene zu entfernten Einheiten der Kontrollebene verschoben wird, was die Kontrolllatenz erhöht, und andererseits eine enge Kopplung zwischen Kontrolleinheiten und Datenebenenkomponenten besteht, was die freie Verteilung von Kontrolllogik einschränkt, wird in der Arbeit eine Kontrollarchitektur vorgestellt, die eine flexible und vollumfängliche Verteilung von Kontrolllogik ermöglicht. Diese Architektur basiert auf einer durch eine Eventabstraktion erreichten Entkopplung in Kombination mit einem flexiblen Eventverbreitungsschema basierend auf dem inhaltsbasiertem Publish/Subscribe-Paradigma. Das leichtgewichtige Design ermöglicht die Rückverlagerung von Kontrolllogik auf die Datenebenenkomponenten, was das Kontrollparadigma von SDN erheblich ausweitet und die ganze Spanne von vollständig dezentralisierter Kontrolle über lokale Kontrolle unter Miteinbezug der globalen Netzsicht bis hin zur vollständig zentralisierten Kontrolle, auf die das SDN-Paradigma ursprünglich beschränkt war, ermöglicht. Dieses Kontrollschema ermöglicht eine Abwägung zwischen Umfang der Zustandsdaten (globale Sicht bis ausschließlich lokale Sicht), Konsistenzsemantiken und Synchronisationsaufwand, sowie Kontrolllatenz und Qualität der Kontrollentscheidungen. Des Weiteren deckt der vorgestellte Ansatz einen breiten Bereich an Mechanismen zur Erhöhung der Kontrollebenenkonsistenz und der Skalierbarkeit ab, etwa durch inhärente Lastverteilung, schnellen autonomen Kontrollentscheidungen, Erkennung von Konflikten in Netzrichtlinien, sowie durch einen Feedbackmechanismus für Regelaktualisierungen auf der Datenebene.

Ein letzter Bereich der vorliegenden Arbeit behandelt die Implementierung einer verteilten Anwendung aus dem Bereich der kommunikationsorientierten Diensteschicht („Middleware“) in der Datenebene. Darin wird eine komplexe Ereignisverarbeitung, oder „Complex Event Processing“ (CEP), auf programmierbaren Netzgeräten unter Verwendung

von Datenebenenprogrammierung, einem neuen und mächtigen Schritt in der Evolution des SDN-Paradigmas, mit der spezifischen Programmiersprache P4 umgesetzt. In diesem Zuge werden die mit einer verteilten Verarbeitung in der Datenebene einhergehenden Herausforderungen herausgearbeitet, insbesondere bezüglich Verteilung und Konsistenz in zustandsbasierter Datenebenenprogrammierung, in welcher interner Zustand, der die Verarbeitung von Paketen bestimmt, durch eben jene Verarbeitung verändert wird, was wiederum die Verarbeitung von nachfolgenden Paketen beeinflusst. Da Paketverarbeitung typischerweise auf hochparallel ausgelegten Verarbeitungseinheiten auf dem selben Gerät unter Nutzung des selben Zustandsspeichers ausgeführt wird, sind strikte Konsistenzsemantiken zur Sicherstellung der Korrektheit der Anwendung unerlässlich. Durch die Paketverarbeitung auf Hardware, welche durch das flexible und mächtige Programmiermodell von P4 ermöglicht wird, konnte der Durchsatz und die Latenz der vorgestellten Datenebenenimplementierung von CEP stark erhöht, bzw. verringert werden. Die Implementierung beinhaltet einen Compiler, der Regeln die in einer eigenen Regelspezifikationssprache verfasst sind in ein P4-Programm übersetzt. Das P4-Programm wiederum besteht aus einem Zustandsautomaten zur Eventerkennung und einer Verarbeitungseinheit für in einem sogenannten Fenster gespeicherte historische Ereignisse.

1 Introduction

The modern Internet, enabling a worldwide exchange of information in the form of packets, and many other kinds of communication networks including data center, enterprise, and domestic networks are vastly dependent on distributed protocols for network control and packet transport. Traditional IP networks along with their hardware devices such as routers and switches, on which those distributed protocols run, constitute the backbone of many networks and are hence vastly adopted, although it is commonly agreed that they are complex and hard to manage [BAM09, KREV+15]. To implement the desired behaviour of the network (*network policies*), network operators need to configure devices typically through low-level and proprietary, i.e., vendor-specific, management interfaces using closed APIs. Given the scale and complexity of modern networks and the demand to maintain optimal operation in the presence of high dynamics regarding failures of devices and shifts in load patterns, their configuration poses a particular challenge. The segmentation of network control due to vendor-proprietary management has effectively hindered holistic and standardized mechanisms for automated adaptation and reconfiguration in current IP networks. Furthermore, current networking devices tightly couple in their firmware the control plane (CP), whose responsibility it is to decide how traffic in the network is to be handled in order to implement desired network policies, and the data plane (DP), also called forwarding plane, which forwards traffic according to the control decisions taken in the control plane. These properties of current IP networks, proprietary management interfaces and bundling of control plane and data plane, have led to very high development and deployment cycles in the order of years or even decades and associated high cost, effectively hindering innovation in network protocols and architectures, as can for instance be seen in the dragging adoption of IPv6 [NGN16].

Software-defined Networking (SDN), is a recent paradigm and architecture to tremendously increase flexibility in networking and foster evolution and innovation in particular of network protocols. To counter the aforementioned deficiencies of traditional network architectures, the SDN paradigm is based on the two concepts described in the following sections: (1) a clear separation of the control plane, exerting network control,

from the data plane, implementing forwarding and processing functionality, and (2) the centralization of control logic on a logically centralized entity, called *controller*.

SDN has been receiving substantial attention from both, academia and industry. Large internet companies like Google and Facebook have been adopting the SDN paradigm both within data center networks (DCNs) [Kol14, SOA+15] and their interconnections in global wide area networks (WANs) like Google's B4, interconnecting 12 data centers [JKM+13]—back in 2013. In a scenario of tremendously growing traffic, Google was able to increase the capacity of a single data center by a factor of 100. Due to its unprecedented flexibility, SDN is also a key technology for operating huge public clouds based on network virtualization, for instance, Microsoft's Azure cloud [Rus15, Gre15] or the Google Cloud Platform [WV18]. Moreover, a large internet service provider has set the goal of controlling 75% of its network with software by 2020 [Don].

The great support of big industrial players like Google, Facebook, Microsoft, large device manufacturers, and telco providers has led to the foundation of large industry consortia. Most important are the Open Networking Foundation (ONF) [Thec], promoting the adoption of SDN through open standards, most notably the OpenFlow protocol [Opeb] and the ONOS SDN controller [Thea], and the Open Compute Project (OCP) [Opea], driving the development of open SDN-enabled hardware by contributing open-hardware specifications of so-called SDN white-box switches and other networking devices along with unified software components, including the ONIE bootloader.

1.1 Separation of Control Plane and Data Plane & Flow-Table Programming Abstraction

SDN originates [FRZ14] from approaches for designing and deploying programmable data plane devices, such as partially proposed in Active Networks [TSS+97] back in the 1990s, the "Forwarding and Control Element Separation" (ForCES) [YDAG04, DSH+10], published in 2004 by the Internet Engineering Task Force (IETF), as well as the more recent OpenFlow [MAB+08] proposed in 2008 and "Protocol oblivious Forwarding" (PoF) [Son13] from 2013. Their common scheme subsumed in SDN is a flow-table abstraction for programming of network behaviour (cf. Figure 2.1, p. 39), i.e., forwarding and processing of packets, to implement desired policies. To this end, traffic of interest is classified by specific combinations of packet headers of any layers and their values, defining *flows* through the network (opposed to traditional forwarding and routing, which would

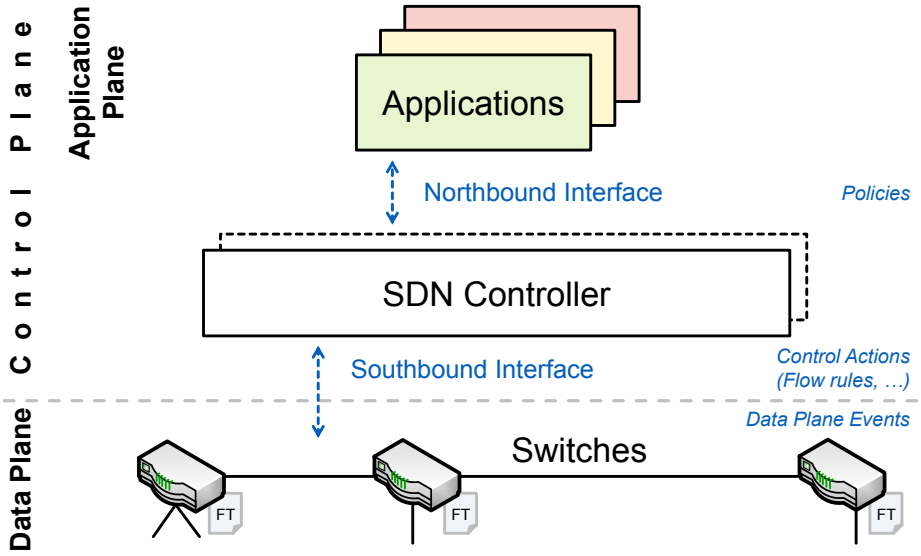


Figure 1.1. The generic SDN System Architecture

typically consider solely packet destination information). Data plane devices, henceforth called switches, implement *flow tables* (FT). Entries in this table, called *flow rules*, match on a subset of the traffic that is specified by the flow, and associate a set of actions that is applied to all incoming packets belonging to that flow. Common actions include dropping or forwarding a packet (to another switch or to the controller) and appending, removing, or modifying of packet headers, for instance adding tags of tunneling mechanisms such as VLAN, or decrementing the time to live (TTL) field in the IP header. The latest evolution of SDN with *Data Plane Programming* enables much more powerful matching and action flexibility by granting access to advanced capabilities on a switch's packet processing hardware. Capable devices have reconfigurable processing pipelines and inter-packet state memory allowing for powerful user-defined packet processing, as well as a flexible parser and de-parser, allowing for the definition of custom packet headers and hence new, user-defined protocols, breaking protocol-dependency—at full line rate processing throughput. This abstraction along with the so-called *match-action semantic* is overall quite powerful: depending on the installed rules, switches can implement simple network functions (NFs) like forwarding or routing, but also perform more complex NFs like firewalling, load-balancing, or traffic-shaping up to yet more complex middlebox functionality, like content-based publish/subscribe or Complex Event Processing.

Figure 1.1 shows a simplified view of a typical SDN architecture. The decoupling of a switch's data plane from the control plane is implemented by a *southbound interface* that

allows the dynamic configuration by remote entities, i.e., the controller, through simple operations (control actions) of adding, removing, and updating of flow rules, i.e., entries in a switch's flow table. A southbound protocol, such as the most prominent OpenFlow Protocol [MAB+08, Opeb], specifies a well-defined API to grant the controller direct access to the switch's flow table and to auxiliary functions, such as monitoring, for instance, gathering network statistics. Furthermore, it provides a back-channel for switch-solicited notifications of events in the data plane, such as incoming packets for which the flow table holds no matching rule, switchport and link failures, etc. The controller thus exercises direct control over the switch and determines the network function(s) the switch is to implement. The controller is furthermore able to detect and (automatically) react to events in the data plane, in order to maintain (optimal) network operation.

The *northbound interface* provides an API to developers of applications on top of a controller framework (*application plane*). It allows developers to use common functionality of a controller framework, such as topology discovery, switch inventory and management, etc. More specifically, control applications are able to consistently use the global network view that is gathered by the controller framework. Moreover, depending on the actual framework, it may offer an abstraction for *high-level network programming*—a big domain in SDN research [MRF+13a, AFG+14, KRG+15]—hiding low-level southbound programming interfaces, i.e., direct resource access to switch flow tables, in favour of allowing for modular and declarative network programming using network programming languages like Frenetic [FGR+13] or Pyretic [RMF+13]. Some SDN literature further differentiates between the application plane and a *management plane* (undepicted) as an interface between the control plane and (human) network operators defining network policies and orchestrate functionality offered by the application plane.

The major benefit of a decoupled control plane with offloaded control plane functions and a standardized and open southbound protocol is that through the separation of concerns, the controller (and the actual applications) can be implemented in software and use non-proprietary interfaces. Compared to traditional networking where control plane functionality was tightly integrated in the switch devices in form of vendor- and device-specific firmware implementations using closed APIs, SDN tremendously facilitates changes and extensions to the control plane and thus ultimately to network operation with drastically decreased innovation cycles. The protocol independence and additional flexibility in terms of packet processing introduced by data plane programming further drastically speeds up development, however relying on special, not yet widely adopted hardware.

1.2 Distribution and Logical Centralization of Control in Software-defined Networking

SDN is based on the paradigm of *logically centralized control* over distributed network elements. Logical centralization is nothing more than the concept of *distribution transparency*, which is well-known from Distributed Systems. Distribution transparency hides the complexity of a physically distributed system from the application by making distribution aspects "transparent", i.e., *not* visible to the application. Thus, the client can be implemented as if the system were centralized. In particular, network control applications implementing network control logic have a global view of the network. In traditional networking, fully decentralized control algorithms had to construct and maintain a (partial) view of the network entirely locally before being able to make local control decisions based on this view. In SDN, the problem of constructing a view of a much larger scope, i.e., an entire global view, is dedicated to the centralized SDN controller, which acquires this global view by gathering network information such as topology information through monitoring of inherently distributed network elements (the switches). This allows control applications to improve the quality of control decisions, consistently taken on the provided global view. Moreover, the SDN controller itself might be (ideally) a distributed system with all its defining properties like replication transparency, fragmentation transparency, and without a single point of failure. For instance, topology information stored in a *network information base* (NIB) might be replicated to and partitioned between many controller instances.

While the data plane is inherently distributed due to its physically distributed network elements, production-grade SDN frameworks also tend to physically distribute the control plane, i.e., the controller, in order to ensure adequate levels of availability and scalability. More specifically, these requirements demand for eliminating a single point of failure by providing sufficient redundancy of controller functionality (availability) and providing sufficient controller performance for an increasing scale of networks to control (scalability). In order to meet these requirements, two distribution models are typically be applied: (1) replication and (2) partitioning.

The rationale of (1) replication, is to distribute application logic implemented in a single entity to a set of identical entities (replicas) mainly to improve reliability, fault-tolerance, and availability. Note that replication may also increase scalability since the workload can then be distributed to a set of replicas instead of a single entity handling the entire load (see partitioning). In the case of SDN, the controller is thus replicated to a set of *controller*

instances that are identical in terms of control logic. The controller instances are assumed to be *deterministic*, which means that their behavior solely depends on its *state*, i.e., data completely describing an instance's current state, such as the network view (the NIB), and control state (control decisions). In other words, the same *input* (data plane event) leads to the same output (control decision) on any instance, iff all controller instances operate on the same state. Furthermore, state is assumed to only be changed by input, in a way specified by the (common and immutable) control logic. Thus, for correct and consistent network control, the state data has to be synchronized among all instances. In general, this can be achieved by a mechanism known as State Machine Replication (SMR) [Sch90], where distributed consensus mechanisms such as Paxos [Lam98] basically ensure that the set of replicas agree on the ordering of inputs and hence each replica processes the same (sequence) of inputs. Another means to achieve controller state synchronicity is to externalize the state data in a logically centralized data store, where a transactional replication mechanism ensures consistent data access from controller instances under ACID properties (atomicity, consistency, isolation, and durability) as known from database systems [HR83]. It should be noted, that keeping the state data synchronized introduces cost in terms of message complexity (increased control plane traffic) and time complexity (increased control latency), depending on the desired level of consistency.

In order to improve scalability, (2) partitioning aims to structurally break an overall problem into tractable subproblems by a formation of disjoint partitions along one or multiple problem dimensions, which can be processed largely independently and hence in parallel to the other partitions. The improvement in scalability stems from distributing the overall load to multiple instances responsible for dedicated partitions. In the case of an SDN controller, state synchronization efforts can be minimized by partitioning along network topology. While being functionally equivalent, instances can be mainly bijectively associated to partitions of the network topology, thus reducing the scope of state data that has to be shared with other instances, consequently reducing synchronization overhead. Partitioning the topology along its layers, for instance in a leave-spine data center network topology, is known as *hierarchical distribution* in SDN literature [LHG+15]. Note, that partitioning along the topology cannot fully eliminate dependency to other partitions, without creating actual network partitions, i.e., network regions that can not communicate with each other. Another partitioning scheme is to partition along network functions. Here, controller instances are not functionally equivalent, but only contain logic for the NF to implement. However, the NF-partitioned instances operate on overlapping or even congruent topology regions, requiring state synchronization of larger scope.

1.3 Consistency in Software-defined Networking

Overall, the ideal of the SDN paradigm is to achieve a consistent behaviour of network control defined by high-level network policies at any point in time, whose control decisions are directly deployed to the data plane, while the data plane behaves exactly as intended by the control plane at any point in time. Moreover, the control plane ideally has an accurate global view that is consistent with the actual physical data plane at any point in time. Deviations from this ideal such as caused by (temporal) inconsistencies in general result in a varying degree of negative implications on network management, i.e., the violation of policies, for instance ranging from slight and temporary network performance degradation or slight deviations of policies up to fatal and long-lasting cease of network operation or total policy violation. Figure 1.2 illustrates the problem space of consistency in SDN, which can be broken down into the following tractable subproblems.

First, consistent behaviour of a physically distributed control plane mandates consistency of control state data among controller instances (**state data consistency**). As described earlier, the problem of state synchronization can be solved by applying mechanisms like state machine replication or be offloaded to externalized datastores. A much-debated question in SDN literature is to define an adequate consistency semantics and whether state data can be differentiated by "importance", requiring stronger consistency semantics for important data and less strict semantics for less important data.

Second, a controller has to detect and reconcile inconsistencies caused by conflicting controller applications or conflicting policies (**policy conflict**)—a problem recognized as *policy composition* in literature [CKLS13]. A simple and obvious example is an application installing communication flows between network elements might easily conflict with a firewall application that is deliberately preventing specific network communication. Canini *et al.* [CKLS13] provide a more intricate example of conflicting policies regarding monitoring and waypoint enforcement. On the same line of ensuring coordination among policies, there is also a need for **coordination** among controller instances, in the case of controller partitioning, where a consistent mapping of instances to topology partitions has to be maintained. The problem is exacerbated by non-disjoint, i.e., overlapping, topology regions with possibly resulting multiple controller associations and hence multiple responsibilities or a partitioning along network functions, basically resembling the policy composition problem.

Third, a prerequisite of guaranteeing consistent decisions of network control applications is to ensure a consistent global view among those applications (see above). However,

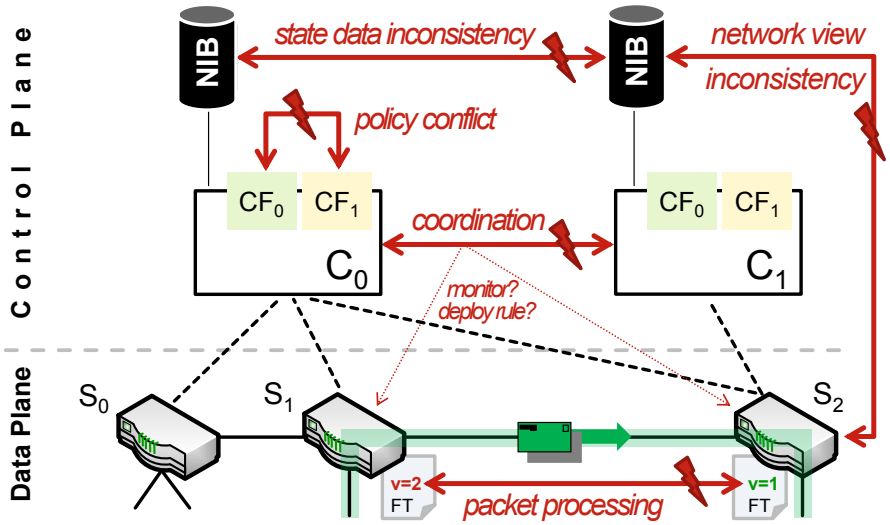


Figure 1.2. Problem space of consistency in Software-defined Networking with respect to the deliberate distribution of the control plane and the inherently distributed data plane.

the global view acquired and maintained by the controller might not represent and thus be inconsistent with the actual ground truth, i.e., the physical data plane state of the network [PZH+17].

Fourth, traffic in the data plane is processed (**packet processing**) according to state held in the switches' flow tables (flow-table programming abstraction). Today's softwarized networks are in constant flux, continuously adapting to changes in the network topology, load patterns, network functions, and overall policies. To implement adaptations, the network has to be reconfigured inevitably. During the process of reconfiguration (network update), the flow tables (FT) of one set of switches might already have been updated, while another set still operates on not-yet updated flow tables. Thus, network traffic might be handled according to different, inconsistent versions of flow table configuration, which in turn might temporarily violate arbitrary network policies—a problem known as *update consistency* [RFRW11, FSV19].

Fifth, in modern data plane programming, packet processing is not solely determined by table entries, but instead considers local inter-packet state. This kind of state is in contrast to table state not changed by the control plane, but through the processing of packets in the data plane, effectively extending the network control loop to the data plane (*stateful data plane processing*). Thus, consistency of local inter-packet state among reconfigurable data plane devices plays a crucial role for the correctness of distributed applications implemented in the data plane.

1.4 Research Focus and Contributions

In this thesis, we address various aspects of the presented research areas of distribution and the problem space of consistency in Software-defined Networking, focussing on three main areas. The thesis is based on work that has been presented to the scientific community in the form of peer-reviewed publications at international conferences [KDR15, KDR17, KDBR17, KMD+18] and within international journals [KDR16, KDR18]. It combines and extends these works and their particular contributions on distribution and consistency in SDN. The contributions of this thesis were endorsed by continuous advising of Prof. Dr. Kurt Rothermel and Dr. Frank Dürr. Furthermore, parts of the implementations and evaluations involve contributions from student's works [Str15, Fet16, Bäu16, Maa18]. More specifically, the main research areas and contributions of this thesis are as follows.

1. Update Consistency and SDN-based Multicast Networks The first part of this thesis focuses on update consistency and SDN-based management of multicast networks. A conducted literature research on update consistency leads us to argue, that in general, updating the network is an intricate and crucial process with a possibly large parameter space. Thus, network management architectures should incorporate awareness of inconsistency effects due to network updates. We present such a management architecture allowing for an appropriate selection of an update mechanism and its parameters based on expected inconsistency effects. We find that multicast benefits much from SDN-based management and identify crucial particularities of update consistency in particular of multicast networks, which have been left largely unconsidered in literature. A thorough investigation of update consistency for the case of multicast routing leads us to the definition of the novel duplicate-freeness correctness property. We show in an extensive analysis why it is impossible to avoid violation of the two invariants drop- and duplicate-freeness for arbitrary multicast network updates using a stateless update approach. We present an update procedure for multicast routing updates that identifies critical update steps, which are fed back into the management's reconfiguration process, along with a lightweight approach that allows for the selection of an update strategy, preventing either drops or duplicates. Furthermore, we present an optimization of an existing powerful, but resource-intensive state-based update approach as well as an approach for in-network filtering of duplicates. These contributions are primarily based on work published in [KDR15] and [KDR16]. The author of this thesis contributed approximately 90% of the scientific content.

2. CONCEPTS FOR FLEXIBLE CONTROL PLANE DISTRIBUTION In the second part of this thesis, we propose an architecture for network control distribution focussing on flexibility in terms of distribution schemes, called ZeroSDN. Our research and the proposed architecture consider many of the aforementioned aspects of distribution and consistency, which are refined in the following. The contributions are primarily based on work published in [KDR17, KDBR17, KDR18].

2.1. MESSAGE BUS AND EVENT ABSTRACTION. A central component of our architecture is the message bus for decoupling controller functionality and control applications from each other and from switches. The message bus is based on the abstraction of events in the data plane and content-based filtering of these so-called data plane events. The event-based decoupling yields numerous benefits over the tighter coupling of standard OpenFlow. It allows for employing a mixture of distribution schemes, replication and partitioning, and inherently implements a load-balancing mechanism. It offers different levels of consistency for state data synchronization. Moreover, its event-abstraction implements a mechanism for the coordination of network control through so-called control plane events, which also allows for the detection of policy conflicts and provides a feedback mechanism for network updates, implementing a precursor for a transactional interface for network updates. The author of this thesis contributed approximately 50% of the scientific content.

2.2. LOCAL DATA PLANE EVENT PROCESSING AND FULL-RANGE DISTRIBUTION. Using a micro-kernel approach to implement very lightweight controller modules allows us to push down control functionality onto the switches and thus implement switch-local control decision making. We call this concept local data plane event processing (LDPEP). Proposing LDPEP, we question the clean-slate approach of network control in SDN, where any logic is removed from switches. With LDPEP, we vastly expand the SDN control paradigm, enabling full-range distribution of SDN control, from fully decentralized control, over local control still profiting from global view (*augmented fully distributed control*) up to SDN-typical fully (logically) centralized control. Allowing a varying scope of local state data enables trading off quality of control decision and consistency requirements (both through the scope of locally held state) with control latency. Consequently, we propose autonomous local procedures, that without any switch-external control allow for temporary fast yet possibly sub-optimal reaction (*intermediate local procedures*), for instance swiftly recovering local link failures (*local fast failover*). We provide a thorough example for LDPEP

with the autonomous forwarding control module, showing control coordination, reconciliation of policy conflicts, and fine-granular local aggregation of global view by leveraging the message bus. The author of this thesis contributed approximately 75% of the scientific content.

2.3. LIGHTWEIGHT VIRTUALIZATION OF LOCAL CONTROL ON WHITE-BOX NETWORKING HARDWARE. Switch-local control mandates an accessible switch control plane, which an emerging class of open networking hardware, so-called white-box switches, fulfills. This property, on the other hand, raises concerns about the safety of the control plane operation that is threatened by the execution of arbitrary local control applications. Thus, we propose safety mechanisms for robust control plane operation based on lightweight virtualization and containerization techniques, providing adequate isolation properties and mechanisms for fine-grained resource control. The evaluation of ZeroSDN's control latency includes measurements of control latency on a physical network testbed based on white-box switch hardware, also evaluating the overhead of state-of-the-art virtualization mechanisms running atop. The author of this thesis contributed approximately 85% of the scientific content.

3. IMPLEMENTATION AND CONSISTENCY CONSIDERATIONS OF A DISTRIBUTED DATA-PLANE APPLICATION In the third part of this thesis, we account for the progression of distribution in data-plane centric SDN. We present an approach called P4CEP as an implementation of a distributed application from the domain of message-oriented middleware on top of programmable network devices using data plane programming—a recently huge trend and SDN's emerging evolutionary step—by using the P4 language. Traditionally, Complex Event Processing (CEP) has been implemented as an overlay of software middleboxes, inferring higher-level knowledge (complex events) by evaluating specific combinations of incoming information (basic events). Enabled by P4's flexible and powerful programming model, we present a data plane implementation of CEP that yields greatly reduced latency and increased throughput due to hardware-based packet processing. Since we want to prevent swapping one dedicated device, a CEP middlebox for another, a dedicated network device solely implementing CEP, we implement a mechanism to allow the co-existence of CEP alongside arbitrary other P4 programs on the same device. We discuss challenges entailed in the distributed data plane processing and address aspects of distribution and consistency in particular for stateful data plane programming, where packet processing changes internal state, which in turn changes the processing of subsequent packets. In distributing CEP, basic events may have to be propagated in a one-to-may pattern, raising

the issue of update consistency of multicast trees, as addressed in the thesis' first part. The implementation of P4CEP consists of a compiler that compiles patterns for the detection of complex events specified in our rule specification language to data plane programs, consisting of a state machine and operators that process *window*s containing historic events. We evaluate the performance and scalability of P4CEP on programmable hardware NICs. These contributions are primarily based on work published in [KMD+18]. The author of this thesis contributed approximately 80% of the scientific content.

1.5 Structure of the Thesis

Chapter 1 has so far given an introduction and motivation for Software-defined Networking in general, and its distribution aspects in order to outline the problem space of consistency in SDN in particular. Having introduced the research area, an overview of the particular focus and research goals of this thesis along with its contributions have been given.

The remainder of this thesis is structured as follows. Chapter 2 lays out conceptual and technical foundations and essential reasoning in the context of SDN as relevant for the subsequent parts. More specifically, it presents a generic SDN system architecture that is refined in later parts as required; specifics of the OpenFlow processing model and southbound protocol specification, as relevant for update consistency, control distribution and to illustrate differences to the data plane programming model; a refinement of data plane consistency along with a classification of existing mechanisms; the design space and trade-offs in network control distribution along with development of architectures and a presentation and discussion of consistency in state synchronization; an overview of white-box networking hardware as used for evaluating ZeroSDN; and data plane programming along with the P4 language and its relation and role for replacement of the NFV and middleboxing model. The subsequent parts of the thesis directly reflect the aforementioned areas of research and contributions. Chapter 3 presents our work on update consistency and management of SDN-based multicast networks. Chapter 4 presents concepts and reasoning for full-range distribution of network control along with a presentation of ZeroSDN, an event-based architecture for flexible control plane distribution. Chapter 5 presents P4CEP as a data plane implementation of complex event processing, focussing on distributed implementation and consistency in stateful packet processing in the data plane. Finally, Chapter 6 concludes the thesis with a summary of our contributions and an outlook on future work.

2 Software-defined Networking

In this chapter, we refine our initial introduction by providing conceptual and technical foundations of Software-defined Networking in general and of the addressed research area of distribution and consistency in SDN in particular along with essential reasoning in order to establish a sound basis for the presentation of our approaches in the subsequent chapters.

2.1 Architecture and System Model

We start by refining the high-level SDN architecture as depicted in Figure 1.1 to a level valid for all of our addressed aspects. Since the scope and focus of those aspects differ, we refine the common system model in the chapters as required but sum up the major differentiations here.

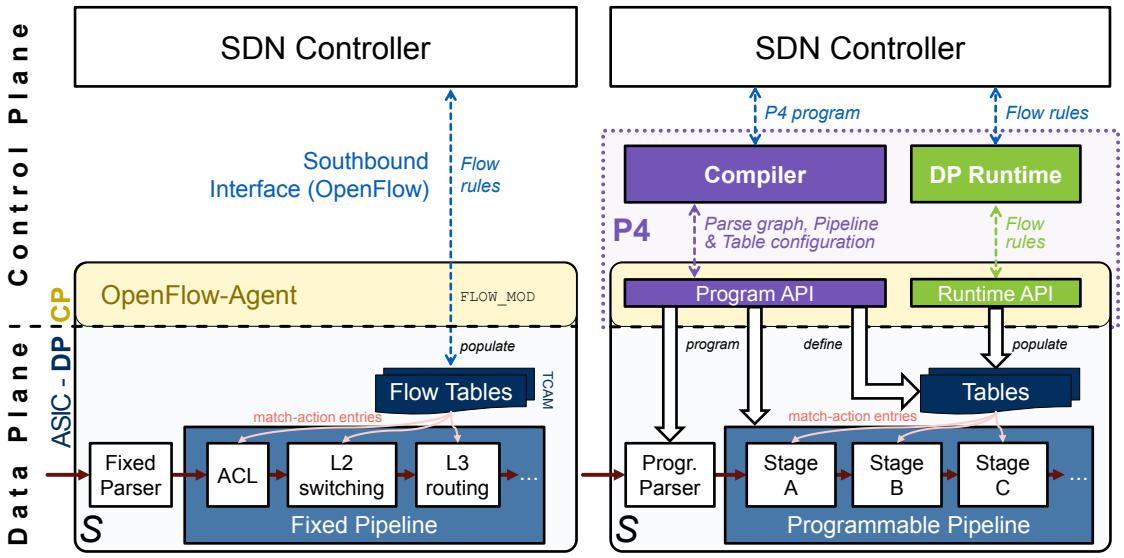
We briefly summarize our common SDN system model as follows. The separation of control plane and data plane is implemented through a southbound interface, allowing a set of distributed controllers in the control plane to configure packet processing behaviour of switches, or other network elements with configurable packet processing, residing in the data plane. A control channel enables bi-directional communication between switches and controllers, passing control actions from a controller to a switch, and data plane events in the opposite direction. The control channel is typically implemented out-of-band in a dedicated and physically or logically separated control network, however in-band implementations exist. Throughout the thesis, we assume an out-of-band control channel. The dedicated northbound interface, enabling network control applications to use controller APIs to implement desired network behaviour (policies), is of negligible importance in our approaches and hence not explicitly modelled but subsumed in the control plane, without loss of generality. Except in Chapter 3, we moreover do not model the management plane explicitly. Furthermore, end-systems, also called hosts, where applications are running and requiring basic or advanced network services such as basic connectivity or advanced messaging patterns like publish/subscribe, are included in the

system model as required. Another differentiation lays in the scope of the control plane. Logically, traditional SDN follows the clean-slate paradigm of network control, where switches are not participating in network control at all, and are thus modelled solely in the data plane. Technically, even in the traditional SDN paradigm, the switch-side implementation of the southbound interface is typically running on dedicated general-purpose hardware units (*control plane processing units*, e.g., CPUs) that are separated from the units for high-speed and specialized packet processing (*data plane processing units* ("switch silicon"), e.g., ASICs), while both units reside on the same switch hardware (see Section 2.5). Thus, the control plane technically extends to the switches to include the implementation of the southbound interface. With ZeroSDN (see Chapter 4) we also logically extend the control plane to the switches by placing actual control logic onto switches.

Throughout this thesis, we use the following convention on notation. We denote a switch entity as S_i. A switch comprises switchports, physical (p_i) or virtual (v_i), possibly connected to another switchport or end-system via a weighted link l. We refer to a controller instance as C_i, where possibly multiple controller instance entities form a logically centralized controller C_{logic}. Control functions are denoted as CF_i and represent network functions (NF_i) or network control applications (see application plane above) implemented within C_i. Considering controller distribution (replication and partitioning), $CM_{i,j}$ denotes a controller module encapsulating CF_i in a replica j forming a logical control module $CM_{i,logic}$. A controller module that is executed at a switch, i.e., a local controller module, is denoted as L_i. State is held in a logically centralized state store SM_{logic}, such as the Network Information Base (NIB), and might analogously be physically distributed, where a state module SM_i is associated to a co-located $CM_{i,j}$.

2.2 OpenFlow—The de-facto Standard of SDN

Due to its central, yet recently diminishing role as a de-facto standard of SDN, in this section we provide an introduction of relevant parts of OpenFlow (OF), in particular for distribution aspects within its processing model and southbound protocol specification. We also discuss limitations of OpenFlow adoption in recent hardware switches.



(a) The southbound protocol OpenFlow is used to populate statically-structured tables of a fixed pipeline and fixed parser on a fixed-function ASIC with match-action entries at runtime.

(b) A P4 program is compiled into parser, pipeline and table structures that are deployed to a reconfigurable ASIC, programmable NIC, etc., before tables are populated at runtime.

Figure 2.1. High-level differences in programming models of conventional fixed pipeline switches, e.g., OpenFlow switches (see Section 2.2.1), and reconfigurable pipeline switches using data plane programming with the P4 language (see Section 2.6).

2.2.1 The OpenFlow Processing Pipeline

The original paper [MAB+08] proposes OpenFlow as a pragmatic compromise to promote the standardisation of switch interfaces to allow researchers for uniform access to internal flow tables and hence experimentations at line-rate and high port-densities on heterogeneous switches, while not forcing vendors to disclose insights into their internal switch architectures.

OpenFlow implements a flow-table abstraction for programming of network behaviour, as described in the introduction and illustrated in Figure 2.1(a). In OpenFlow, communication flows through the network are defined through flow-table entries, matching on headers of existing and well-established layer 2 to layer 4 protocol headers and physical layer properties like ingress switchport-identifier. In the evolution of the OpenFlow protocol specification [Opeb], the number of supported matching fields have been continuously increased (from 20 in OF v1.0 to 41 in OF v1.5), as has the number of applicable actions (from 13 to 59) [KREV+15], in an attempt to extend and flexibilize OpenFlow-

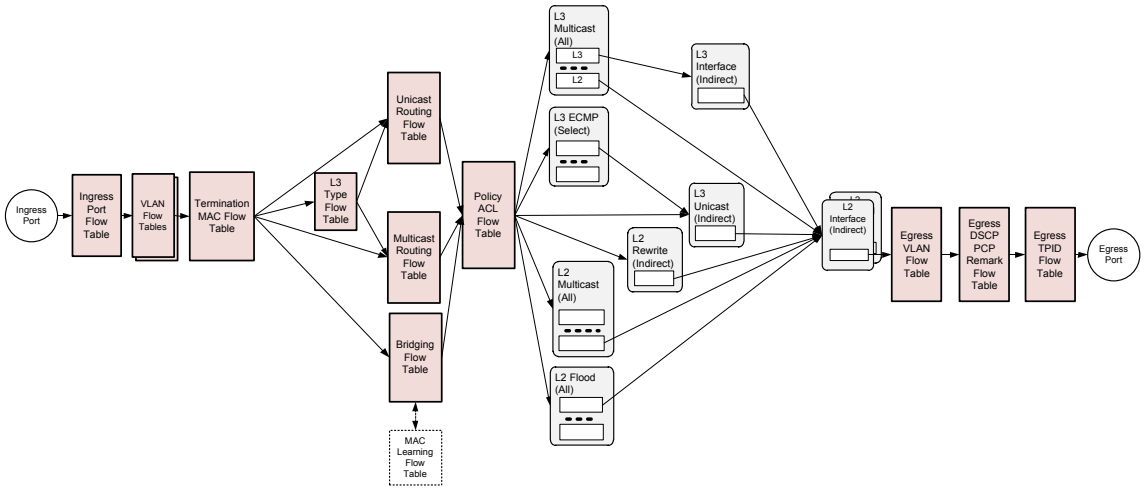


Figure 2.2. Fixed processing pipeline for bridging and routing functionality of a switching ASIC under the OpenFlow data plane abstraction (OF-DPA) [Brod].

based SDN. However, in contrast to data plane programming, matching in OpenFlow is still limited to established network protocols. We provide a high-level comparison of the programming abstractions of OpenFlow and data plane programming in Figure 2.1, while introducing data plane programming and the P4 language in Section 2.6.

The single flow-table model in OF v1.0 has soon been extended to a multi-stage pipeline consisting of multiple flow tables. In order to increase throughput, hardware switches implement a pipeline execution model where packet processing is split into stages that are associated to distinct execution engines of the switch silicon which run and thus process packets in parallel [BGK+13]. The stages implement different parts of packet processing for instance operating on different header fields (for matching and for applying actions). For instance, the OpenFlow Data Plane Abstraction (OF-DPA), which abstracts hardware pipelines of current OF-enabled hardware switches, differentiates 34 tables (ingress, action, and egress tables) for the implementation of a number of fixed pipelines implementing common network functions in the context of data center networking, like bridging/routing, tunneling, ACLs, traffic shaping, metering, etc. Figure 2.2 exemplarily shows a fixed processing pipeline for bridging and routing functionality. Note that those tables are heterogeneous, resulting in limitations on the mapping of pipeline stages to flow tables as well as restrictions on table capacities, flow-entry types, matching semantics (exact matching, ternary matching, match-field sizes) and correspondingly employed memory type (SRAM, CAM, TCAM), applicable actions, and counters.

Due to its crucial role in particular for update consistency, we would like to briefly

address limitations of flow-table memory. In particular, tables for exact matching, for instance for lookups of MAC-address to switchport mapping, could be stored in SRAM. However, the worst-case linear lookup time complexity and even faster lookups through algorithmic-optimizations like hashtables or tries may yield insufficient scalability, considering matching of tens to hundreds of millions of packets per second in modern switches with terbit-per-second-range throughput leaving only a tiny time budget of a few clock cycles for matching. Hence, optimized content-addressable hardware memory (CAM) with O(1) lookup time complexity are often used. Binary CAMs are typically used for fast exact matching and combined to multiple CAMs for longest prefix matching (LPM) of fixed prefix-lengths. For arbitrary matching, also called wildcard matching, ternary CAM (TCAM) are the most prevalent memory type to be implemented on switching ASICs.

TCAM's superior lookup time complexity comes at the cost of greatly increased power consumption (about a factor of 100 in comparison to conventional RAM [STT03]), ASIC die size consumption, and hence price. Thus, it is a fairly scarce resource, typically implemented in sizes to accommodate in the order of only thousands of entries with fully-quantified ternary matches, i.e., ternary matching on any header field, on recent typical top-of-rack data center switches [SCF+12, KPK15, PIC18].

2.2.2 The OpenFlow Control Channel

The OpenFlow control channel provides an interface for connecting OF-enabled switches to OF-capable controllers. In addition to the aforementioned bi-directional control communication of control actions, most prominently flow modification messages (`OF_FLOW_MOD`), and data plane events, most prominently packets without a matching flow rule (`OF_PKT_IN`), the controller can also inject packets to the data plane (`OF_PKT_OUT`). The OpenFlow control channel by default is secure, however plain TCP can be used. By using TLS, it inherits the security features authentication, integrity, and confidentiality.

As mentioned earlier, the control channel can be implemented in-band and, most typically, out-of-band. OpenFlow supports multiple concurrent control channels from a switch to controllers, following two schemes. First, *auxiliary connections* provide redundancy of a single control channel by maintaining multiple connections to the *same* controller, for instance using different transport protocols (TCP or UDP or their secure implementations TLS and DTLS). Second, since OF v1.2 the *multiple controller functionality* may improve reliability by maintaining multiple concurrent control channels to *different* controllers in an active/active and active/passive scheme. Controllers coordinate which channels are to be

active, i.e., the controller gets all switch-solicited messages and can fully control the switch, and which channels are to be passive, i.e., the controller does *not* get switch-solicited messages and can *not* fully control the switch. However, vendor-specific implementation in hardware switches might impose restrictions, e.g., on the number of concurrent controller connections [Hew16], limiting practicability. To overcome compatibility issues, an $n:m$ switch-controller-mapping can be implemented by an OF-external control channel demultiplexer, such as ZeroSDN's SwitchAdapter controllet (see Section 4.4.2), that runs locally or in close proximity to each switch and proxies OpenFlow messages from the switch to respective controllers, and vice-versa.

2.2.3 OpenFlow-specific Mechanisms for Update Consistency

As we will detail in Chapter 3, data plane consistency puts stringent requirements on the handling of OpenFlow messages on switches, in particular requiring guarantees about (1) the ordering in which incoming control actions are processed and (2) about the reliability of message processing. In general, a switch has to entirely process every control action conveyed in a message received from a controller. Control actions in general are not positively acknowledged, however in case an error occurs, the switch has to inform the controller, thus implicitly implementing a negative-acknowledgement mechanism, which is a necessary condition for the processing reliability requirement.

The ordering requirement (1) is not guaranteed by default. Most important for update consistency, the order of the insertion of flow entries in tables might arbitrarily differ from the order the corresponding `OF_FLOW_MOD` messages have been received. However, OpenFlow provides a *barrier* mechanism that can be used to enforce in-order processing of messages. Upon reception of an `OF_BARRIER_REQ` message, the switch ensures that all previously received messages have been processed and subsequently acknowledges with an `OF_BARRIER_REPLY`, before processing any new messages. This mechanism also can be used to effectively separate interdependent control actions.

In recent OF versions (since v1.4), the *bundle* mechanism effectively satisfies the reliability requirement (2) by providing transactional, i.e. *all or nothing*, semantics for message processing. It allows for the definition of a sequence of control actions, that are applied as a single OpenFlow operation—the *bundle*. Control actions are collected in a temporary staging area without taking effect. Upon being committed, they are applied to the switch state, e.g., tables, and acknowledged, iff all control actions have been successfully executed. Otherwise, all state control actions are rolled-back and a

negative acknowledgement, i.e., the error message of the failing action, is sent to the issuing controller. Moreover, the bundle mechanism recognizes two flags, for in-order execution of control actions within a bundle, analogously to the barrier mechanism, as well as for *packet-atomic execution* of the bundle, which means, that packets incoming while the bundle is applied should either be processed with none or with all of the control actions having been applied, which resembles a strict update consistency semantic.

However, both, the barrier and the bundle mechanism are not mandated to be actually implemented by OF-compliant switches. To date, most OF-enabled hardware switches do not implement bundles at all. Furthermore, a recent study revealed flaws in their implementation on OF hardware switches [KPKC18] that do claim to support them.

2.3 Data Plane Consistency

In this section, we provide background information regarding update consistency, which addresses implications of changing flow-table entries on the adherence of policies and other desired network properties. After introducing the network update problem, we refine the update process in the network control loop and describe classes of mechanisms that solve it, i.e., mechanisms that consistently implement network updates.

2.3.1 Update Consistency

Today's softwarized networks exhibit a high degree of dynamics, which is also a main motivation for the SDN paradigm in the first place. For instance, in modern data center networks, especially in the cloud computing context, shifts of network load and network functions, e.g., through migration of virtual machines or dynamic scaling processes in Network Function Virtualization (NFV), or provisioning of cloud-tenant resources that are to be interconnected by the underlying network (network virtualization), require an almost continuous adaptation of the network. To implement adaptations, the network has to be reconfigured inevitably. The SDN *network control loop* (see Figure 2.3) aims to maintain a steady operation of the network in accordance to global network policies by reacting to disturbances, i.e., changes in the control plane or the data plane (both deliberate, such as policy changes, and involuntary, such as load shifts, failures, etc.), by computing necessary adaptations and consequently updating the data plane accordingly.

Updating the data plane can be described as the transition from an old network state λ to a new network state λ', where a state λ at time t comprises the set of flow-rules U

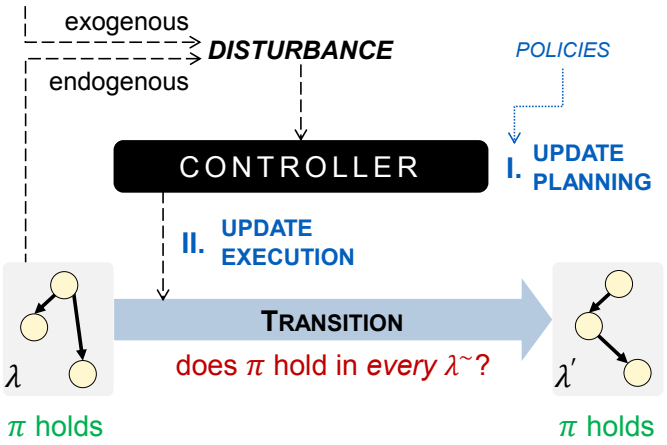


Figure 2.3. A typical SDN network control loop maintaining steady operation of the network in accordance to global network policies, reacting to disturbances by adapting, i.e., updating the data plane.

that are installed in the switches flow tables at t. A state hence is encoding all network policies at t. To advance to a new network state λ', the affected switches have to be updated individually. Since updating flow-table entries is an inherently asynchronous process, i.e., flow mod messages can get lost or delayed and furthermore the delay until a rule update has been applied varies among the switches, packets in the data plane may be processed according to a mixture of new and old rules as they traverse the network, while it is being updated. Thus, a network property π that holds in λ and in λ' my not hold in an intra-update state $\lambda^{\sim}$, possibly violating network invariants Π, where Π is given by the network policies comprising arbitrary π. Figure 2.4 shows an example of updating a flow's route through a network, leading to two inconsistency effects, looping of packets (due to transient cycles) and en-route dropping of packets (due to a missing flow-rule (black hole)). If consistency properties are *not* violated, updates are denoted to be *consistent* or *correct*. Reitblatt *et al.* [RFR+12] stipulate a refinement of the notion of update consistency considering temporal aspects of the data plane: (1) *per-packet consistency* requires that during an update, a packet is processed entirely according to a single state, i.e., a single policy or a single composition of policies, whereas (2) *per-flow consistency* requires that packets of an entire flow, for instance, all segments of a TCP connection, are processed according to a single state. The *network update problem* [RFR+12, MW13] asks for a sequence of control operations that implement a network state transition $\lambda \rightarrow \lambda'$ such that a set of invariants Π holds in any possible intermediate state $\lambda^{\sim}$, or in other words, the ensure the transition consists entirely of correct updates.

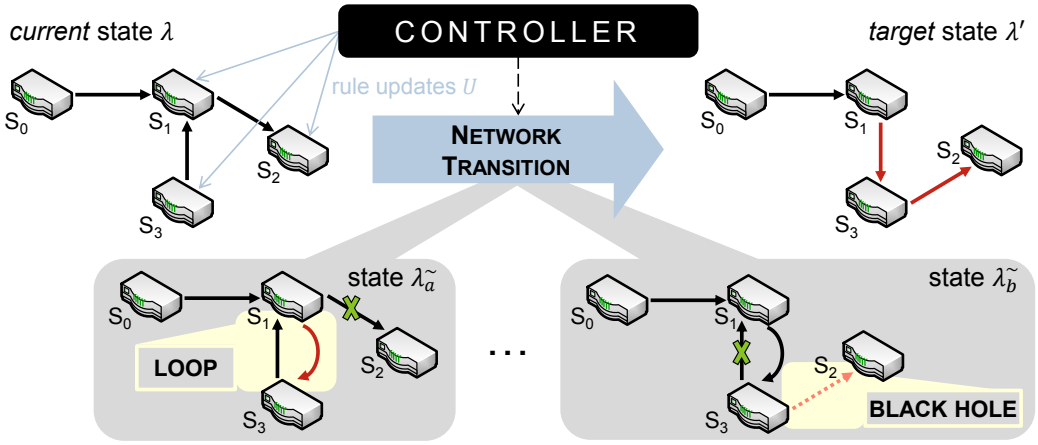


Figure 2.4. Illustration of an exemplary routing update leading to intermediate states $(\tilde{\lambda_a}, \tilde{\lambda_b})$ during the state transition from *current* to *target*. Temporarily, a loop and a black hole are introduced respectively, violating desired network properties (invariants).

Multifarious correctness properties have been stipulated in literature. In the following, we give a classification and present the most relevant properties. (1) *Path-properties* [MW13, FMW16, XYL+17, FLMS18], including way-pointing (enforcing the traversal of specific switches or end-systems), shortest-path-routing, loop-freeness (absence of transient loops), or drop-freeness (absence of missing segments in routes); (2) *network-properties*, including isolation of specific flows (end-systems, traffic classes, etc.) [LRFS14], duplicate-freeness (absence of packet duplication, see Section 3.3); (3) *link-properties* [JLG+14, ZLT+18] considering capacity limitations of links (congestion-free updates) and timing constraints; and (4) *switch-properties* such as memory limitations (flow-table size) or specific switch capabilities [RHC+17].

A violation of invariants in the best case reduces the network's efficiency, possibly violating soft requirements such as Quality of Experience (QoE). In the worst case, it may severely jeopardize stipulated hard invariants, e.g., when black holes occur in routing, which is a network function of utmost importance. Since the management of networks includes definition and implementation of requirements on the network operation, we argue in Section 3.1 to create awareness about update consistency in network management in order to be able to circumvent effects that stem from update inconsistencies. This implies the selection of suitable update methods based on the network specifications to maintain.

2.3.2 Data Plane Update Mechanisms

We refine the network control loop[1], as illustrated in Figure 2.3, that an SDN controller implements as follows. After monitoring events in the network and subsequently analysing whether these events are to trigger an adaptation of the network, a network state transition is possibly implemented by a *planning phase*, where a set of necessary flow updates U to implement the adaptation are determined, and an *execution phase*, in which U is applied to the data plane. There are two classes of mechanisms for achieving update consistency. (1) The very powerful two-phase update mechanism that guarantees the maintenance of arbitrary invariants during a network transition, and (2) lightweight mechanisms centered around coordination in the planning phase (generation of update plans).

(1) Reitblatt's 2-phase update approach [RFRW11, RFR+12] effectively eliminates the possibility of any $\lambda^{\sim}$. Consequently, iff π holds in both λ and λ', π also holds during $\lambda \rightarrow \lambda'$. In other words, any property that holds in both the old and new state is guaranteed to hold during their transition. The key to such strong consistency is to employ state information in both flow-rules and packets traversing the network. A version tag v is injected to each packet that enters the network, such that flow-rules installed in the switches can match on a specific version tag. Hence, each state λ_i can be associated to a version tag v_i. Reconfiguration is achieved by installing a new set of rules with an increased version match field v_{i+1} (associated to λ') on all switches affected by the transition. Once the new set of rules has been entirely installed, the ingress switches are instructed to tag newly incoming packets with v_{i+1}. Increasing the tagged version number effectively switches the configuration that new packets are processed according to and hence λ' is effective for all newly entering packets. A traversing packet is thus processed according to a single coherent state (*per-packet consistency*), such that there exists no $\lambda^{\sim}$. While being very powerful on the one hand, the two-phase update approach has severe drawbacks on the other hand, limiting its practicability. First of all, it is not transparent for data plane traffic since it actively modifies the packets by encoding v in a vacant header field (typically the VLAN tag), consequently placing a technical dependency on an available[2] header field. Second, unless the installation of the new rule set encoding λ' is completed, it requires to store both, the old and new rule sets in parallel in each affected switch. Hardware switches store their flow tables in ternary content-addressable memory (TCAM), which is a power-hungry and fairly limited resource (see Section 2.2.1), with typical capacities of

[1] overall resembling a MAPE cycle as known from the domain of Autonomic Computing [KC03]

[2] that is, a header field which is neither used for other flow specification, nor used by any (future) traffic in the network

thousands of rule entries for current hardware switches [SCF+12, KPK15, PIC18]. Third, (only) after the old rule set is not required any more, i.e., all packets of v_i have left the network, all rules matching on v_i have to be removed from the switches, causing either control overhead through flow-deletion messages or increasing update completion time in case of relying on OpenFlow's automated flow-timeout removal mechanism. Furthermore, depending on the update frequency, there might by more than two rule sets be installed in parallel, consequently exacerbating rule-space consumption of affected switches further more. Overall, this mechanism inflicts overhead in the execution phase in terms of rule-space and update time. In [KRW13], the authors present an extension to their original approach allowing to trade-off time required to perform a consistent update against rule-space overhead required to implement it. A special two-phase approach, however of typically limited practicability, is to redirect all affected packets to the controller while a transition is being implemented [McG12].

Considering the drawbacks of two-phase updates, (2) stateless approaches based on coordination in update plans received a lot of attention. This class of update mechanisms does not rely on state information and in general can not guarantee the maintenance of arbitrary invariants, however, algorithms that guarantee specific properties (see Section 2.3.1), including loop-freeness and drop- or duplicate-freeness in multicast networks (see Chapter 3) exist. Basically, the overhead to ensure correctness is shifted from the execution phase (as with the two-phase updates) to the planning phase. In this phase, stateless approaches generate *update plans* (also called *update schedules*) that define ordered sets of updates to tightly coordinate the actual execution of the updates later on in the execution phase. Typically, update plans rely on an appropriate ordering of updates among switches (update-ordering approaches) or their precise temporal execution (timed-update approaches). They may consist of regular flow-rule updates, temporary ("helper") updates, logic operators, and timed-updates [MSM16], i.e., time-triggered updates with an associated wall-clock time to become effective. For the determination of update plans, search-based approaches such as [LWZ+13, MFC14], and constructive approaches such as [JLG+14, MW13], have been proposed. We refer to [FSV19] for a survey of constructive approaches. We present our update-ordering approach for multicast networks along with our algorithm for update plan calculation in detail in Section 3.4, where we also present a hybrid approach that combines a two-phase mechanism applied to critical updates with a lightweight update ordering mechanism applied to uncritical updates.

2.4 Design Space and Trade-offs in Network Control Distribution

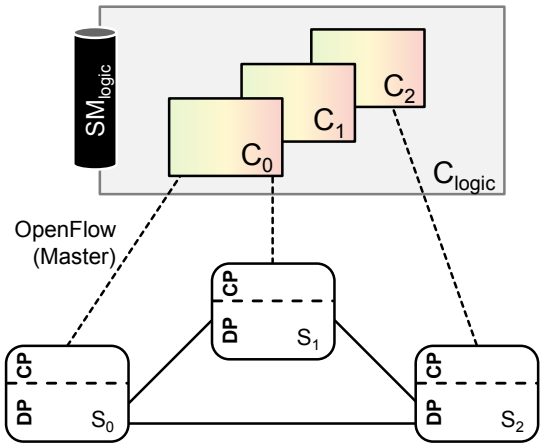
In this section, we lay out the design space of network control distribution. We refine the distribution schemes replication and partitioning, as introduced in Section 1.2, by reflecting the evolution of controller architectures proposed in literature before we discuss trade-offs regarding consistency in state synchronization.

2.4.1 Evolution of SDN Controller Architectures

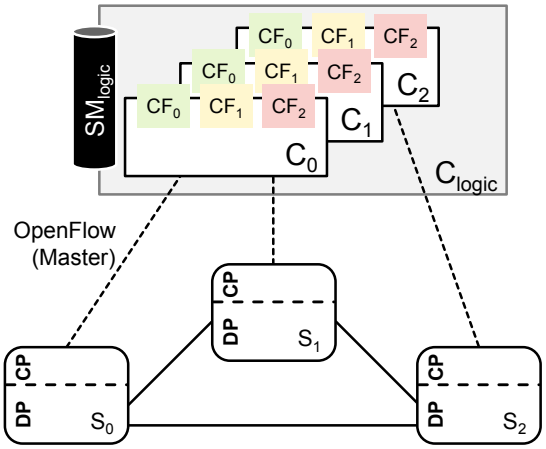
Many SDN controllers have been implemented so far based on the concept of logically centralized control. For most comprehensive surveys we refer to [KD17, BSM18], while focussing on most relevant or prominent approaches in the following. Figure 2.5 depicts the evolution of controller architectures with respect to distribution and modularization.

First SDN controllers were *monolithic systems*, implementing the controller as one process. The SDN controller connects through the southbound interface to the switches using, for instance, the OpenFlow protocol, and the control applications interface with the SDN controller through a northbound interface, e.g., a Java API or REST interface. Although physical centralization is appealing due to the simplicity of its implementation, physically centralized controllers cannot meet the scalability requirements of modern networks, in particular considering data center networks (DCN) with scales in the order of tens of thousands of switches [BAM10, YTG13]. Most prominent representatives of physically centralized controllers are the controller frameworks NOX [GKP+08], Maestro [Ng], Floodlight [Big], and Ryu [Ryu]. Furthermore, it has also been shown that physical control centralization is not a viable option for SDN-based wide-area networks (WANs) regarding failure-resiliency and scalability [MK17]. To increase availability and scalability, the monolithic process implementing all control logic may be replicated (Figure 2.5(a)) in an active or passive replication scheme [FBMP13].

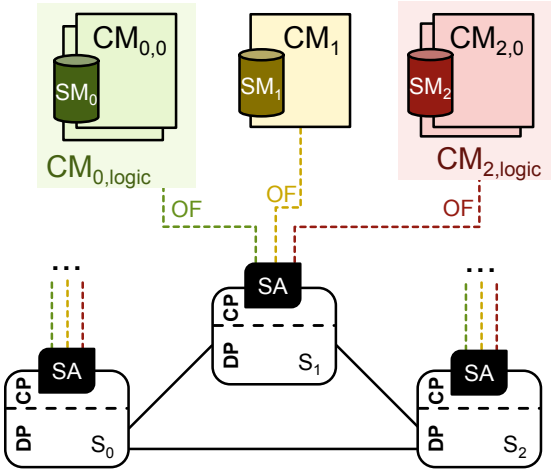
Very similar to the evolution of monolithic operating system kernels like the Linux kernel, this monolithic design was soon extended to a *modular monolithic design* (Figure 2.5(b)), where control modules implementing certain control functions (*network functions*) can be dynamically (un-)loaded into the controller process at runtime. Two examples showing that this design is still used in practice are the popular ONOS [BGH+14, Thea] and OpenDaylight [OF] controllers relying on OSGi [OSG]. However, their modular controller architectures remain monolithic since they still rely on a central controller executing



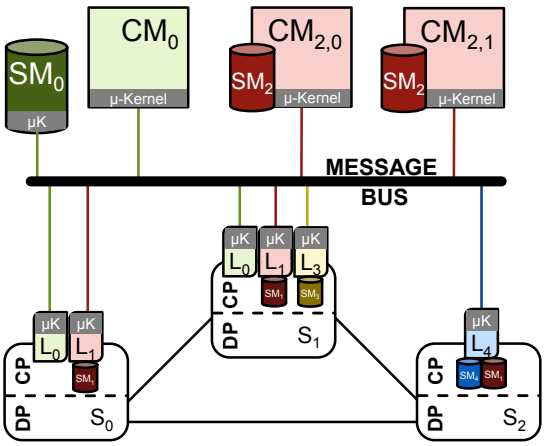
(a) Non-modular, monolithic, replicated;
C: controller instance, SM: state module, S_i: switch



(b) Modular, monolithic, replicated;
C: controller instance, CF: control function, SM: state module



(c) Modular, partitioned by network function, replicated;
CM: control module, SA: local OpenFlow demultiplexer (switch adapter)



(d) Envisioned full-range distributed control of local (*L*) and external control modules (*CM*) and corresponding state modules (*SM*) (see Chapter 4)

Figure 2.5. Evolution of distribution in SDN controller architectures

all modular control functions in one process. This generation of controllers typically employs replication. Most prominent representatives of replicated frameworks besides OpenDaylight and ONOS are Onix [KCG+10] and HyperFlow [TG10]. In order to improve not just availability but to actually improve scalability, the set of switches to control is partitioned, where each controller replica is associated a mostly disjoint subset of switches to control (*horizontal partitioning*). Typically, the set of switches is partitioned along topological properties. For instance, in the common leave-spine topology in DCN, each leave can be associated a dedicated controller replica. While horizontal partitioning allows for scalability with the network size, it also raises the need for coordination. In the DCN example, partitioning along the leaves requires tighter coordination on the spine layer. For instance, establishing dedicated inter-leave communication from an end-system residing in leave a and one residing in leave b requires involvement of multiple controller instances, controlling the leave switches of a and b and controlling the spine switches on the path $a \rightarrow b$.

Another partitioning scheme distributes network control along functional properties (*vertical partitioning*, illustrated in Figure 2.5(c)). Similar to the modular monolithic design, individual control functions can be factored out into control modules, which are now partitioned between different physical machines instead of fully replicating all control functions on all machines. For instance, Kandoo [HYG12] proposes a two-layered controller hierarchy, where a root controller handles only rare events of global scope, and local controllers handling all frequently occurring local events. On the same line, Google's B4 [JKM+13] implements separation of concern by employing a control layer for intra-DCN traffic and a control layer for inter-DCN, i.e., WAN traffic. Another example is partitioning along the flow-space, where for instance, long-lasting high-traffic flows (*elephant flows*) such as long-lasting TCP connections possibly transporting bulk data are handled disjointly from short-lived low-traffic flows (mice flows) such as connection-less UDP traffic possibly transporting traffic of real-time applications like media voice-over-IP (VoIP). This way, dedicated control modules can account for flow diversity by employing flow-type specific optimizations, for instance, traffic engineering for elephant flows (see Section 4.3.2.3) and optimizations focussing on timeliness for mice flows. Note, that the vertical partitioning requires multiple concurrent control channels (see Section 2.2.2) in the case of non-disjoint control module association.

A last step is to freely combine replication and partitioning schemes while also allowing to push down control logic onto switches themselves (Figure 2.5(d)). With ZeroSDN in Chapter 4 we elaborate on this scheme and make a case for full-range distribution

of network control by discussing its rationale and presenting its underlying event-based network control.

2.4.2 State Consistency and Synchronization

Having discussed different distribution schemes employed by established SDN controllers, we now address implications of distribution in terms of consistency in the control plane. We denote *control plane state* to comprise all data relevant for control plane operation and hence control decision making, or more specifically, the global view of the network, which is typically referred to as network information base (NIB) in literature, as well as state of control applications.

Inconsistencies in the control plane can arise in two ways. (1) Inconsistency between assumed network state (control plane state) and the actual network state (physical data plane state). Just as for update consistency (see Section 2.3), an (even non-distributed) controller that takes a control decision based on a stale view of the network might introduce inconsistency effects that violate network invariants. (2) Inconsistencies between control plane sate of different controller instances in a distributed control plane. Controller instances that take a control decision based on a view that differs among the instances may lead as well to the violation of network invariants, in particular of policies defined by network control applications or severely degrading application performance, as shown for inconsistencies in an SDN-based load balancer [LWH+12]. In the following, we concentrate on the latter, i.e., (2) consistency of state in a distributed control plane.

Temporary state inconsistencies are inevitable in distributed systems. Based on the consistency model, such inconsistencies are either resolved internally and guaranteed *not* to be exposed to clients of such systems (strong consistency) or be tackled in a best-effort manner where internal inconsistencies *might* be (temporarily) exposed to clients (weak consistency). Preventing negative implications due to inconsistency effects calls for mechanisms ensuring strong consistency of control plane state, which consequently ensure that any controller instance takes the same control decision, given the same input event that triggers the decision making. Basically, consistency of the control plane state can be ensured by serializing updates to that state, i.e., all controller instances have to reach consensus about the order of events that modify the control plane state. In order to reach that consensus, the controller instances have to coordinate, typically employing consensus protocols, such as Viewstamped Replication [OL88], Paxos [Lam98], Raft [OO14], or Zab [JRS11].

However, synchronization mechanisms ensuring strong consistency come at the cost of introduced synchronization overhead that may compromise responsiveness to handle data plane events. For instance, Onix [KCG+10] provides a replicated transactional database backed by a replicated state machine (SMR). While providing strong consistency, the authors state severe performance limitations for high rates of control actions that modify the control plane state and hence trigger a synchronization of that state. This trade-off between consistency semantics and synchronization overhead and hence control latency has been subject of a large body of literature debating the question of determining an adequate level of consistency for control plane state [KCG+10, TG10, LWH+12, BRKB13, KZFR15, YG16a, PZH+17]. On the one side of the spectrum, for instance Levin *et al.* [LWH+12] motivate and implement a strongly consistent state synchronization, as well as Bothelo *et al.* who rely on an external data store backed by an SMR approach based on BFT-SMART [BSA14], yielding "strong consistency at acceptable" performance bearable for certain SDN applications. Katta *et al.* with Ravana [KZFR15] require an even stronger semantic of *exactly-once* delivery of controller-switch messages. On the other side, a number of works question the necessity of strong consistency. For instance, the authors of SCL [PZH+17] argue that too strict requirements on state consistency hinders control plane operation since every update operation on control plane state requires a quorum to be created upfront, which limits control responsiveness. Furthermore, the authors argue, that the even stricter exactly-once requirement of Ravana more drastically degrades availability, since it requires the system to be unavailable in the presence of failures.

Another question debated in literature is whether strong consistency should be the *sole* option. While for instance Hyperflow [TG10] relies solely on passive synchronization of control plane state yielding weaker *eventual consistency*, other controllers allow for a differentiation of control plane state by providing strong consistency for important state data, for instance, global policies relevant for all controller instances, and less strong semantics for less important state data, for instance transient data that is less relevant for other controller instances. In particular, the influential Onix framework in addition to the replicated transactional database providing strong consistency as mentioned above also provides a synchronization mechanism based on distributed in-memory hash tables (DHT), yielding weaker eventual consistency. As for the carrier-grade controllers OpenDaylight and ONOS, they both provide a strongly consistent data store based on Raft, where ONOS, just like Onix, additionally provides an eventual consistent data store based on an optimistic replication mechanism assisted by a gossip-mechanism (anti-entropy protocol)

[BGH+14, MGBM17]. A recent paradigm called *adaptive consistency* even calls for a mechanism to adjust the consistency level during runtime [AM16, SK18].

In-line with the argumentation about providing flexibility in terms of consistency semantics, we present in a student's work [Str15] a proof-of-concept implementation of an externalized distributed data store for OpenDaylight, based on the ISIS^2 toolkit, which follows the virtual synchrony (VS) model [BJ87]. Virtual Synchrony is solving the problem of ensuring a total ordering in the delivery of multicast messages (atomic multicast [DSU04]) which is equivalent [CT96, MHS11] to the consensus problem as used in SMR for the serialization of inputs to instances of a replicated state machine. Based on the observation that applications often do not require solely strong consistency, ISIS^2 provides synchronization primitives of varying fault models and guarantees regarding message ordering, and hence consistency semantics, ranging from FIFO over causal and total ordering to in-memory Paxos and disk-persisted Paxos.

We conclude from literature, that the concentration on strong consistency for control plane state synchronization is questionable due to its high overhead and mechanisms of less strict semantics have shown to be sufficient at (greatly) improved control responsiveness. We would like to add that application-specific optimizations of state synchronization exist, as shown for instance in the context of pub/sub in messaging middleware [BTK+15]. They typically rely on the principle of reducing the scope of data that has to be synchronized.

With ZeroSDN we draw a lesson from the debates in literature by combining the approach to provide flexibility with respect to distribution schemes and consistency in synchronization of control plane state with a means to reduce the scope of data to be synchronized. Our event-based message bus abstraction (see Section 4.2) implements state synchronization by propagation of input events among controller instances with a flexible dissemination scheme (content-based pub/sub). As shown above, there is a large set of synchronization techniques available, which our message bus can transparently implement. To increase responsiveness, i.e., decrease control latency, we enable a reduction of the scope of control plane state to be synchronized by allowing switch-local control decision making operating on a scope of state data flexibly ranging from solely local to full global view. Hence, we enable yet another trade-off between scope of state, synchronization overhead (and hence control latency), and quality of control decisions (see Section 4.3.1).

2.5 White-box Networking—Anatomy of Open SDN Hardware Switches

Alongside the proliferation of SDN, recent years have seen an increasing trend towards white-box networking. In traditional black-box switches, the control plane is tightly coupled to the underlying hardware and is only accessible through proprietary CLIs or APIs and, in case of SDN support, an interface for remote programmability of the data plane behavior, e.g., through OpenFlow. SDN separates the data plane from the control plane, allowing for control decisions based on a global network view typically taken remotely in the control plane. Similarly, white-box networking consequently decouples the data plane, where specialized hardware (typically ASICs) process packets, from the control plane, where control-software determines the behavior of this processing. In white-box networking, the switch hardware is not tightly coupled to the switch's software stack, the so-called network operating system (NOS). Typical white-box networking NOSes consist of a standard Linux OS and control software running atop, forming the switch's control plane, as illustrated in Figure 2.6. This decoupling is for instance reflected in the fact that white-box switch hardware typically is sold separately from the NOS. Similar to the transition in the server market from proprietary server software and hardware to open operating systems like Linux on commodity off-the shelf hardware, or the decoupling from proprietary hardware (softwarization) of network functions (NFs) in Network Function Virtualization (NFV) [Eur12], white-box networking offers superior flexibility at greatly reduced capital expenditures. The proliferation of white-box switches is reflected in the increasing number of open specifications of both, hardware and software for white-box switches that are provided to the public domain by big players like Facebook and Microsoft for instance in the OpenCompute Project [Opea]. The white-box market share is expected to double within the next five years [Mar].

While white-box switches typically feature the same data plane processing hardware (switch silicon) as proprietary products, their computing resources on the control plane have reached a level comparable to small workstations and are still becoming increasingly powerful. Contrasting black-box switches, white-box NOSes are completely accessible, allowing for the execution of arbitrary applications on their control plane. These properties allow for the exploitation of the switch's locality. We present our concept of Local Data Plane Event Processing (LDPEP) later in Section 4.3.2.

One thing to note is that the current white-box switch landscape exhibits a high heterogeneity with respect to hardware, i.e. switch silicon and control plane architecture and

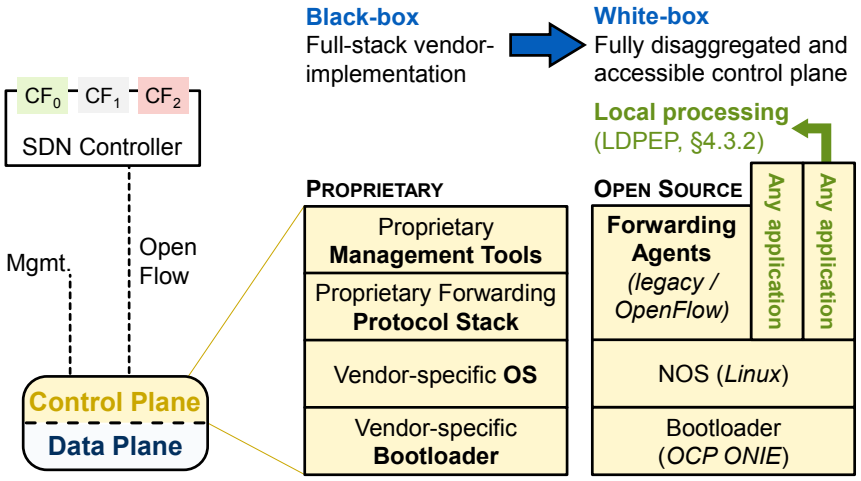


Figure 2.6. Comparison of software stacks of conventional (black-box) and white-box hardware switches where applications can be executed on an accessible control plane comprising mostly open source software.

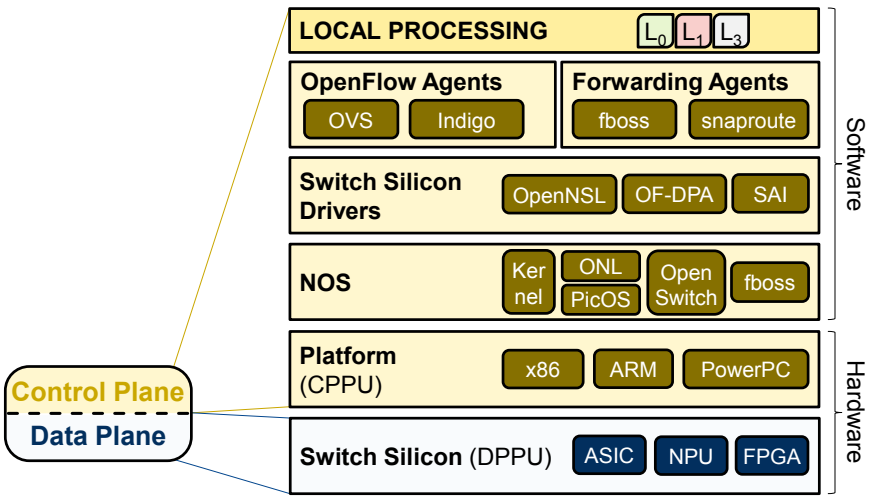


Figure 2.7. Heterogeneity of White-box switch's hardware and software stacks along with typical components.

software, i.e. operating systems and forwarding agents, as illustrated in Figure 2.7.

A first differentiation is found in the switch-silicon, which differs in type, e.g. ASIC or NPU, and model. Considering forwarding performance, flexibility, capability, and accessibility, the selection of switch silicon is crucial. A second differentiation lays in the hardware architecture of the control plane as one of x86, PowerPC, or ARM.

On the software side, although all currently available NOSes are Linux-based, they differ in the aspects openness (closed- or open-source), used kernel, and Linux-distribution. A typical white-box network operating system (NOS) comprises the following components: (1) A base OS, typically Debian with a Linux-kernel. (2) Components for accessing platform hardware including the switch silicon, fans, LEDs, etc. Access to the switch silicon is provided through drivers, which are built against a typically proprietary silicon SDK. The top layer of the driver offers an abstracted API for configuration of the switch silicon hardware pipeline. The most relevant APIs are Broadcom's Open Network Switch Library (OpenNSL [Brob]) and OpenFlow Data Plane Abstraction (OF-DPA [Broc, Brod]) as well as the generic switch abstraction interface (SAI [Pro]). (3) A forwarding agent that interfaces with the driver to program the data plane. Most prominent are: Indigo OpenFlow Agent (OF-DPA), SnapRoute (OpenNSL, SAI), and Facebook FBOSS (OpenNSL). Prominent NOSes include (a) Open Network Linux (ONL): open-source (part of the *Open Compute Project*), broad selection of silicon drivers and forwarding agents, (b) Pica8 PicOS: proprietary, based on Open vSwitch (OVS), and (c) Cumulus Linux: proprietary. All named have open control plane access, however, just ONL and PicOS provide OpenFlow forwarding agents. We discuss the implications of white box switche's openness and heterogeneity in Section 4.3.4.

2.6 Data Plane Programming—The Evolution of Data Plane centric SDN

Recent developments in SDN have given rise to a new evolutionary step of network programmability. Data Plane Programming, like advocated by the popular P4 initiative [BDG+14], has hence become a huge trend in SDN. It features protocol-independent and flexible packet processing in networking hardware, opposing OpenFlow's matching mechanism that is limited to static headers of established network protocols and the rather static hardware processing-pipelines of traditional switch silicon. In a nutshell, data plane programming leverages the increased capabilities and programmability of

modern networking hardware, such as network processors, FPGA-augmented switch silicon, programmable switching ASICS like the popular Tofino ASIC, or programmable NICs to extend the expressiveness of packet processing in the data plane. Data plane programming paves the way for complex yet efficient processing of high-volume data in the network at line-rate, for instance in the domain of data analytics or stream processing, as we show later.

In this section, we briefly introduce the rationale of data plane programming by elaborating on the P4 programming language, before we discuss its implications redefining the relationship to related fields like Network Function Virtualization and the middlebox model.

2.6.1 The P4 Language

Like OpenFlow was considered the de-facto standard or even a synonym for SDN, P4 to date is considered the de-facto standard for data plane programming. P4 is a domain-specific language designed to allow programming of packet processing. The eponymically titled original proposal "Programming Protocol-Independent Packet Processors" [BDG+14] converged into an open-source language maintained by the non-profit P4 Language Consortium with a broad set of industrial contributors [Con]. P4's original specification called $P4_{14}$ [P4 18c] was released in 2014, as was the paper, and was succeeded by the $P4_{16}$ specification [P4 18d] in late 2016. Major extensions and differences between the two versions are well described in [BD17].

P4 was designed around a more general notion of packet processing than mere forwarding. It declares the following design goals which drastically extend the networking paradigm that was formerly limited by the capabilities of traditional and OF-enabled networking hardware (cf. Section 2.2.1).

(1) Reconfigurability: opposed to traditional networking hardware where the packet processing pipeline is tied to the underlying fixed-function ASIC, the processing semantics of P4-enabled networking hardware, called *P4 targets*, can be flexibly changed after their deployment through reconfiguration in the field by a controller.

(2) Protocol independence: P4 targets are not to rely on and be limited by fixed definitions of protocols in terms of packet formats and headers. Instead, a P4 program comprises the definition of packet parsers [GVHM13] that interpret an abstract bit-string representation of a packet and hence allow for the extraction of header fields and values on which later entries of match-action tables operate on.

(3) Target independence: The specification of a P4 program is to be agnostic of the specific type of target the program is later deployed on. The P4 compiler however includes a target-specific (back-end) part that is aware of the target's capabilities and transforms a target-independent description specified in the uniform P4 language to a target-specific program deployed to hardware. The P4 compiler chain and its target-dependencies are illustrated on the example of our P4CEP workflow later in Figure 5.5 on p. 149. A classification of P4 targets along with a discussion of challenges of hardware implementation is given later in Section 5.1.1.

Like OpenFlow, P4 implements a table abstraction for programming of network behaviour which is however much more powerful due to the aforementioned design principles. As illustrated in Figure 2.1(b) on p. 39, the ingress pipeline of a P4 target consists of a programmable parser and multiple stages of programmable match-action tables. The egress pipeline (undepicted) analogously consists of multiple tables and a deparser. A P4 program accordingly comprises (1) a specification of (multiple) parsers and their possible combination in a parse graph along with definitions of headers, (2) definitions of table structures (matching fields and applicable actions), and (3) a control program (*control flow*), determining the relative sequence of tables (stages) and their conditional execution, hence specifying specific paths through the pipeline that packets of specific flows follow. The logic in the control flow can incorporate and maintain two types of state. Intra-packet state, most notably intrinsic packet metadata such as ingress port, packet length, and timestamps and inter-packet state[3] in the form of registers, counters, and meters, where only the inter-packet state is persisted in stateful memory and hence retained across packets, enabling stateful packet processing.

The reconfiguration process consists of compiling a P4 program and deploying the P4 program through a programming API, consequently defining the pipeline constituents (1-3) as mentioned above. The OF-equivalent of flow programming is the population of the defined tables with table entries through a dedicated runtime API. Reconfiguration typically is a disruptive process ceasing packet processing while it is running, whereas the population of tables is a non-disruptive runtime process (just as with OpenFlow). Interestingly, P4 used to lack a uniform, target-independent runtime API, i.e., a southbound interface for populating tables, which OpenFlow implemented from the start. In 2018, the P4 Language Consortium released *P4 Runtime* [P4.19] as a unified interface for populating flow tables, which has been rapidly adopted by the Google Cloud Platform [WV18].

[3]Technically, tables constitute another form of inter-packet state memory, which can however only be modified by the control plane while being read-only for the data plane.

2.6.2 Embracing Middleboxing and Network Function Virtualization

In Network Function Virtualization (NFV), network functions (NFs) such as firewalls, NAT gateways, or load balancers, are flexibly moved from costly dedicated hardware middleboxes onto commodity server hardware using virtualization techniques.

Remote hardware is used for NF implementation in both the middlebox model, i.e., proprietary appliances on closed hardware, and in NFV, i.e., general-purpose hardware running virtualized software NF-implementations. While dedicated middleboxes are placed on or at least close to the path, virtualization hosts providing vNFs are typically remote, i.e., off the path. On the one hand, packet processing in software running on general-purpose hardware has become remarkably fast. On the other hand, *ex-situ* packet processing inherently requires to re-steer the traffic to traverse additional hardware entities—hence the name *middle*-boxes. By steering traffic through remote hardware, additional round trips are inherently inflicted, consequently increasing application latency. Thus, packets are ideally processed *in-situ* at high-performance network elements that they naturally traverse, consequently combining forwarding and processing. Furthermore, in modern networking, traffic typically has to traverse multiple (v)NFs (*service chaining*). Although NFV may mitigate chaining costs through consolidation of multiple VNFs into a single physical host, the traffic still has to traverse multiple software components (virtual switches, hypervisors, virtual NICs). Overall, significant latencies accrue depending on the physical or logical distance of (V)NFs and chain lengths. For WAN scenarios, the incurred latency can easily reach an order of tens to hundreds of milliseconds. Moreover, the probability of failures increases with increasing chain length.

Many network functions, such as load-balancing or firewalling, depend on exerting fine-grained control over network traffic and ultimately boil down to providing connectivity—or deliberately not providing connectivity—and thus to forwarding behavior which nowadays SDN is able to flexibly control. Even with traditional OpenFlow, complex network-centric appliances such as content-based routing can be substituted by in-situ packet processing directly on switches data plane, providing line-rate throughput [BTK+17] and eliminating the need for a remote middlebox or a virtualized network function (the *broker*). Recently, many approaches within NFV-related SDN have proposed *in-situ* middlebox replacements, i.e., the implementation and distribution of network functions onto the switch data plane. Generic frameworks are for instance OpenBox [BBHH16] and NetBricks [PHJ+16].

Data plane programming shifts this frontier even further, enabling pushing down more

complex network functions to switches. In the recent trend of *in-network computing*, NFs of middleboxes and application functionality from end-systems are offloaded to programmable network elements using data plane programming while leveraging the performance of specialized forwarding hardware. In Section 4.3.3, we discuss the relationship between processing in ZeroSDN with data plane programming and network function virtualization. In Chapter 5, we present a P4-based in-network implementation of Complex Event Processing as a representative of a more complex application from the domain of message-oriented middleware requiring stateful packet processing.

3 Update Consistency and Consistent Management for SDN-based Multicast

In Section 2.3.2, we described two classes of mechanisms to ensure correctness of network updates. While two-phase updates guarantee the maintenance of any property during an update, they are resource-intense in terms of flow rule memory, which is a scarce resource on hardware switches, and time, required for the installation and removal of parallel versions of flow rules. Update ordering approaches, on the other hand, are lightweight and already solve many problems for unicast route updates and even minimal procedures for some invariants like way-point enforcement and loop-freeness exist (see Section 2.3). However, there are no thorough investigations of update ordering approaches for the multicast paradigm, although in particular multicast could benefit very much from consistent updates to increase performance and user experience. As we show in this chapter, update consistency entails numerous crucial particularities that are to be considered in the management of multicast networks.

Multicast is a messaging pattern that implements efficient one-to-many and many-to-many communication (*group-communication*). On a broader sense, the multicast pattern basically comprises any communication where messages are at some place replicated in order to be sent to a group of receivers (but in differentiation to *broadcast* typically not to *all* receivers). Throughout this chapter, we consider this broader definition, which subsumes the well-known IP Multicast.

Multicast tremendously benefits from SDN [IKM14]. Opposed to the distributed creation and maintenance of multicast distribution trees, logical control centralization tremendously reduces complexity and allows for optimized routing, as has been shown in [ARTN13] and will be shown in this chapter. Membership management and control naturally benefit from centralization alike. Multiple classes of applications rely on multicast, including large-scale media streaming in WANs, like distribution of live television broadcasts using IPTV. To provide resiliency, both, data and services in data centers are

typically distributed and replicated, using e.g., Apache Hadoop or Infinispan, heavily relying on multicast. Furthermore, the replication of messages is a base mechanism for many other message-passing patterns like publish/subscribe (see Section 4.1) and is widely used in message-oriented middleware, for instance implementing Complex Event Processing (see Chapter 5).

In this chapter, we make the case for update consistency awareness (see Section 3.1) and perform an in-depth analysis for the concrete case of multicast networks. We propose an update ordering and hybrid approach that tackles the update problem of avoiding dropping and duplication of messages in multicast networks. The relevance of drops is obvious. For instance, in audio or video streams, dropped packets result in dropped frames, which may severely reduce the Quality of Experience (QoE). Duplicate messages waste bandwidth and might lead to bandwidth bottlenecks during updates, which again might degrade the QoE of the application (e.g., a video application not receiving sufficient bandwidth anymore). Moreover, duplicates might confuse the application if it is not prepared to handle them. As an overall implication of drops or duplicates, temporary degradation of network performance and thus application QoE degradation can be stated.

In detail, the contributions of this chapter are as follows:

First, we propose a generic system architecture for network management, incorporating knowledge about update consistency to allow for the selection of an appropriate update mechanism and its parameters.

Second, we specify a network update correctness property specific to multicast—*duplicate-freeness*—which has been formerly unconsidered in the context of network update consistency. In our extensive analysis, we then prove that in general it is impossible to avoid violation of the two invariants drop- and duplicate-freeness for arbitrary multicast network updates using a stateless approach.

Third, we identify and define necessary conditions on multicast tree updates, leading to undesired effects that possibly break invariants. We show that update ordering is a degree of freedom, resulting in maintenance of drop-freeness, while sacrificing duplicate-freeness and vice versa. This allows for an *update strategy* that avoids drops at the cost of duplicates and vice versa. Either behavior can be achieved by a deliberate selection of a respective strategy. We furthermore introduce the *loop-freeness* invariant, which our approach additionally maintains.

Fourth, we conduct a detailed analysis of packets traversing the network while an update is being applied and show that the update order as perceived by the packets may differ to the order the controller has initiated these updates, depending on their

propagation delay. We show the implications of this reordering and present a method to prevent it.

Fifth, we propose a generic multicast update procedure. We introduce the *path update algorithm*, which decomposes a global multicast network transition into update steps for which invariant maintenance can be guaranteed, leveraging the degrees of freedom identified in the analysis. We outline the involved algorithms and procedures that translate tree changes to SDN rule updates and executes these updates in a guaranteed order, maintaining desired invariants. In addition, we present a mechanism to mitigate update-caused duplications through in-network filtering. Furthermore, we present a novel alternative update approach as an optimized state-based approach that maintains arbitrary invariants, while rule space consumption, i.e., TCAM space, is minimized. Overall, we particularly highlight feedback of the prevailing network state and update situation to the network management, as well as its decisions about the approach selection and parameter determination.

The remainder of this chapter is organized as follows. In Section 3.1, a generic system architecture for network management, incorporating update consistency is presented. Henceforth, this scheme is applied to the concrete case of multicast networks, where update consistency is shown to be of particular relevance. Section 3.2 formally introduces the multicast model. Section 3.3 provides the problem statement and an in-depth analysis. Section 3.4 describes our flexible update approach for multicast trees utilizing update ordering, with optional duplicate filtering, or an optimized hybrid approach. To investigate the implications of inconsistency effects, in Section 3.5 their frequencies and impact for real WAN scenarios are evaluated, both, analytically and empirically, through direct measurement in the data plane under update. Section 3.6 states related work. Finally, conclusions are provided in Section 3.7.

3.1 Update Consistency In Network Management

Overall, SDN has been fostering the evolution of network management [KF13, AB14], to allow for complex management of heterogeneous network elements and possibly overlapping network functions. In this section, we describe a generic system architecture for SDN network management under awareness of inconsistency effects due to network updates.

Figure 3.1 gives an overview of the proposed system architecture and control flow. We build on a typical SDN architecture (see Section 2.1), where management and control

are strictly separated from the actual packet forwarding in the data plane. Specifically, our architecture embeds a dedicated control loop (see Section 2.3) that implements an adaptation mechanism providing network updates that are consistent with given consistency criteria and avoiding undesired inconsistency effects. We subsume the control and management plane in a logically centralized entity denoted as ***Network Manager*** (NM), which implements the network configuration logic and handles the communication with the switches to deploy the configuration. We assume the Network Manager to have global knowledge of the data plane. To achieve scalability, the NM might be transparently distributed. ***High Level Policies*** (HLPs) represent the definition of network functions as well as global network constraints (QoS, QoE), the data plane has to implement and adhere to. HLPs might be declaratively defined in a high-level network programming language, such as Frenetic [FHF+11, MRF+13b], or be concretely implemented as SDN controller modules. Multiple network functions, such as routing, traffic monitoring, load balancing, or multicast, may co-exist. Their composition [CKLS13] is handled by the Network Manager. Initially, a concrete configuration of a network function NF_i is derived from its HLP_i (planned configuration) and pushed to the data plane where it eventually becomes effective, such that the **effective configuration** (data plane) reflects the **planned configuration** (management plane).

In this chapter, we focus on *change management*: based on the current configuration, the NM reacts to **changes** in both, the management plane, such as changes of the HLP, and the data plane. Data plane changes may be **generic**, such as topology changes, and thus affecting all network functions, or **network function specific**, such as a host joining a multicast group. In the configuration control loop, change events are interpreted as disturbance and trigger a **reconfiguration process**, in which the NM in reaction adapts to the changed conditions. This consists of two steps: (1) A set of rule updates U that change the current planned configuration of the NM to a new, adapted configuration, has to be calculated. This is naturally highly network function specific. In this chapter, we show how incremental updates of multicast traffic distribution trees are calculated. (2) In order to implement the network state transition, the **update** is then to be applied to the data plane. Based on how this is done (**update method**), the update execution possibly leads to transient **inconsistency effects** that might also affect other NFs.

The selection of an **update method** and its **parameters** has severe implications not only on the type of occurring inconsistency effects, but also on their extent, the reconfiguration duration, and data plane resource consumption, i.e., switch rule space. Thus, the NM assesses different types of information, describing the prevailing update situation, in order

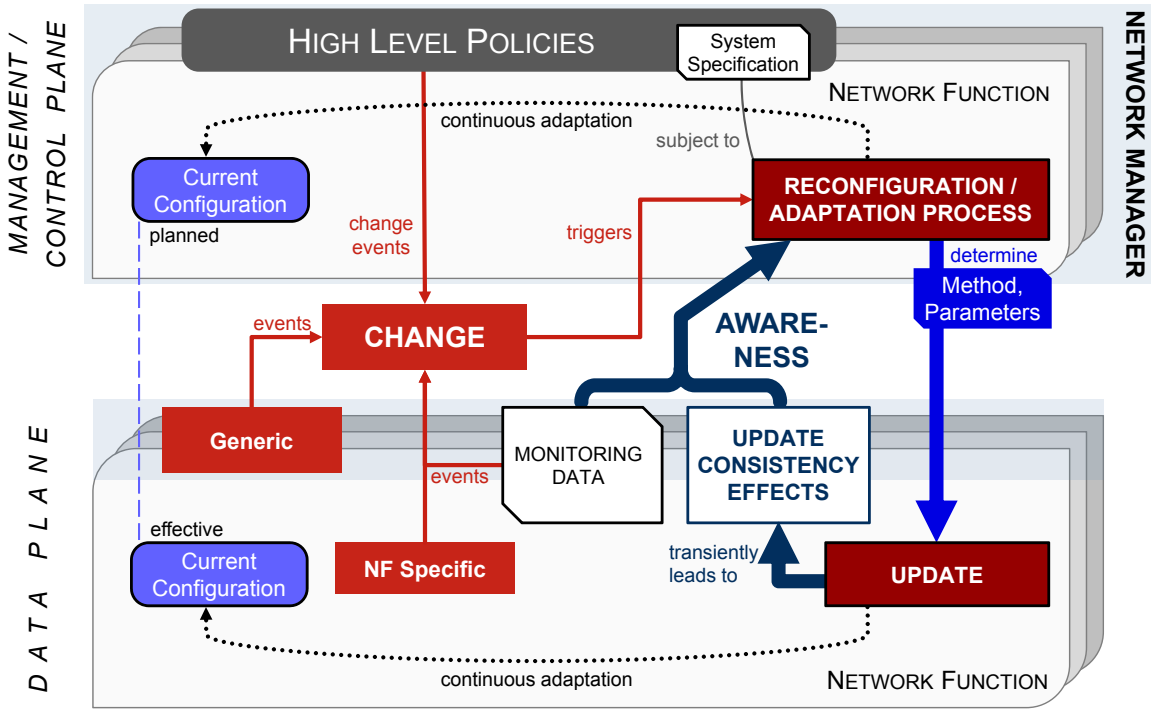


Figure 3.1. Overview of the proposed SDN-based system architecture for update consistency aware multicast networks.

to decide on an appropriate method and its parameters. This information includes expected update inconsistency effects, specific to the available update methods, characteristics of the triggering change event, the affected HLPs, and **monitoring data** from the data plane, such as statistics of NF-specific or generic flows. While the expected type of effects of an update method are known *a priori* to the NM, its extent can be estimated through a static analysis of U. The monitoring data, e.g., flow-associated packet rates [vADK14], even allows the NM to empirically estimate the expected number of affected packets. The NM is thus able to evaluate a method-parameter combination against the stipulated system specification or concrete HLP goals. This allows for a dynamic selection of an adequate update mechanism and determination of its parameters on the granularity of NFs and concrete updates. We leave the description of concrete algorithms for selection and quantitative parameter determination for later work, and rather focus on describing specific update methods.

3.2 Multicast Model

In this section, we describe preliminary assumptions and introduce multicast tree updates along with relevant consistency properties defining the problem statement.

Without loss of generality, we follow the IP Multicast Service Model [Dee89, Fen97]: there are possibly many senders and several receivers, denoted as *multicast group*. Logical addressing assigns a single class-D IP address to each group. Group messages are sent to respective destination multicast IP addresses over a distribution tree that defines the routing of the multicast traffic through the network. In our analysis, we focus on multicast traffic distribution and thus only consider switches in the distribution tree, not actual member end-systems (hosts). While irrelevant for the analysis, our approach assumes group management to be handled using the Internet Group Management Protocol (IGMP [Fen97]) as end-system-to-switch protocol.

We call a switch with connected group members (hosts belonging to group) a member switch. Routing is entirely done by SDN switches, which snoop IGMP signaling packets and report group membership to an SDN controller, enabling logically centralized group and tree management. We furthermore assume a network consisting of a set of SDN switches $sw \in SW$ (deviating from Section 2.1) with associated ports $p \in P$ connected over bidirectional links $l \in L$ with associated weight $w(l)$, forming a topology graph T. The distribution tree is denoted as a directed acyclic graph (DAG): $G(SW_{MC} \subseteq SW, L_{MC} \subseteq L)$. Switches are associated special roles as depicted in Figure 3.2 and listed below:

Table 3.1. Definition of switches roles in a multicast distribution tree

switch role	definition	cardinality
root / source	$S = \{s\} \subset SW_{MC} : deg_{in}(s) = 0$	$\lvert S\rvert = 1$
group members	$m \in M \subseteq SW_{MC}$	$\lvert M\rvert \geq 2$
relays (single out-port)	$rel \in Rel \subseteq SW_{MC} : deg_{out}(rel) = 1$	*
replicators (multiple out-ports)	$r \in Rep \subseteq SW_{MC} : deg_{out}(r) > 1$	*
non-tree switches	$SW \setminus SW_{MC}$	*

Throughout this chapter, we use the terms switch and node as well as link and edge interchangeably, depending on the current focus on either the networking aspect or its graph-theoretical representation. Packets are intentionally *replicated* and sent out on multiple links of *replicator* switches. We differentiate between relays and replicators, since replicators are shown to play a distinguished role for update consistency, as we will

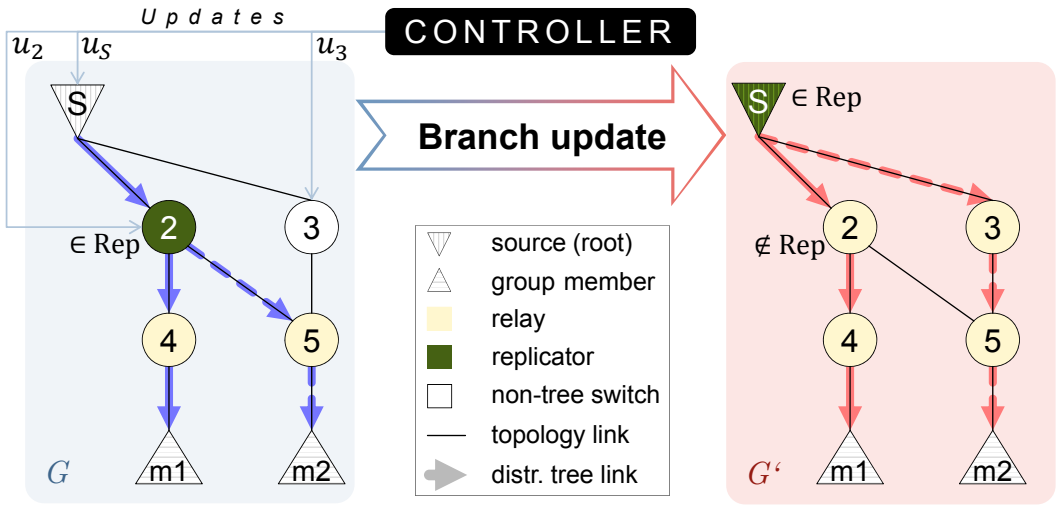


Figure 3.2. Update of a branch $s \rightarrow m_2$ (dashed lines) within a multicast distribution tree (G), leading to a new network state, i.e., distribution tree (G').

describe in the analysis. Note that group members, i.e., switches with connected member hosts, may also relay or replicate, they are not necessarily leaves of the tree. For ease of illustration, we assume a single multicast tree, which either might be a source-based tree or shared tree. We consider only group traffic of one multicast group, i.e., one destination IP address. This simplification does not limit the generality of our approach. It is valid for multiple groups and thus multiple distribution trees as well.

3.3 Problem Statement and Analysis

In this section, we first give the problem statement (Section 3.3.1). Next, we state the impossibility of combined drop- and duplicate-freeness (Section 3.3.2). We then refine conditions on updates which lead to violation of invariants (Section 3.3.3). We introduce a central structure for both the analysis and approach (Section 3.3.4). We then introduce the *loop-freeness* property (Section 3.3.5). Finally, we conduct an analysis of the update order as seen by packets being in the network during an update (Section 3.3.6).

3.3.1 Problem Statement: Multicast Tree Updates

Applying network updates to a multicast network, a global network state translates to a distribution tree instance. We assume distribution tree calculation to be a non-incremental process: in reaction to a topology or membership change, a tree is computed entirely

anew, irrespective of the extent of the actual change that triggers that recalculation. Thus, even small changes may result in huge differences in the recalculated tree. Our goal is to advance from an old distribution tree (G) to a new distribution tree (G') by finding an update order, while maintaining certain invariants (Π). More specifically, we want to guarantee drop-freeness and duplicate-freeness, where the latter informally describes the reception of an unintended duplicate.

Note that we exclude the actual calculation of the distribution tree from the problem scope. We assume the distribution tree as a minimum Steiner tree [HHLY14]. The *Steiner tree problem* asks for a tree, spanning a set of terminals ($V_T = S \cup M$) at minimum cost ($\sum_{l \in L_{MC}} w(l)$, where $w(l)$ is given by a cost metric such as bandwidth or latency), possibly including additional non-terminal elements, called Steiner nodes (V_{St}). Figure 3.11 shows an overlay Steiner tree on top of a WAN topology. This variant of the Steiner tree problem has been recognized as NP-hard within Karp's original 21 NP-complete problems [Kar72]. However, approximations with polynomial runtime complexity [CGSW14] as used in our evaluation (Zelikovsky's $^{11}/_{6}$-approximation [Zel93] with $O(|SW| \times |L_{MC}| + |V_T|^4)$) exist.

3.3.2 Impossibility Result

Claim: It is impossible to avoid violation of the two simultaneous invariants drop-freeness and duplicate-freeness for arbitrary transitions using a stateless update method.
Proof: We assume that maintaining both invariants at any time *was* possible, i.e., there exists exactly one effective path from s to each m_i at any time. Consider a scenario as shown in Figure 3.2, where a transformation from G (left) to G' (right), both correct multicast trees, is performed. In G, packets from a source s are sent to a replicator switch 2, which replicates the packets and forwards them to relay switches 4 and 5, where the respective replicas are forwarded to member m_1 and member m_2 respectively. In G', s takes over the replication and forwards to 2 and 3, while 2 only forwards to 4 and thus does not replicate anymore. To implement this transformation, the output port list of switches s, 2 and 3 have to be updated by rule updates u_i for switch i as presented in Table 3.2 and described in the following. To improve readability, we denote updates that add out-ports, i.e., add new path segments, as u_i^+, whereas updates that remove out-ports, i.e., remove existing path segments, as u_i^-.

Path $s \rightarrow 5$ has to be installed (u_s^+, u_3^+), whereas path $2 \rightarrow 5$ has to be removed (u_2^-). This implies the "shift" of the replication from 2 to s. The execution order of these updates is crucial: Figure 3.3 shows intermediate states $\lambda_1^{\sim}$, $\lambda_2^{\sim}$ of two update order

Table 3.2. Port change-set representation of a branch update

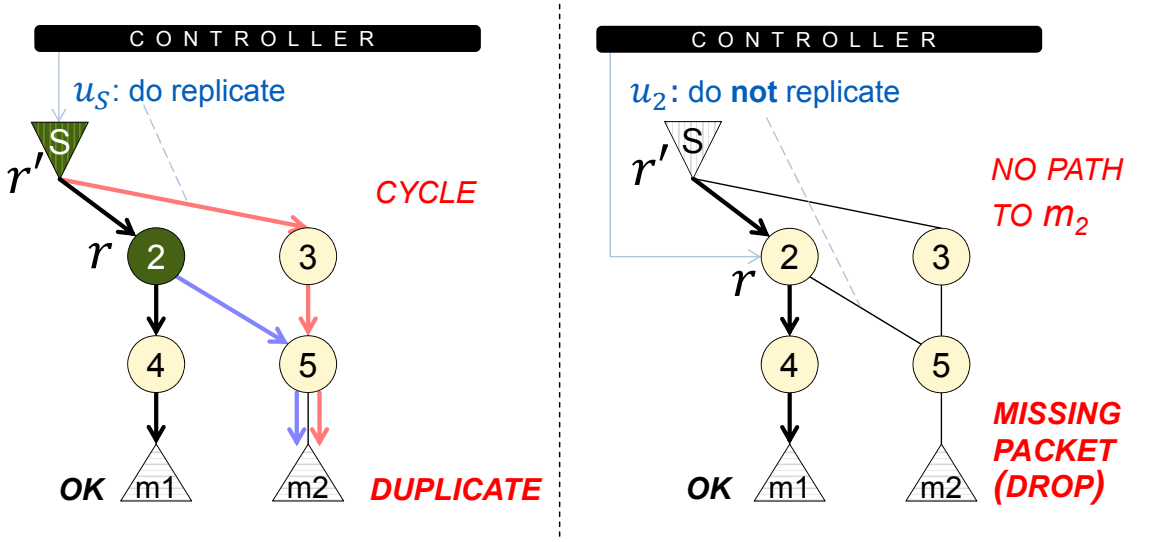
Switch i	Out-port set		
	old (PO_i)	new (PO'_i)	update
s	{2}	{2,3}	$u_s^+ = PO_s \cup \{3\}$
2	{4, 5}	{4}	$u_2^- = PO_2 \setminus \{5\}$
3	∅	{5}	$u_3^+ = PO_3 \cup \{5\}$

permutations, where u_3^+ and u_s^+ ($\lambda_1^\sim$) and u_3^+ and u_2^- ($\lambda_2^\sim$) have been executed. In other words, path $3 \rightarrow 5$ has been installed first, followed by an update of the new replicator r' to do replication in $\lambda_1^\sim$ and, respectively, followed by an update of the old replicator r to stop replication in $\lambda_2^\sim$. We use the *happens-before* relation [Lam78] to express an order of events: event e_1 happens before event e_2 iff $e_1 \lhd e_2$. We do not consider latency aspects for now, hence we subsume in an update event u_i the sending of an update message from the controller as well as the reception and the execution at a switch i. A detailed analysis including additional timing aspects, such as propagation delay, is given in Section 3.3.6. Formally, those two cases depict intermediate states of update orders $\lambda_1^\sim : u_3^+ \lhd u_s^+ (\lhd u_2^-)$ and $\lambda_2^\sim : u_3^+ \lhd u_2^- (\lhd u_s^+)$, where the respective last update has not yet been executed.

Obviously, a cycle has been introduced in $\lambda_1^\sim$ through a new effective path $s \rightarrow m_2$. A packet p entering the network at s, denoted as event $switch(p,s)$, at that point in time will get replicated twice and follow both paths which results in two replicas reaching m_2, which we call a *duplicate* at m_2. In $\lambda_2^\sim$, with $u_3^+ \lhd u_2^- \lhd switch(p,s)$, neither s nor 2 are replicating. Hence, there is no effective path $s \rightarrow m_2$ at all, resulting in a missing packet at m_2. Note that for $\lambda_1^\sim$ it does not matter whether u_3^+ is executed before or after u_s^+. Both orders, $u_3^+ \lhd u_s^+ (\lhd u_2^-)$ and $u_s^+ \lhd u_3^+ (\lhd u_2^-)$, result in an intermediate state with multiple effective paths to m_2. Analogously, there exists an equivalence class of update orders causing drops due to an intermediate state with no effective path to m_2: $u_3^+ \lhd u_2^- (\lhd u_s^+)$, $u_2^- \lhd u_3^+ (\lhd u_s^+)$, $u_s^+ \lhd u_2^- (\lhd u_3^+)$, and $u_2^- \lhd u_s^+ (\lhd u_3^+)$. Hence, in any of all six possible update order permutations, either *duplicate-freeness* or *drop-freeness* is violated which contradicts our assumption. We thus have proven that there exists no correct update procedure w.r.t. both, drop- and duplicate-freeness. □

3.3.3 Conditions for Violation of Invariants

Although we have proven that in general we cannot guarantee both desired properties at the same time, we describe the conditions for the violation in the following.



(a) $\lambda_1^{\sim}$: $u_3^+ \lhd u_s^+ \lhd switch(p,s)$ (b) $\lambda_2^{\sim}$: $u_3^+ \lhd u_2^- \lhd switch(p,s)$

Figure 3.3. Traces of packets (arrows), entering the network in two intra-update states $\lambda_1^{\sim}$ and $\lambda_2^{\sim}$, where u_3^+ and u_s^+ (a) and u_3^+ and u_2^- (b) have been executed, resulting in duplicates and drops, respectively.

***Branch update, replicator pair*:** We refer to the kind of update as shown above, where there is a change in the path from s to some branch member, as *branch update*. We define a *replicator pair* (r, r') as a pair of old and new replicator, where $r \in G$, $r' \in G'$. In the example, r is switch 2 and r' is s. Note that, as described, postponing u_3^+ in $\lambda_1^{\sim}$, i.e., $u_s^+ \lhd u_3^+ (\lhd u_2^-)$, does not solve but only defers the problem: as long as $3 \rightarrow 5$ is missing, duplicates are stopped at 3, however, eventually, u_3^+ and u_2^- have to be executed. Anyway, pushing a critical update, i.e., that update that finally establishes a new path and thus would cause duplicates, from a replicator downstreams along unicast paths constitutes an additional degree of freedom.

This reveals the reason for the impossibility result: replicator updates inherently involve updating a pair of distinct switches, affecting common subsequent switches downstreams of them. Ignoring m_1, this transition would be a trivial unicast path update, were after $3 \rightarrow 5$ has been installed, the single switch, s, would be updated through a single non-competing update to forward to 3, instead of 2. In our multicast scenario, this is not feasible, since 2 still has to forward packets downstream via 2 to m_2 and thus has to receive packets from s.

***Dependency of replicators*:** In the depicted type of replicator change, both, r and r' of a replicator pair are common elements of at least one path from s to all $m \in M$. Hence,

there exists a dependency among (r, r') in forwarding packets on such paths. In such cases, we denote replicators of a pair to be *dependent*. We refine this definition in the following.

***Replicator move (downstream/upstream)*:** If a dependency among (r, r') is present, we call the two associated updates $(u_r, u_{r'})$ a *replicator move*. A replicator move is directed. It is denoted an *upstream move* if r' is upstreams of r, i.e., r' is a (transitive) predecessor of r on at least one path from s to all m. In this case, as in the example, r is dependent on r'. This dependency stems from the dependency of packets, being forwarded on such a path: The events of r receiving a packet p from r' ($e_{r,rec}$) and subsequently sending p further downstreams ($e_{r,snd}$) are causally dependent on the event of r' sending a message to r ($e_{r',snd}$). Thus: $e_{r',snd} \lhd e_{r,rec} \lhd e_{r,snd}$. Vice versa, if r is a predecessor of r', it is denoted a *downstream move*. A minimal illustration of both, a downstream and an upstream move is show in Figure 3.6 on p. 76.

***Replicator swap*:** If replicators are not dependent, we call the update a *replicator swap*.

In conclusion, we state that simultaneous drop- and duplicate-freeness is not possible for updates that involve a change of a replicator pair (r, r'). However, depending on the update ordering, one of the inconsistency effects, either drops or duplicates, can be prevented.

This degree of freedom can be leveraged by a deliberate selection of an inconsistency effect that is tolerable, while the other one is prevented. We denote the exploitation of this particular degree of freedom within multicast network update as ***update strategy***. As explained in the introduction, it depends on the concrete application that uses multicast whether violating one property is preferred over violating the other. In general, this decision is to be made by the network manager. Henceforth, we assume drops more fatal than duplicates, and thus, without loss of generality, argue from this perspective.

3.3.4 Central Analysis Structure: The Delta Graph

In the following, we introduce a central structure for both, our analysis and approach. As seen in the examples so far, inconsistency effects due to replicator pair updates do not necessarily affect all members. Furthermore, a network update might consist of several replicator pair updates as well as of uncritical updates. To be able to identify relevant nodes of a network update as well as to identify affected nodes of individual replicator moves, we introduce the *delta graph* (G^{Δ}). It captures differences of two distribution trees G and G'. Informally, it can be constructed by merging G and G', removing all common

edges, followed by removing all unconnected vertices. Formally, we define (def.) G^{Δ} as follows:

Def. G^{Δ}**:** $G^{\Delta} = \left[(G - L'_{MC}) + (G' - L_{MC})\right] - SW^0_{MC}$, where L_{MC} is the set of links and SW_{MC} is the set of switches of G. Primes indicate reference to G'. SW^0_{MC} denotes unconnected nodes, i.e., $\forall sw \in SW^0_{MC} : deg_{in}(sw) = deg_{out}(sw) = 0$.

Figure 3.4 illustrates an intermediate step after merging exemplary G and G', where ellipses denote unicast paths of arbitrary length, to provide generalization. Thin, blue edges (bottom paths) are exclusively in G and thus represent old paths, to be removed as part of a network update, whereas thick, red edges (top paths) are exclusively in G' and thus represent new paths, to be installed respectively. Common edges (dashed), both in G and G', and respective nodes (dashed) are not to be changed and thus removed from G^{Δ} in a subsequent step. We further define a set of *join nodes* $N^{>}$ as follows:

Def. ***join node*****:** $\forall j \in N^{>} \subset SW^{\Delta}_{MC} : deg_{in}(j) = 2$. Each j is associated with a replicator move. The effects of a corresponding replicator move affect all downstream nodes of j.

Def. P**,** P'**,** ***split node*** $s^{<}$**:** We further denote the old path from s to j in G as $s \xrightarrow{P} j$, and the new path from s to j in G' as $s \xrightarrow{P'} j$, analogously. To identify a replicator pair (r, r'), we back-traverse P and P' in G^{Δ}, starting from j, until no further predecessor exists or a common predecessor in both P and P', denoted as split node $s^{<}$, is reached. The end nodes on P and P' define r and r', respectively. If $s^{<}$ exists, a non-competing update, which is performed by an update of a single node, $s^{<} = r = r'$, is present. Non-competing updates arise for instance when a single edge, connecting $s^{<}$ and j is replaced by a path $s^{<} \xrightarrow{P'} j$. Since they are not critical in terms of update consistency, we do not consider them to be replicator updates. In the exemplary G^{Δ} in Figure 3.5, four join nodes j_i define replicator moves with respective replicator pairs (r_i, r'_i) and paths (P_i, P'_i).

3.3.5 Maintaining Loop-freeness

A special update case that has shown to occur very frequently in our evaluation scenarios arises, when old and new paths are interleaved, leading to *swap paths*, i.e., edges both in G and G' but with opposite direction, as illustrated in Figure 3.5. Consider the following naive but drop-freeness-maintaining update: after processing of $j_1 = 5$ (install $4 \xrightarrow{P'_1} 5$, remove $3 \xrightarrow{P_1} 5$), a transient loop $3 \to 4 \to 5 \to 3$ is introduced during processing of $j_2 = 3$, while $5 \xrightarrow{P'_2} 3$ has been installed and $3 \xrightarrow{P_2} 4$ has not yet been removed. Note that if duplicate-freeness is to be maintained, i.e., P_i is removed before P'_i is installed, naturally, loops and duplicates do not appear. As the occurrence of loops is dependent on

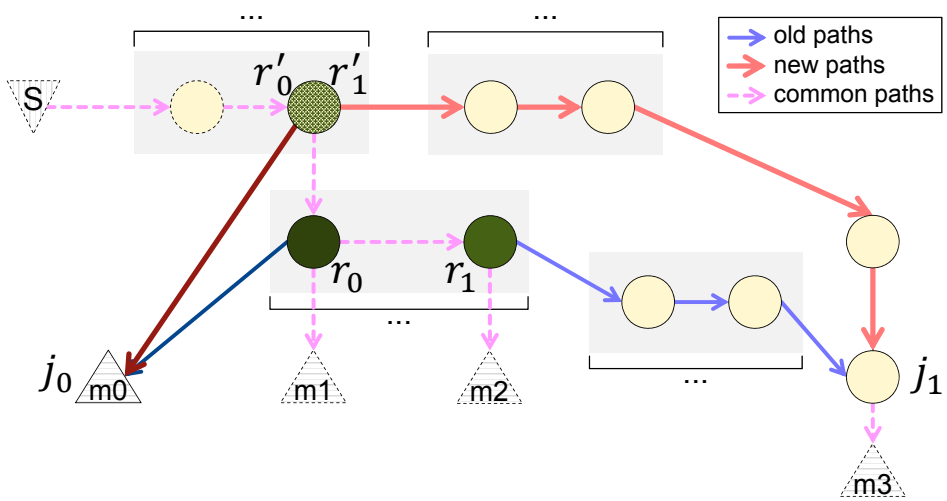


Figure 3.4. Intermediate step of G^{Δ}-construction, after exemplary G, G' have been merged. Dashed edges and vertices are removed in a subsequent step.

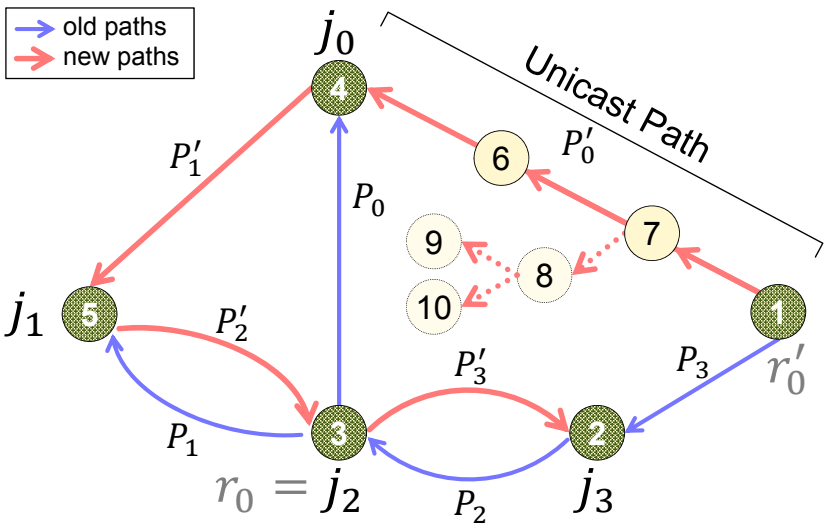


Figure 3.5. Traversal of an exemplary delta graph (different from Figure 3.4), leading to a correct ordering among the updates. The dotted subtree represents joining group members. Replicator pairs (grey labels) other than (r_0, r'_0) are omitted for brevity.

the update strategy, we define a third correctness property, *loop-freeness*, which is relevant for competing replicator move updates when drop-freeness is to be guaranteed. Avoiding cycles can however be achieved through a suitable update ordering. We derive a general update ordering, maintaining also loop-freeness in Section 3.4.

3.3.6 Effective Update Order

While we have focused on the formal structural analysis of multicast network updates so far, in the following, we incorporate further aspects of time to complement the prior analysis. In this section, we describe the situation for packets traversing the network while an update is being applied. We show that propagation delay of group messages may lead to an inversion of inconsistency effects and thus has to be included in the approach in order to be able to guarantee an invariant as selected by the update strategy. On the other hand, it can be used to mitigate inconsistency effects.

3.3.6.1 Analysis

We differentiate between two types of events: (1) Events due to the packet forwarding process in the data plane: message receipt (packet ingress) and processing (forwarding), which possibly leads to multiple message sending. We consider processing latencies negligible compared to link propagation delay and thus only consider the latter in this analysis. Therefore, we subsume all named sub-events in a single *switch event*. (2) Events due to update messages from the controller: update message receipt and processing (execution). We assume that the network manager is aware of the control channel latencies and handles the timing of control messages accordingly. We continue discussing this assumption at the end of this subsection. We depict *update event*s to mark the end of the update execution, i.e., when the update has become effective on the data plane.

For ease of demonstration, we simplify the example from Figure 3.2 by omitting nodes which are solely relay nodes in both G and G', i.e., switches 3 to 5, as shown in Figure 3.6.

We start with the upstream replicator move (see Figure 3.6 with $r = s$ and $r' = 2$), where r is a successor of r' and thus dependent on r', as in our running example. We assume the drop prevention update strategy, at the cost of duplicates, and thus an update order of $u^+_{r'} \lhd u^-_r$. Figure 3.7(a) shows a space-time diagram, depicting packet traversal with varying propagation delays and in intermediate states of an upstream move. A time axis for each node is given, where arrows depict events. A packet *pk* is entering the network at the source node s and is forwarded on ingress by each switch i in a switch

event $switch(pk, i)$ according to the rules installed at i at the time of the switch event. The traversal of each packet is captured by a respective trace, reflecting the history of update events as seen by the traversing packet. As shown in Figure 3.7(a), packets always reach m_1 correctly, i.e., exactly once, such that we do not include them in traces (dotted arrows).

Packet and trace names are binary coded according to the state that was present at network ingress or switch events, respectively. For instance, packet pk_{11} (rightmost) enters the network after all updates have already been executed and become effective (irrespective of their order): $\cdots \lhd switch(pk_{11}, r')$. Packet pk_{11} results in a correct trace tr_{11} (rightmost), since all switch events happen after the update of the respective switches. A packet pk_{00} is entering the network before u_r^- and $u_{r'}^+$ and is thus switched at the ingress switch $s = r'$ before $u_{r'}^+$: $switch(pk_{00}, r') \lhd u_{r'}^+$. Depending on the propagation delay of $r' \rightarrow r$, denoted as $T_{r',r}$, packet pk_{00} might reach r before its update or in an intermediate state after its update. In the following, we assume two cases of a small and large $T_{r',r}$ respectively. The first case, $switch(pk_{00}, r') \lhd switch(pk_{00}, r) \lhd \cdots$, results in a correct trace tr_{00} (leftmost trace). In the second case, $switch(pk_{00}, r') \lhd u_r^- \lhd switch(pk_{00}, r)$, however, r' did not yet replicate, whereas r has, due to its update, already stopped replicating, resulting in a drop at m_2 in the trace tr_{01}. Due to propagation delay, the effective update order, i.e., the update order as experienced by the traversing packet, is inverted from $u_{r'}^+ \lhd u_r^-$ to $u_r^- \lhd u_{r'}^+$.

As shown in the motivating example, a packet ingress in an intra-update state, i.e., within the volatile phase (T_{vol}), pk_{10} with $u_{r'}^+ \lhd switch(pk_{10}, r') \lhd switch(pk_{10}, r) \lhd \cdots$, results in a duplicate (tr_{10}). However, analogously, through propagation delay, pk_{10} might see a different update order, $u_{r'}^+ \lhd switch(pk_{10}, r') \lhd u_r^- \lhd switch(pk_{10}, r)$, which leads to a correct trace tr_{11}, although the packet ingress has happened at an intra-update state.

For upstream replicator moves with an update order of $u_r^- \lhd u_{r'}^+$ (see Figure 3.7(b)), inversion through propagation delay cannot happen, since the dependent replicator r is updated before its predecessor r' and thus, irrespective of $T_{r',r}$, tr_{10} cannot occur. Thus, pk_{00} either leads to tr_{00} (correct) or tr_{01} (drop), where pk_{01} necessarily leads to tr_{01}.

In the case of downstream moves, where r' is a successor of and thus dependent on r, the situation is inverted. Downstream moves with $u_r^- \lhd u_{r'}^+$, i.e., the update strategy to prevent duplicates at the cost of drops, pk_{10} may lead to drops, as can be verified in Figure 3.7(c). Through propagation delay, this effect may be inverted, such that duplicates instead of drops occur. Analogously, downstream moves with $u_{r'}^+ \lhd u_r^-$, (not shown) may result in duplicates in case of intermediate states, also due to propagation delay.

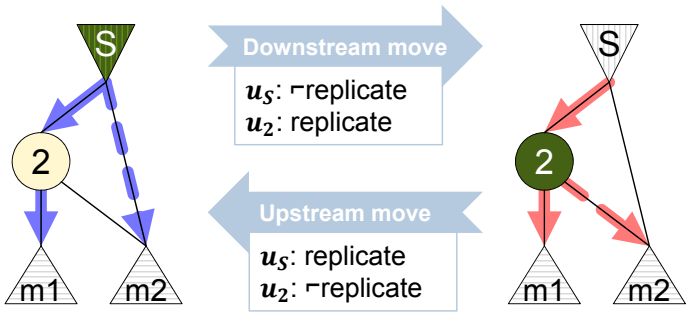
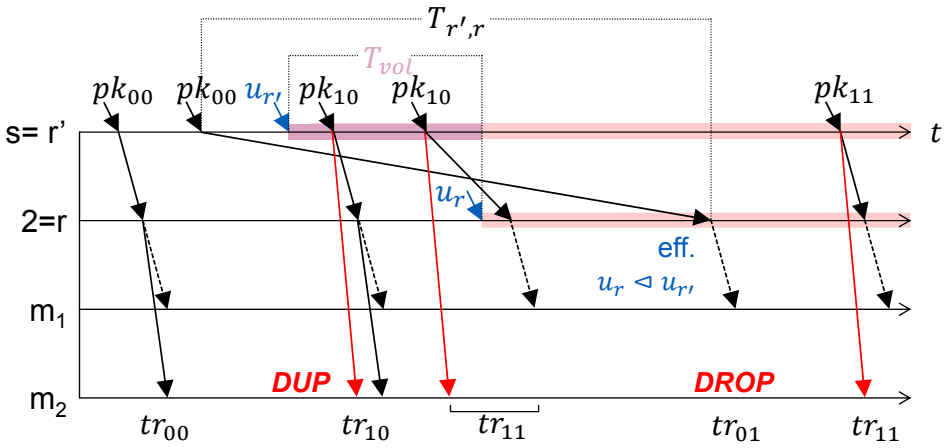


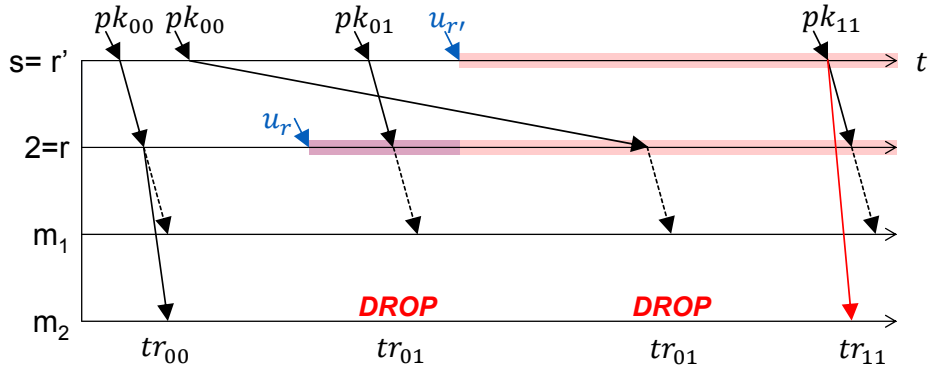
Figure 3.6. Simplified example for the analysis of intermediate states, showing downstream and upstream replicator moves and corresponding updates.

We thus conclude: in general, two conditions for perceived update reordering through propagation delay between (r, r') can be stated: dependency among (r, r') and the upstream node being updated before the downstream node. Due to the independence of replicator swaps, they are not prone to reordering. Replicator moves present dependency but only two out of four possible cases, those fulfilling the second condition, are prone to reordering: upstream moves with $u^+_{r'} \lhd u^-_r$, possibly leading to drops through propagation delay, instead of duplicates, and downstream moves with $u^-_r \lhd u^+_{r'}$, possibly leading to duplicates through propagation delay, instead of drops.

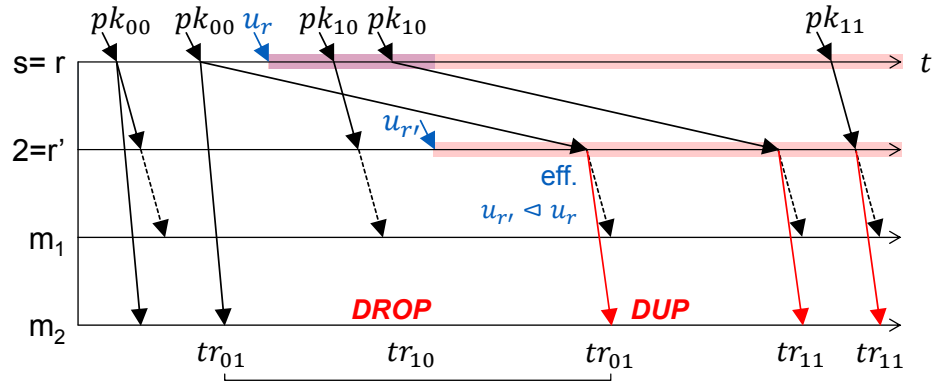
Effective update reordering due to propagation delay obviously counters the update strategy. However, first, typically the network manager would select one static update strategy for the whole multicast group traffic. Thus, there would only be one case left where drop-freeness cannot be fully guaranteed. Second, typical flow modification processing delay of OpenFlow-enabled hardware switches [RSU+12, HYS13] are at least one order of magnitude larger than typical one-hop latencies of LAN or WAN links. Anyway, since we enable the network manager to be aware of this effect and its conditions, it is able to estimate the effect's extent and evaluate its criticality. The network manager has global knowledge of the topology and the changes to be applied to the multicast network, including the type of replicator change. This allows for a static analysis of the effect's extent (see Section 3.5.1). Along with the empirical measurement of data rates, the network manager can even estimate the expected number of drops (see Section 3.5.2). Based on this evaluation, the network manager may either simply choose to tolerate the reordering or apply a method to eliminate effective update reordering, which we present in the following.



(a) Upstream replicator move with $u^+_{r'} \lhd u^-_r$



(b) Upstream replicator move with $u^-_r \lhd u^+_{r'}$



(c) Downstream replicator move with $u^-_r \lhd u^+_{r'}$

Figure 3.7. Intermediate states of replicator moves as perceived by traversing packets, considering propagation delay, which may invert the effects.

3.3.6.2 Mitigation approach

In the analysis, we have identified a crucial measure for the reordering: the propagation delay between the replicators, $T_{r',r}$, which, in a real-world network, would typically include multiple hops and might thus accumulate to the order of tens to hundreds of milliseconds for WANs. The network manager's global knowledge also allows for a measurement of path latencies and thus the determination of $T_{r',r}$.

To eliminate effective update reordering, for cases where these would occur, the update of the downstream replicator is artificially delayed by $max(T_{r',r}) + t_s$, where t_s denotes a safety margin which may possibly be added to handle outliers of $T_{r',r}$. For instance, in the example of Figure 3.7(a), this guarantees that the last packet pk_{00} with ingress before $u^+_{r'}$ does not reach a yet updated r, such that drops are prevented. On the downside, however, this approach increases the volatile phase T_{vol}. Consequently, it constitutes a trade-off between potentially increasing the extent of tolerable effects through decreasing the extent of undesired effects. Furthermore, it is prone to jitter, i.e., variation of link delay. While the jitter in LANs is typically small enough to be safely ignored here, in WANs it might have to be considered in the determination of $T_{r',r}$. With small jitter, however, the extent of tolerable effects is small, since packets entering the network within the volatile phase (pk_{10}), would, through the artificial delay and small propagation jitter, most probably reach a yet updated downstream replicator, leading to a drop and duplicate free trace (tr_{11}).

An optimization of this delay-based method is to minimize T_{vol} through the incorporation of the network manager's knowledge about the update rates of the involved SDN switches, similarly to [JLG+14]. This would allow creating exactly timed update schedules, which could be precisely executed by the network manager, utilizing the timed network updates approach [MSM16]. The timed update approach relies on sufficiently synchronized switch clocks, which is a reasonable assumption for both, LANs and WANs. On modern hardware using data plane programming, a synchronization error < 50 ns (99th percentile) can be achieved over 4 hops in a LAN with heavy network load [KJC19]. For WAN scenarios, GPS-based synchronization, with errors < 200 ns according to [ubA] and own measurements, can be used. Since we enable awareness of the network manager also for the stated trade-off and dependencies, it is able to reason and decide on a concrete method, including the determination of parameters and optimizations. In Section 3.4.4, we present another update approach which guarantees the maintenance of arbitrary invariants by combining stateless updates with optimized state-based updates to minimize the rule space consumption.

3.4 Flexible Approach for Multicast Tree Updates

Next, we present a flexible update approach for multicast distribution trees that feeds back the prevailing update situation to the network manager, which dynamically decides on an update mechanism to be used. In Sections 3.4.1 to 3.4.2 we describe a stateless update mechanism which allows for the selection of one primary invariant to be guaranteed (*update strategy*), where optional duplicate filtering (Section 3.4.3) may be applied. In Section 3.4.4 we present a hybrid, i.e., both, stateless approach and state-based approach, which maintains arbitrary invariants, while rule space consumption is minimized. An overview of the proposed update procedure is given in Figure 3.8, relevant notations for this section are listed in the following:

Table 3.3. Table of relevant notations for Section 3.4

sw	switch (node)	$s_i^{<}$	split node i	l	link
s	source node	$G^{(\prime)}$	distribution tree	pk	packet
j_i	join node i	G^{Δ}	delta graph	tr	packet trace

$r_i^{(\prime)};\ (r_i, r_i')$	replicator node $\in G^{(\prime)}$, associated with j_i; r. pair
$P^{(\prime)}, sw_1 \xrightarrow{P^{(\prime)}} sw_2$	path in $G^{(\prime)}$, from sw_1 to sw_2
$L^- \,/\, L^+$	paths to be removed/installed in one update step
$u_{sw}^- \,/\, u_{sw}^+$	rule update (removal/installation) at sw
T_{sw_1,sw_2}	propagation delay between sw_1, sw_2

The update procedure comprises five steps, as presented in Figure 3.8:

S1) We capture differences in the distribution tree, which was recalculated due to events in the network, such as topology changes, e.g., link and node failures or utilization changes, as well as changes in the group membership due to joining or leaving nodes (churn). The ***change analyzer*** constructs the delta graph (G^{Δ}), representing all changes.

S2) The *change analyzer* returns all join nodes, their update type, i.e., (up-/downstream) replicator move, replicator swap, non-competing, as well as the number and identity of affected nodes to the network manager's **reconfiguration process**. There, an appropriate update mechanism is selected along with its parameters, based on the prevailing update situation.

S3) Based on the selected mechanism and parameters, the necessary data plane updates are calculated by the ***path update algorithm*** that decomposes the tree changes into incremental edge updates. To this end, it traverses G^{Δ} in order to identify and classify all changes into branch updates, replicator updates, and added and removed edges due to

member dynamics. Then it defines update sequences by building a partial ordering over sets of edge updates, associated with an update type. The order depends on the given update strategy.

S4) These sets of edge updates are translated into SDN rule updates by the ***rule update generator***.

S5) Lastly, the updates are applied to the data plane in guaranteed order by the ***update executor***, which executes all updates based on the update mechanism, selected by the network manager.

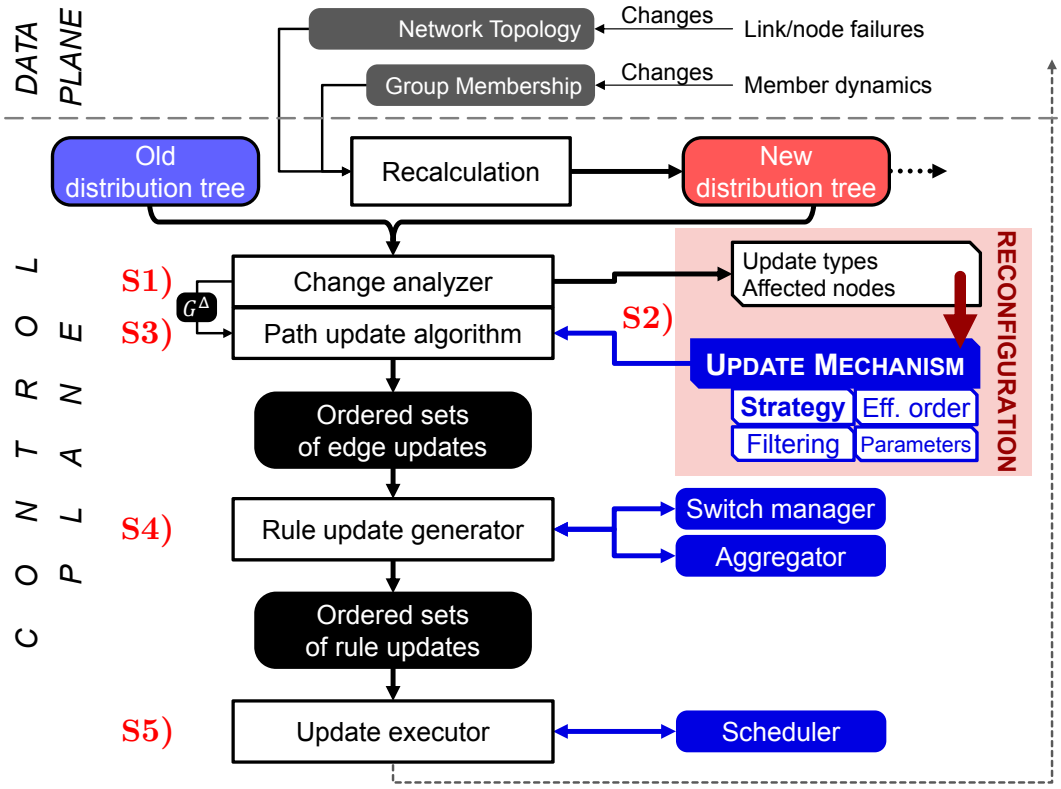


Figure 3.8. Overview of the proposed multicast update procedure, where the specification of the update mechanism to be used is determined in the network manager's reconfiguration process.

3.4.1 Processing Graph Changes (S1 to S3)

The ***change analyzer*** creates G^{Δ} (S1), which contains all changed edges of a pair of old and new distribution tree (G, G'), reflecting their transition. Thus, every edge l is associated with either G or G'. An edge l is in G^{Δ} iff $l^- \in G \wedge l^- \notin G'$ or $l^+ \in G' \wedge l^+ \notin G$.

Within the transition, all $l^- \in L^-$ are to be removed and all $l^+ \in L^+$ are to be installed.

3-phase G^Δ-decomposition: Through constructive traversal, the ***path update algorithm*** decomposes G^Δ into path segments (S3), i.e., ordered sets of edges, and defines an ordering of these steps. One step contains maximum-length path segments, while still maintaining invariants, e.g., unicast paths of arbitrary length as depicted in Figure 3.5 (running example) can be combined in one update. We refer to paths in G consisting of l^- as old paths and vice versa for new paths. An update step is defined as a pair (L^-, L^+). On the one hand, the execution order within one update pair is determined by the update strategy: in general, if drops are to be prevented, edges are installed before edges are removed and vice versa, when preventing duplicates. On the other hand, the order among update pairs is crucial: intuitively, the general "remove-before-add" procedure implies careful removal, such that only edges that are independent, i.e., not needed by downstream nodes, are removed. This is, e.g., to verify, that an old path $sw_1 \xrightarrow{P} sw_2$ is removed only when it either has been replaced by a new path $sw_1 \xrightarrow{P'} sw_2$ or $sw_2 \notin G'$. The first case would be caused by a replicator move, the second case would be caused by a leaving member or a relay node in G becoming a non-tree node. To ensure this behavior, i.e., to determine a proper inter update pair order, we traverse G^Δ, starting at s.

Phase 1 - Determining join node sequence: In the first traversal step, we employ a depth-first search (DFS) to obtain an ordered list of all join nodes $j_i \in G^\Delta$ (indices depict sequence in Figure 3.5), to be processed in order later. Note that s is not necessarily an element of G^Δ and furthermore, G^Δ might be a forest. We thus traverse G' starting from the closest (most upstream) node of s for each possibly isolated tree of G^Δ which is in G' (node 1 (rightmost) in Figure 3.5).

Phase 2 - Join node processing: Then, join nodes are processed in order, where for each j_i, the number of affected nodes is assessed, as is a pair (L_i^-, L_i^+). The pairwise update step is determined by a backward DFS, traversing G^Δ backwards (upstreams), starting from j_i. Edges from G are appended to L_i^-, analogously edges from G' are appended to L_i^+. The traversal stops when another join node or the end of the path is reached. In the example of Figure 3.5 (ignoring the dotted subtree), processing of j_0 would yield $(L_0^- = P_0,\ L_0^+ = P_0')$ with $P_0 = [3 \to 4]$ and $P_0' = [6 \to 4,\ 7 \to 6,\ 1 \to 7]$. As described, the ends of the respective paths define the replicator pair (r_i, r_i'), with $r_0 = 3,\ r_0' = 1$ in the example (grey label). If $r_i = r_i' = s_i^<$, the replicator move is classified as non-competing. Finally, L_i^- is reversed, such that the most upstream edge is removed first, continuing in downstream direction. Individual update steps are stored in an ordered set, in order of the join node processing.

To consider member changes, within the backward DFS, a forward DFS is started when a node sw has $deg_{out}(sw) > 1$. The subtree branching from 7 in the example (dotted lines), is traversed by a forward DFS, started when the backward DFS reaches 7, and is inserted as a nested sublist within P_0'.

Phase 3 - Processing residuals: After processing join nodes, there might be residual, unhandled parts in G^{Δ}, which are subsequently handled in a second traversal step. These can be isolated trees that do not contain a join node. Root nodes with $deg_{in}(sw) = 0$ are determined and processed in order by employing a forward DFS to identify residual paths and create update steps accordingly.

3.4.2 Translating Graph Changes to the Data Plane (S4 to S5)

After the path update algorithm has decomposed the (G, G') transition into a partially ordered set of edge update steps, the update steps have to be translated into SDN flow rule updates on a per-switch basis. Since distribution tree calculation is executed by the logically centralized network manager and thus both, topology data and switch management data is present, this step is straightforward: the ***rule update generator*** (S4) translates the directed edges l of the distribution tree into an associated switch-switchport pair (sw, p_{sw}), where sw is the source of l, in the order given by the update steps. The *update aggregator* component keeps track of rules installed in sw's flow table and determines an appropriate incremental rule update action[1] for each (sw, p_{sw}).

Lastly, the ***update executor*** executes all rule updates (S5) given by the rule update generator using an update mechanism as determined by the network manager (S2). For the stateless update mechanism, it coordinates the execution on multiple switches and guarantees total execution order. Thus, the invariant, selected by the update strategy is guaranteed to be maintained throughout the whole update process. Update scheduling and execution has been subject to intensive research, e.g., [JLG+14, PKCK14, XYL+17]. We thus consider scheduling to be out of the scope of this work. Moreover, we identified several opportunities to leverage parallelization of update execution in our approach, whose implementation we leave for future work.

[1] When, e.g., a flow-table entry in sw for group messages already exists, it is either deleted or its out-port action is changed to include or exclude p_{sw}.

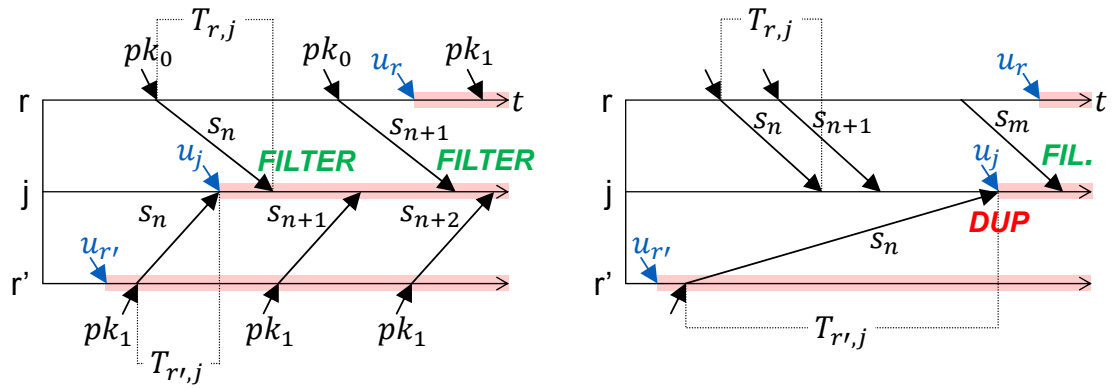
3.4.3 In-network Duplicate Filtering

In this section, we describe an approach to mitigate duplicates which may occur in our stateless update mechanism with *drop-prevention* strategy, i.e., $u^{+}_{r'} \lhd u^{-}_{r}$. This approach is applicable to both replicator moves and replicator swaps. We tackle the symptoms of this update inconsistency by installing an additional rule at the join node j, associated with a replicator pair (r, r'), which aims to detect and filter duplicates after the actual replication (post-filtering). Consider a replicator swap, where the path from the split node $s^{<}$ to j over r ($s^{<} \xrightarrow{r} j$) and over r' ($s^{<} \xrightarrow{r'} j$) are completely disjoint, as in Figure 3.5 with $j = j_1 = 5$, $s^{<} = 1$, $r' = 4$, $r = 3$. The base filter principle is illustrated in Figure 3.9(a): when the first packet pk_1 reaches j over the (new) path P' we consider all consequent packets reaching j over the (old) path P to be duplicates. Thus, on ingress of pk_1 at j, we install a rule to drop all packets, reaching j over the ingress port that is associated with the link to j's predecessor in P. For the sake of illustration only, we assume that each packet is assigned a strong monotonically increasing sequence number s_i at ingress. In the illustration, the ingress of pk_1 with s_n from r' triggers the installation of the drop rule at j (u_j), which identifies subsequent packets from r (pk_0 with s_n, s_{n+1}) as duplicates and drops them accordingly.

In standard OpenFlow, flow rules typically cannot change the state or content of other rules, which is mandatory here. However, even in early releases of commonly used SDN switch software implementations, such as Open vSwitch [PPK+15], local switch logic has been enabled by implementing the *Nicira Extensions*[2]. They implement, *inter alia*, a MAC learning switch, where packet ingress triggers the installation of new forwarding rules, completely based on local logic, without controller involvement. We propose an approach for switch-local control logic (autonomously) handling generalized control plane and data plane events in Chapter 4. Incorporating local switch logic thus, analogously, allows us to pre-install the described drop rules that are activated by local logic, i.e., packet ingress, by the switch, without time-consuming controller involvement.

However, depending on differences in the accumulated propagation delay of $r \xrightarrow{P} j$ and $r' \xrightarrow{P'} j$, this approach can only guarantee partial duplicate filtering. As illustrated in Figure 3.9(b), duplicates pass unfiltered, when $T_{r',j} > T_{r,j}$. Vice versa, when $T_{r',j} < T_{r,j}$, yet unreceived packets from r may be filtered, leading to effective drops. However, similar to the effective update order method, effective drops can be avoided by deferring u_j such that filtering becomes effective after the last non-duplicated pk_0 from r reached j.

[2] https://git.io/vgTKL



(a) Total: $T_{r',j} \approx T_{r,j}$ (b) Partial: $T_{r',j} > T_{r,j}$

Figure 3.9. In-network duplicate filtering: the ingress of a packet from r' triggers the installation of the drop rule at j, which identifies subsequent packets from r as duplicates and drops them accordingly. The filter effectiveness is dependent on $T_{r',j} - T_{r,j}$.

In conclusion, the effectiveness of the filtering approach is anti-proportional to the difference of $T_{r',j}$ and $T_{r,j}$, however, removing any (extent of) unnecessary load from a network is beneficial. Awareness of this dependency allows the network manager to gauge the costs and benefit and dynamically decide whether to apply duplicate filtering (S2).

3.4.4 Hybrid Update Mechanism

While effective order elimination and duplicate filtering mechanisms are able to partially decrease the extent of invariant violation by tackling the symptoms of inevitable update inconsistency, in this section we present an optimized state-based approach, eliminating the reason of inconsistency. We present an optimized version of the prominent *two-phase update* approach of Reitblatt *et al.* [RFR+12], combined with a stateless update ordering as an alternative update approach for multicast networks.

We optimize the original approach (see Section 2.3.2) as described in the following. Through the conducted extensive analysis on the concrete problem of update consistency of multicast networks, we identified the critical parts of reconfiguration, where drop- and duplicate-freeness breaks, as the replicator pair (r, r'). In order to maintain both—in fact, arbitrary invariants—we employ a two-phase update, but limited to the replicator pairs. While the calculation of the necessary updates stays unchanged, their execution order (S3) is changed: $P'_i \setminus r'_i$, denoted as P'^0_i, is installed, before a two-phase update, limited to (r_i, r'_i) is conducted. Note, that the installation order of P'^0_i is even irrelevant, since the associated path is not used until the two-phase update of (r_i, r'_i) has been executed. After

(r_i, r_i') has been updated, P' is effective, while P is ineffective due to the update of r, such that P_i^0 can be safely removed in arbitrary order. The update order of $u_{P'^0_i} \lhd u_{(r,r')} \lhd u_{P_i^0}$ has to be guaranteed though.

Depending on the severity of the update inconsistency effects, the network manager might decide (S2) to prefer rule space capacity and use this approach only if effective update order and duplicate filtering mechanisms would yield bad results, rather than blindly apply it whenever the technical condition (a vacant header field) is met.

3.5 Evaluation

Our evaluation consists of two stages: first, we analytically evaluate the impact, i.e., the number of replicator moves and affected nodes, for a varying degree of network dynamics (random member and link changes) for a small and large WAN topology. Second, we apply our approach to these graph changes to transform them into rule changes and corresponding network updates, which we execute on an SDN network emulated with Mininet [LHM10]. We empirically measure the occurrence of drops and duplicates directly on the data plane, while varying the update strategy and the degree of the random reordering of the given update sequences. All stages were executed on a dual-socket Intel Xeon E5-2687Wv3 (10 physical cores at 3.1 GHz per socket) with 128 GB RAM, running CentOS 7.

3.5.1 Stage I: Impact Analysis of Network Dynamics

Methodology: For Stage I, we implemented a *scenario generator* which first creates a random initial state (source node and members selection) for a given number of members m and a given topology, along with an according distribution tree. The scenario generator then simulates random data-plane events, i.e., (a) member changes and (b) link changes, which trigger a recalculation, leading to a new tree. This change-recalculate process is repeated in a stepwise execution model, where one step is denoted by a time period p_i, as illustrated in Figure 3.10. All trees of one scenario generator run are called a *scenario*. Distribution trees are calculated using a C++-implementation of Zelikovsky's $^{11}/_{6}$-approximation [Zel93, CGSW14] of the minimal Steiner tree problem [HHLY14].

For (a) membership changes, we employ a probabilistic model, where p defines the probability of a node changing its membership state in the subsequent step. For (b) link changes, we assume a congestion / over-utilization of a link, which we simulate by a

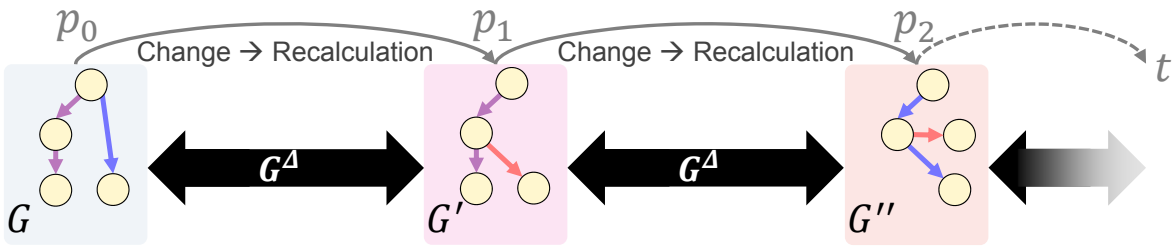


Figure 3.10. Scenario Generator, stepwisely recalculating the distribution tree, triggered by simulated link and member changes.

temporary significant increase of the respective edge weight, where c denotes the number of over-utilized links, per period.

In each period, the change analyzer creates a G^Δ and assesses three metrics: (1) the number of replicator moves, (2) the total number of affected nodes, and (3) the distribution of replicator update types within that period. Note that non-competing replicator moves are not considered here. Furthermore, recall that (affected) nodes do not consider potentially connected end-systems, which would be affected as well. The path update algorithm calculates all edge update sequences, which are dumped and used in Stage II.

In Stage I, we use two real-world WAN topologies of different scales (number of vertices v, number of edges e): the European National Research & Education Networks (NREN)[3] with $v = 440$, $e = 599$, as well as the IP-backbone of the German Research & Education Network (DFN X-WiN) with $v = 50$, $e = 76$. Link latencies are used as initial edge weights. For NREN, latencies missing in the obtained data set were interpolated (with an added Gaussian-distributed random error), whereas latencies of the X-WiN were obtained from the web service of its active probing system[4]. The number of initial members m is given as a fixed ratio of $m/v = 0.4$. An exemplary Steiner tree based on the DFN X-WiN underlay topology is shown in Figure 3.11. The degree of dynamics is gradually increased in six levels, ranging in $p \in [0.005, \ldots, 0.1]$, $c \in [0, \ldots, 10]$ (p: member change probability; c: number of simulated link over-utilizations). To reduce the impact of structural dependency on the initial state, each level is evaluated by a common set of five scenarios. Each scenario consists of 200 periods.

Results: Figure 3.12 shows period- and scenario-aggregated mean ratios of replicator moves. Error bars in the figure and stated variances henceforth refer to the standard deviation of the inter-scenario aggregation, i.e., among scenarios. As expected, a strong

[3] obtained from `http://www.topology-zoo.org/eu_nren.html`
[4] `http://pallando.rrze.uni-erlangen.de:8090/services/MA/HADES/DFN`

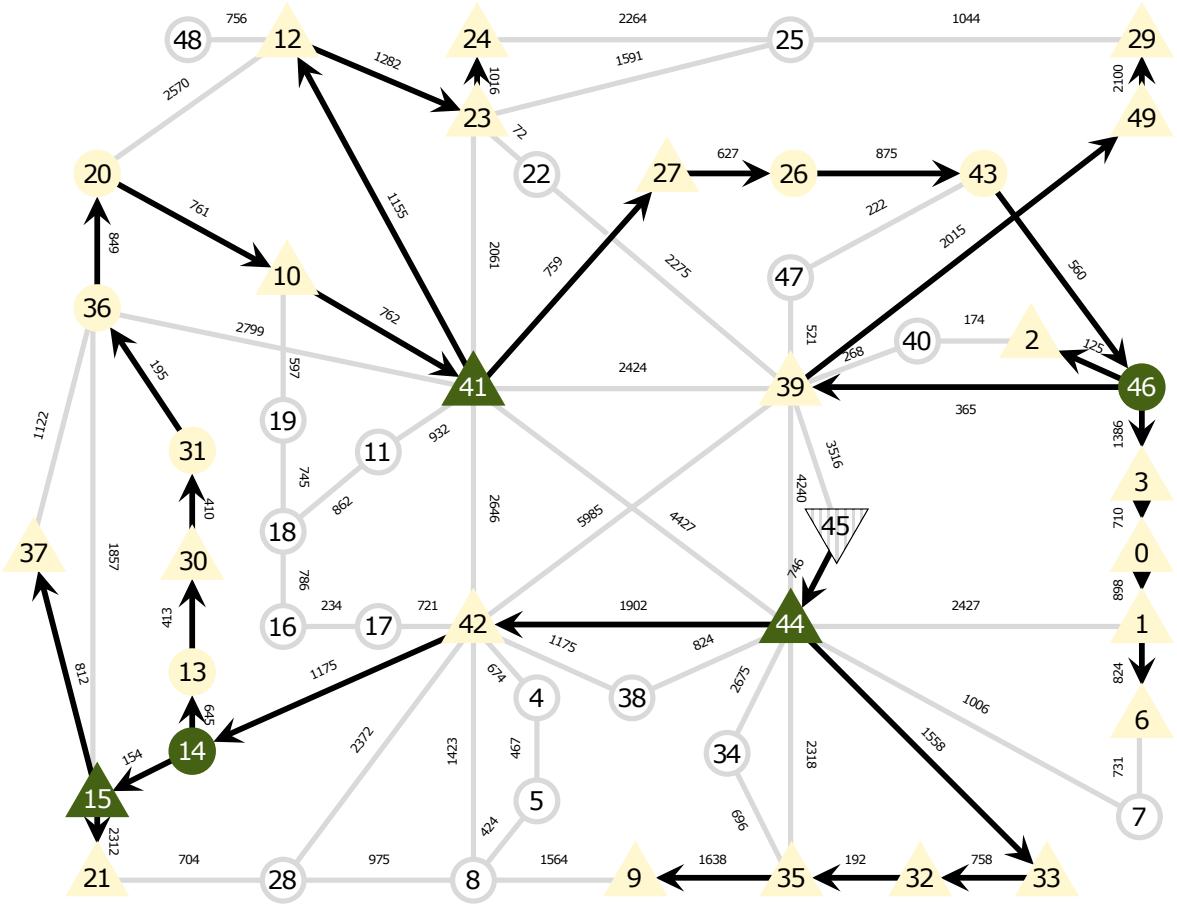


Figure 3.11. DFN X-WiN base topology (light edges, light and unfilled vertices; edge labels: avg. link delay in µs) with superimposed exemplary multicast distribution tree (dark edge l; link delay $w(l)$; downward triangle: source s (*node 45*); upward triangle: member m; dark vertex: replicator r; light vertex: relay *rel*). In the Steiner tree model (see Section 3.3.1), the source node s and member nodes M constitute the set of terminals $V_T = S \cup M$, whereas replicators *Rep* and relays *Rel* constitute the set of non-terminals (Steiner vertices) $V_{St} = Rep \cup Rel = SW_{MC} \setminus T$.

correlation between degree of dynamics and extent of effects can be inferred.

The **number of replicator moves** directly reflects the degree of dynamics. Constantly low variance indicates low dependency on both topology and scenario.

The **number of affected nodes** naturally reflects the number of replicator nodes. Even small/few changes in the network already cause a significant extent of effects, mostly stemming from simulated link over-utilization. For instance, for $(p = 0.005, c = 1)$ the number of affected nodes is 24.21 ± 13.72 (NREN) and 13.06 ± 2.72 (X-WiN). Respective statistical ratios[5] are 5.5% of all NREN nodes and 26.1% of all X-WiN nodes. However, a high variance and thus a high dependency on the underlying topology and its initial conditions can be stated. Naturally, the specific position of the respective replicator pairs in the graph along with initial conditions and scenario parameters, such as average distance to the source or graph diameter, strongly influence the number of affected nodes.

The **distribution of replicator update types**, excluding non-competing replicator updates, is shown in Figure 3.13. The *churn* type depicts moves, where either $r \notin G'$ or $r' \notin G$. In both, NREN and X-WiN, $\approx$ 80% of the moves are replicator moves, where, on average, downstream moves are twice as likely as upstream moves for X-WiN with moderate and high dynamics and about 25% more likely for NREN, irrespective of the degree of dynamics. For *drop-prevention*, the effective update reordering, happening at upstream moves, is thus shown to be of potential practical relevance, if HW switches' flow update rates are unconsidered, as discussed. Reorder-free replicator swaps are more significant for NREN with $\approx$ 17% on average than on X-WiN with $\approx$ 10%, both showing anti-proportionality wrt. the degree of dynamics. The churn type ratio is rather constant at $\approx$ 5% for both base topologies.

We summarize the observations in stating that even moderate degree of dynamics lead to a significant extent of tree changes and thus to a significant amount of nodes, affected by update inconsistency.

3.5.2 Stage II: Empirical Validation

Methodology: In Stage II, we translate the edge updates from the scenarios of Stage I into corresponding `FlowMod` messages (OF 1.3) and execute them in an emulated network with X-WiN topology and given link characteristics (latency). We use Mininet [LHM10] version 2.2.1 with Linux Traffic Control (TC) enabled links, which allow for

[5] Note that this ratio does not imply the coverage of actual nodes, since a single node can be affected by multiple replicator moves within a period.

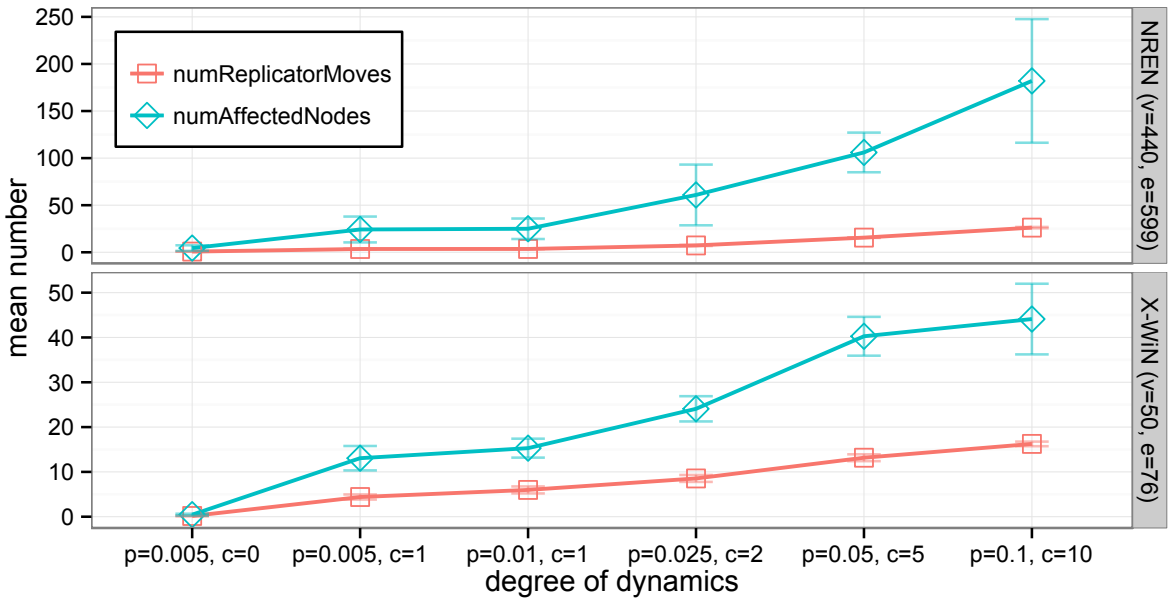


Figure 3.12. Analytic evaluation (Stage I): period and scenario aggregated mean values of number of replicator moves and affected nodes for varying degree of dynamics. Even small dynamics cause a significant extent of network changes.

link latency emulation, in combination with Open vSwitch (OVS) [PPK+15] version 2.3.2. The rule update generator and the update executor are implemented as a module for the Python-based Ryu SDN controller. Since the update executor has to guarantee total update execution order on a switch basis, OF BarrierMessages (see Section 2.2.3) are used as flow-modification feedback mechanism, i.e., updates sent to a switch for execution, block sending updates to other switches, until the executions of all former updates are acknowledged.

Recent SDN hardware switches have a limited flow modification capacity of around 40 flows per second (update rate) [HYS13, RSU+12] due to inherent properties of TCAM. We simulate this rate through an artificial delay in the update executor. However, to show general applicability, we present an extended evaluation with update rates up to 1000 flows per second at the end of this section. However, other work, including [JLG+14], suggest a high volatility of the flow update rate and strong dependency on a number of factors, such as control-plane load, number of installed rules, rule priority and complexity (actions). The update executor processes all translated updates of a scenario period, before progressing to the next period. This results in a typical execution time of 1.1 ± 0.3 s per period.

We measure the occurrence of drops and duplicates within one period directly on the

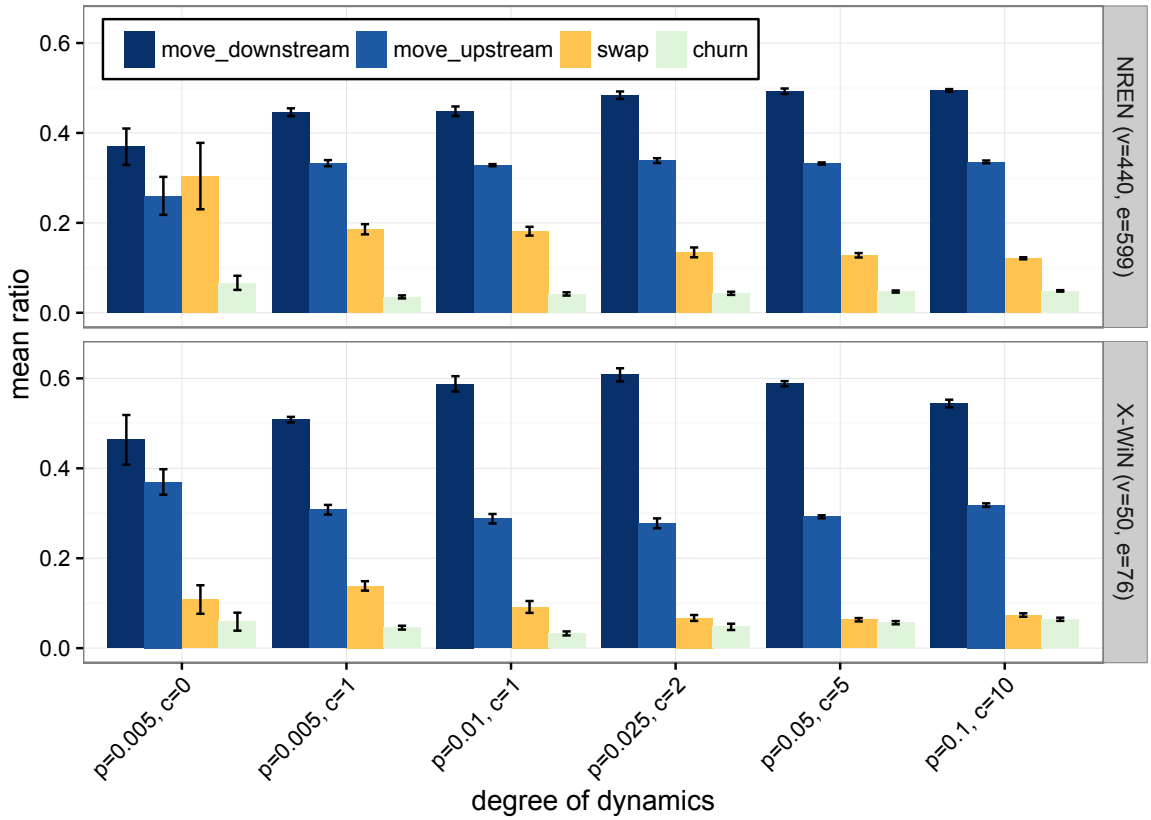


Figure 3.13. Distribution of replicator update types

data plane, while updates are being executed. Therefore, the sender node is added as a Mininet host that constantly sends group messages, containing a sequence number as payload, at a rate of 250 pps (packets per second), which is a realistic number, e.g., for media streaming, and a packet size of 50 Bytes. We capture the complete traffic on all Mininet network interfaces, i.e., switchports. The number of captured packets is denoted by n. Through evaluating the sequence numbers of captured packets on a switchport, we directly measure the number of duplicates du. To assess the number of dropped packets dr, we evaluate sequence number gaps between two consecutively captured packets on a switchport, respecting period-borders. We then aggregate duplicate and drop values to the switch level by summation and evaluation of possibly overlapping sequence number and their gaps, respectively.

Here, edge updates of the five scenarios with moderate degree of dynamics ($p = 0.005$, $c = 1$) from Stage I are used. The update strategy is varied between drop prevention (`ADD_F`) and duplicate prevention (`REM_F`), to validate the effectiveness of our approach.

As a baseline, we provide a random update order. To show the implications of deviation from the pure strategies up to complete randomness, we employ a gradually increasing extent of random reordering: p_s denotes the probability of a message within an ordered list of messages to be chosen for reordering. The set of chosen messages are then randomly reordered, using Fisher–Yates Shuffling [WIS39, Knu97]. While unchosen messages stay at their list position, chosen messages are replaced by shuffled messages. Note that only the ordered elements within an update pair (L_i^-, L_i^+) are shuffled, whereas the order among update pairs (strategy) is maintained.

Results: Figure 3.14 shows mean **packet drop factor** (quotient of number of dropped and number of expected packets: $\lambda_{dr} = {}^{dr}/_{dr+n}$) and **duplication factor** (quotient of number of duplicates and number of captured packets: $\lambda_{du} = {}^{du}/_{n}$) of affected nodes for varying degree of reordering, where the extremes (leftmost, rightmost) show pure strategies (no shuffling: $p_s = 0$), with a gradual 0.25 p-increase (partial shuffling) towards the center, which is fully shuffled ($p_s = 1$). As illustrated, our approach can be shown to be correct: the drop prevention strategy (left) successfully prevents drops $\lambda_{dr} = 0 \pm 0$, at the cost of duplicates $\lambda_{du} = 0.05 \pm 0.005$ and vice versa for duplicate prevention (right): $\lambda_{du} = 0 \pm 0$, $\lambda_{dr} = 0.09 \pm 0.01$.

Deviating from a pure strategy, e.g., drop prevention, is shown to result in introduced effects, actually to be prevented, e.g., drops, as expected. While the extent of the inverse effect, i.e., duplicates, decreases strong monotonically with increasing p_s in the case of drop prevention, curiously this is not the case when deviating from pure duplicate prevention. One possible explanation is the small topology scale, typically resulting in short message lists, where the effectiveness of probabilistic shuffling is small, such that the extent of reordering is similar for a large range of p_s-gradations.

Another asymmetry is present in the average λ_{dr} being almost $2 * \lambda_{du}$. This asymmetry however naturally follows from the nature of removal and installation of unicast paths to a respective join node j: for the non-shuffled case of *add first*, the new path to j is first fully installed, before the old path is removed. Until the last edge in the installation phase has been installed, neither drops nor duplicates occur. In contrary, with *remove first*, the first removal of an edge immediately results in drops, lasting over the removal and installation phase, until the last edge of the new path has been installed. Thus, drops are much more likely than duplicates. Similarly, for shuffled cases, the probability to have the only unicast path to j broken by a reordered and thus premature remove-update is much higher than the probability to have a complete redundant unicast path installed, despite the mixing of installation and removal updates. For averagely larger path lengths, e.g., in

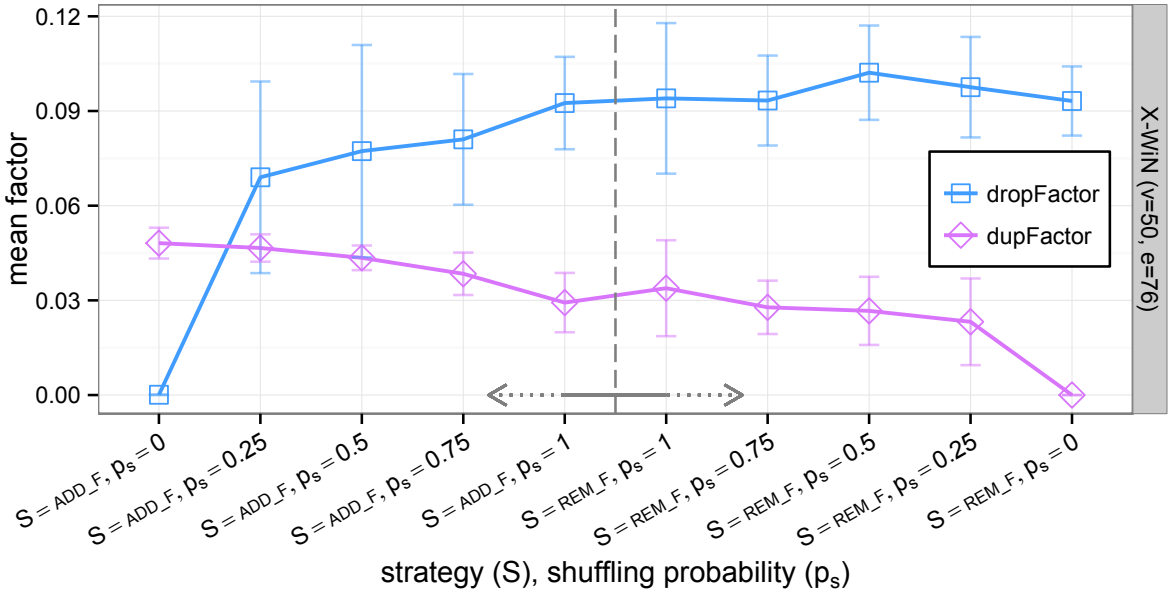


Figure 3.14. Empirical data plane effect occurrence evaluation of affected nodes (Stage II): mean drop and duplication factors for varying strategy and degree of random update message reordering, validating the correctness of our approach.

larger topologies, this effect is expected to be of much higher extent.

In a last evaluation, we have increased the assumed flow update rate to 1000 s^{-1}, the order of magnitude we could see on upcoming SDN switch hardware. In order to reliably assess update inconsistency effects, we target a packet rate that is one order of magnitude higher than the update rate, i.e., 10 kpps. On our evaluation machine, Mininet does not scale further than approx. 3 kpps for the X-WiN base topology, using default Linux virtual Ethernet (veth) pair links. Thus, we have to use OVS specific *OVS patch links*[6] to interconnect OVS bridge ports. Since these interfaces are not exposed to the OS, performing packet capture on them is not possible. We thus add a *Linux dummy interface* to each OVS bridge and mirror all packets from the other (regular) OVS ports to it. Capturing on those dummy interfaces thus allows for assessing the complete network traffic. The results shown above were confirmed for increased update and packet rates, however, the changed evaluation method introduces a packet reorder rate of about 1%. Furthermore, OVS patch links do not support link latency emulation, so that we could not consider timing aspects.

[6] see manual page of `ovs-vswitchd.conf.db`

3.5.3 Evaluation Conclusions

Our evaluations have shown the relevance of the addressed problem of multicast tree updates even for moderate degrees of network dynamics and have validated the correctness and practicability of our update order approach.

The analytical part (Stage I) has shown that both, the underlying topology and the scenario, have a low impact on the occurrence of replicator moves, which are causing inconsistency effects. They have shown a high prevalence of about 80% of all replicator update types. Furthermore, even small and few changes in the distribution tree were shown to cause a significant extent of inconsistency effects in terms of number of affected network nodes, ranging from 5.5% of all NREN nodes to 26.1% of all X-WiN nodes with an apparent higher impact of the underlying topology.

The empirical evaluation of inconsistency effects on the data plane (Stage II) have shown that drop-freeness can be achieved at the cost of 5% duplication rate, whereas duplicate-freeness comes at the cost of 9% packet drop rate, even for update rates exceeding TCAM's update rate by two orders of magnitude. A deviation from the correct update ordering (through a random permutation of update steps or a systematically selection of the opposite strategy) would result in averages of $\approx$2.5% to 5% duplication rate and $\approx$7% to 10% drop rate. Since duplication is caused by an additional effective path to affected nodes consisting of possibly many path segments, duplication effects are overall of lower extent than packet drops, which occur if an effective path is broken and hence already when only a single path is missing in the path.

3.6 Related Work

SDN constitutes a perfect match to solve key deployment problems that have been impeding the adoption of multicast in large-scale real-world scenarios so far. Especially for data center networks, a management method of multicast in overlay networks is described in [NHS12]. Avalanche enables secure and bandwidth-efficient multicast in DCNs [IKM14]. Scaling and routing of multicast in data-center topologies is investigated in [LYYW11]. For multicast-based streaming with multiple simultaneous streams among multiple WAN sites under real-time requirement (3D teleimmersion), [ARTN13] introduces an SDN-based control protocol allowing for seamless reconfiguration of the network. Bandwidth and connectivity invariants are maintained using a state-based update procedure. Stateless network updates are not considered in these works. However, the number of recent works

indicate a clear trend for SDN enabled adoption of multicasting in both, DCNs and WANs.

As mentioned in the introduction, in the domain of network update consistency, a state-based mechanism guaranteeing arbitrary invariants [RFR+12] as well as stateless approaches, based on update ordering have been proposed. A minimal stateless procedure as well as an overview of correctness properties and their interdependencies are presented in [MW13]. With the aim to improve update speed, dynamic scheduling of consistent updates respecting these interdependencies is proposed in [JLG+14]. A stateless search-based approach is presented in [MFC14]. Neither approach considers the peculiarities of updating multicast networks though.

In the domain of network management, a rather early contribution from the year 2013 explores SDN's ability to ease management and configuration of a variety of networks [KF13]. The authors propose a network control architecture focussing on the interdependency of high-level network policies, declared in a functional programming language, and low-level network configuration, including its deployment in a campus network. More recent and detailed work on the SDN-enabled management of large-scale networks propose a layered and distributed control plane. The authors of [FBC+15] focus on intra-domain control and management of large-scale networks and present revised algorithms for hierarchical routing and local link-failure recovery in the context of management distribution, exploiting global network knowledge. In [TCCP15], a layered management and control framework for fixed backbone networks is proposed, along with a placement algorithm for control entities. This allows for adaptive resource management operations as demonstrated on two exemplary use cases, adaptive load-balancing and energy management. Neither of these approaches considers efficient incremental updates or identifies the relevance of update consistency.

3.7 Conclusion

In this chapter, we have proposed a generic system architecture for network management, focussing on change management. We have proven that it is impossible in general to achieve drop-freeness and duplicate-freeness simultaneously just by ordering updates in a multicast network. We have presented a detailed formal analysis of this update problem. In order to alleviate this problem, we have proposed a flexible update approach, allowing for selecting a strategy that either prevents duplicate or drops. We have shown that update consistency is multifarious and comprises many degrees of freedom, spanning a large configuration space. In combination with the severity of impacts on the network

performance, this has shown the relevance of update consistency in network management and argues for incorporating update consistency awareness in the network reconfiguration process.

Our evaluation has shown the relevance of the addressed problem even for a small degree of network dynamics and has validated the correctness of our update order approach: drop-freeness can be achieved at the cost of as few as 5% introduced duplicates. Since duplicates, in contrast to drops, have been shown to be of less extent and certainly can be assumed less fatal for most applications, our approach is highly practical.

4 Full-range Distribution of Event-based Network Control

In Section 2.4.1, we described architectures for control plane distribution. Given that distributed SDN controllers hence already exist today, can we conclude that their evolution has reached its end? We argue that this is not the case. In this chapter, we thus make the case for a full-range distribution of SDN controllers and present our novel event-based architecture, providing maximum flexibility with respect to distribution and improved manageability. We motivate our concept by identifying the following inflexibilities and deficiencies of recent distributed control schemes.

First of all, implementing *fully distributed network control* (without switch-external control functions) is not anticipated. In other words, the traditional SDN approach mandates an external network controller, be it monolithic or distributed. Direct switch-to-switch communication in the control plane and hence coordination for network control [VCB+15] is not possible. This reflects the clean-slate paradigm shift from distributed network control to logically centralized control, where switches are just "dumb" network elements, stripped from any control logic but specialized to do fast forwarding, according to rules defined by an "intelligent" remote controller implementing all network control logic. Traditional decentralized control algorithms, such as link state routing, are removed from the switches and replaced by centralized algorithms running on the controller. On the one hand, this reduces the functionality of switches to a bare minimum, allowing for minimal switch resources and design. On the other hand, outsourcing all control from the switch comes at the cost of increasing latency due to incurring switch-controller round-trip times (slower reaction), increasing load on the control network, or difficult implementation of robust logically centralized control relying on additional machines that can fail. Therefore, we argue that a highly flexible SDN architecture would allow for the full spectrum of distribution, from fully centralized to fully distributed control. In other words, control logic has to be brought back onto the switch. Although execution of control logic on the switch hardware on the one hand has been conceptually proposed in literature [CMT+11, HYG12, DHM+14], due to lack of distribution support or high computational

resource demand, in concrete implementations it has been reduced to offloading of certain functionality, such as packet generation [BBBS16] or state machine logic [BBCC14]. To fully exploit the locality of switches, we argue to include the switch in the control distribution and allow for decision making on the local scope. Besides the extremes—fully centralized and fully decentralized control—we argue that network control decisions are ideally be taken as local as possible, in order to *minimize control latency*, while leveraging the logically centralized paradigm of SDN through access to global knowledge in order to *improve decision quality*. Since requirements, such as timeliness, optimality, and consistency, may tremendously differ between network functions, a network control architecture should provide the flexibility for balancing these trade-offs for each individual network function to account for their heterogeneity. For instance, for forwarding decisions at a switch, full global knowledge is typically not required. The focus rather lays on timeliness in order to reduce forwarding latency. In contrast, traffic engineering or monitoring are applied on a much broader time scale and thus looser latency constraints, but relying on more global knowledge for improved solution quality.

Secondly, with the current concept, we observe that controllers tend to be quite heavy-weight (which might also be a practical reason why control has been removed from switches). For instance, in order to just receive packet-in events, the ONOS controller requires a full-fledged OSGi environment with a total code size of $\approx 216\,\text{MB}$. Controllers that are more lightweight typically lack modularity or distribution capability. We argue that it should be possible to identify a minimal feature set that every control module can implement to communicate with switches and other distributed control modules. Anything else should be factored out into the implementation of the control function. In other words, we advocate a lightweight *micro-kernel* approach for SDN controllers instead of a heavyweight monolithic controller architecture.

Thirdly, we observe that switches and controllers are still tightly coupled, which hinders the free distribution of control logic. For instance, an OpenFlow control channel (see Section 2.2.2) typically requires a TCP connection to a controller. Since TCP is inherently connection-oriented, spawning new control applications at other machines or migrating them between machines is cumbersome and potentially disruptive [KCGJ14, BBHK15]. We argue that switches must be *decoupled* from the SDN controller. This can be achieved by using state-of-the-art communication middleware as already successfully used in other domains for the communication between services [Cha04]. As a side effect, choosing a suitable communication middleware also allows for implementing control logic in virtually any language and to support event-driven as well as request/response types of interaction.

The **main contribution** of this chapter is a novel architecture for a distributed SDN controller fulfilling all of the above requirements: (1) high flexibility with respect to distribution of control logic covering the whole design space from logically centralized to fully distributed control; (2) micro-kernel controller architecture for distributed lightweight controller modules, so-called controllets; (3) push-down of controllets implementing control logic onto switches, allowing for fast local decision making while leveraging global knowledge if required; (4) decoupling of controllets through a message bus supporting content-based filtering of so-called data plane events. Furthermore, we address challenges in practical deployments of switch-local controllets, where we employ lightweight virtualization techniques to cope with hardware heterogeneity and to implement isolation and resource control for a safe and controlled control plane operation.

The rest of the chapter is structured as follows. In Section 4.1, we describe the architecture of our distributed SDN controller together with an overview of the basic concepts. We proceed with describing the message bus in more detail in Section 4.2. In Section 4.3, we discuss how our concept enables highest flexibility in terms of control distribution, before we present local logic based on global knowledge, along with multiple applications. We also address the relation to data plane programming and Network Function Virtualization as well as challenges in practical deployment on networking hardware. In Section 4.4 we elaborate on implementation aspects, followed by an evaluation of performance and scalability of our distributed architecture as well as results from the deployment of on white-box networking switch hardware in Section 4.5. We discuss related work in Section 4.6 and conclude the chapter in Section 4.7.

4.1 System Architecture

We start by introducing the basic architecture of our distributed SDN controller (see Figure 4.1).

Our approach is based on what we call a *micro-kernel architecture* for SDN controllers. We split network control logic into lightweight control modules, whose instances we call *controllets* (CM_i). In contrast to a monolithic controller, controllets do not require a heavyweight execution environment. Instead, we execute each controllet in a separate process, possibly being also physically distributed, and enable communication between them. The micro-kernel (μK) just provides basic functions for messaging including publish/subscribe message routing and passing (in particular of OpenFlow messages), and registration and discovery of controllets and switches. Any other functionality like

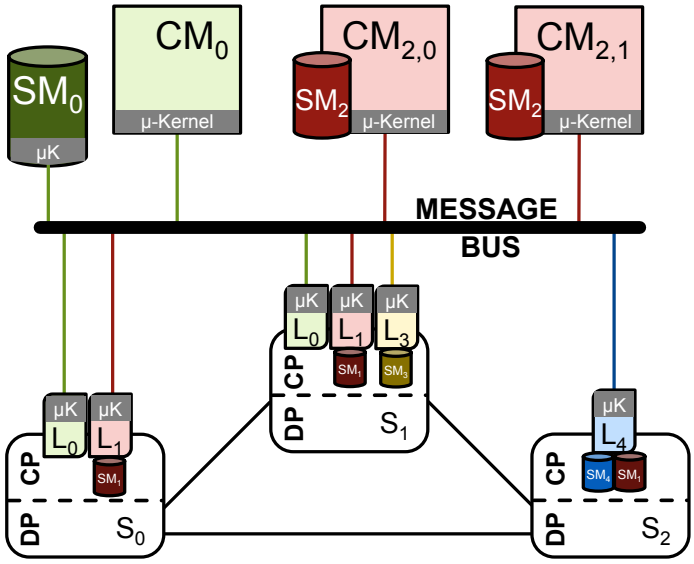


Figure 4.1. ZeroSDN's full-range distributed architecture: μ-kernel architecture with fully distributed local *(L)* & external controllets *(CM)*, interconnected by a message bus.

network topology management, routing, etc. is implemented by the controllets' "business" logic. One advantage of having a slim functionality for the SDN micro-kernel is that we can port the micro-kernel with little effort to different languages enabling us to basically use any language for the implementation of controllets. Moreover, the lightweight nature of controllets enables us to push down control logic by executing controllets directly on switches (S_i), instead on remote server hardware. We denote controllets running locally on switches as L_i. Opposed to a monolithic solution, the distribution of control logic comes at the cost of increased complexity for the distribution of its state. We discuss trade-offs in control distribution in Section 4.3.1.

Communication is based on a unified *message bus* to decouple controllets from switches and other controllets, both, logically and physically. We are thus able to reduce the switch-controller coupling to inter-module communication over the message bus. Each controllet and switch can communicate with other controllets or switches through the message bus by sending events using the publish/subscribe (pub/sub) paradigm, or sending direct messages using the request/response paradigm. Decoupling controllets and switches allows for flexible distribution including migration of controllets, and dynamic spawning or exchanging of controllets at runtime. The message bus implementation is integrated into the micro-kernel. Overall, this architecture allows for maximum flexibility.

4.2 The SDN Message Bus: Decoupling Controllets through Events

Our architecture is based on *event-based communication* to decouple the producers of events from their consumers in both, time (asynchronous communication) and space (distribution of logic between nodes). In this section, we show why the event abstraction is both desirable and well suited for network control and describe the design of our message bus along with an example.

In the domain of SDN, we particularly consider so-called *data plane events* (DPE) stemming from packets or state changes of data plane elements (switches and end-systems). Virtually anything in the data plane can be modelled as a data plane event, for instance, the addition or removal of network elements, link status updates such as detected congestion or over-utilization as discussed in Chapter 3, and packet ingress or egress. From certain DPEs state information can be inferred, such as knowledge of the physical network topology and end-system protocol state, e.g., TCP-sessions. A DPE is either processed in the hardware forwarding-pipeline of the switch, e.g., a packet ingress is processed according to the flow rules installed in the switch's TCAM (*fast-path*), or is being forwarded to the control plane (*slow-path*), e.g., when no matching forwarding rule exists. In the latter case, the switch silicon passes the associated packet to the switch's CPU, where it is encapsulated into an OpenFlow `PACKET_IN` message. When not processed locally (see Section 4.3.2), the switch publishes the DPE to the message bus, which delivers it to controllets that are subscribed to this kind of event. The message bus is responsible for routing event notifications to their subscribers. Since DPEs often include matches on packet header fields, we argue that the message bus should support *content-based filtering* of events [EFGK03]. Therefore, event conditions include matches on header field tuples or any other metadata. This paradigm can also emulate standard client/server communication (request/response), multicast, or topics [EFGK03] using filters on receivers, groups, topics, etc.

Event routing in the message bus is exemplarily illustrated in Figure 4.2: an ingress TCP segment from an end-system at S_0 is encapsulated in a DPE (`OF_PKT_IN`) and published to the message bus, where a remote monitoring firewall controllet (`Mon`) and one instance of a remote forwarding controllet (`Forw2`) have matching subscriptions, i.e., are responsible for such events, and are consequently delivered the event. As a result of processing this event, `Forw2` sends a packet-out message (`OF_PKT_OUT`) over the message bus directly to S_2 using the request/response pattern. Analogously, `Forw2` installs

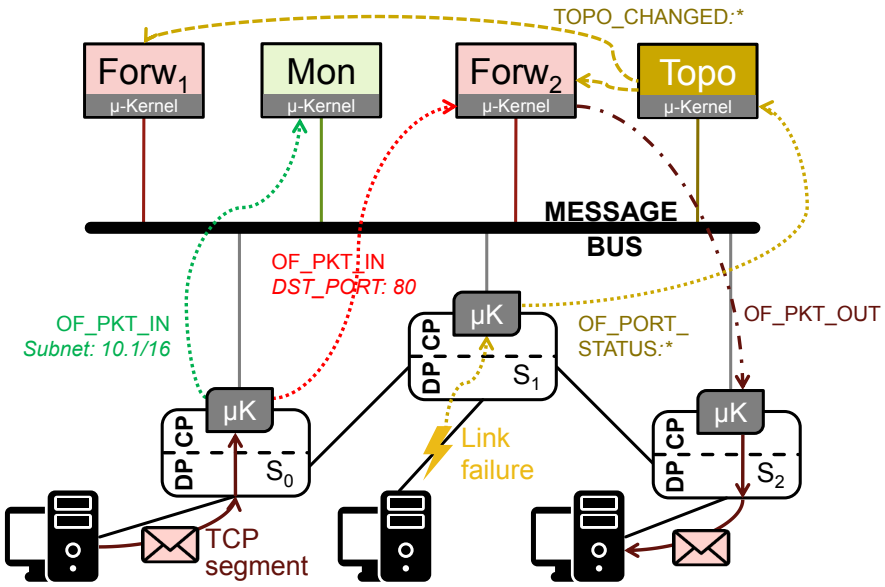


Figure 4.2. Content-based routing (publish/subscribe) of data plane events (dotted) and control plane events (dashed) for exemplary subscriptions and direct messaging (request/response; dash-dotted) over the message bus, decoupling remote controllets.

a flow from S_0 to S_2 by sending flow modification messages (`OF_FLOW_MOD`; omitted in Figure 4.2 for readability).

However, we do not restrict ourselves to *basic data plane events*, but also consider *complex data plane events* involving, for instance, multiple packets and timing conditions. For instance, a complex event could be triggered by a certain sequence of packets, or the non-arrival, i.e., absence, of a certain packet over a defined period of time, also across multiple switches. Typically, switches only fire basic events, which are then forwarded to subscribing controllets, which in turn evaluate complex event conditions to fire complex data plane events. In Chapter 5, we offload Complex Event Processing from end-systems and middleboxes to switches and programmable NICs. This concept is also applicable for the evaluation of complex data plane events, assuming such programmable network elements are available also in the control plane.

Another type of events, used for inter-controllet communication, is the control plane event (CPE), which bears state changes or other events of the controllets' business logic or their micro-kernel, such as topology changes, multicast-group membership changes as discussed in Chapter 3, changes of any high-level network policy such as firewalling, or recovery/shutdown of controllets. CPEs are mainly used for coordination among controllets. In our example, a link failure DPE (`OF_PORT_STATUS`) at S_1 is disseminated to

the message bus and delivered to the subscribed Topo controllet, which hence adapts its knowledge about the network topology. Consequently, Topo informs interested controllets by publishing a CPE (`TOPO_CHANGED`), for which all forwarding controllets have subscribed to react to topology changes, and so on. In the multicast example (Chapter 3), an IGMP message (DPE) would be received by a subscribed multicast controllet, which infers a change of group membership from the DPE and consequently triggers the recalculation of the multicast distribution tree, possibly by another controllet using a CPE, and executes the necessary tree transition through a set of update DPEs (`OF_FLOW_MOD`s).

Recent SDN research has shown that consistency in an inherently distributed system of switches and controllers might require certain semantics of the delivery of messages [KZFR15]. The message bus transparently implements a range of semantics, such as *exactly once* or the relaxed *at most once*, by employing corresponding messaging primitives, such as atomic multicast, (un-)ordered multicast, etc. Thus, the message bus provides arbitrary guarantees on message delivery (reliability) to controllets as building blocks for implementing network control with flexible consistency semantics that match the criticality of respective control tasks.

Since the message bus is a crucial system component, we want to briefly discuss its implications regarding scalability and reliability. In traditional messaging middleware, publish/subscribe used to be implemented by a hardware appliance or a software-based component, the *broker*, which manages subscriptions and implements filtering of messages in a centralized fashion. To prevent swapping one centralized component (centralized SDN controller) for another (centralized message broker), we employ a distributed solution that exhibits high scalability: Modern *brokerless* message bus implementations use efficient transport mechanisms for event dissemination, like multicast or unicast with publisher-side subscription-based filtering or even hardware-based filtering with line-rate performance [BTK+17], targeting scalability to hundred thousands of subscriptions [nan17], which suffices to accommodate typical data center networks [SOA+15]. We provide macro-evaluations of our message bus implementation in Section 5.5. Should performance issues arise nonetheless, e.g., due to an insufficiently dimensioned control network, scalability can be improved by employing a message bus hierarchy, where the scope of controllets is limited, e.g., reflecting tiers on modern data center network topologies, such as core, spine, and leaves. Regarding fault-tolerance, we stress that a worst-case complete failure of the message bus translates to a broken control channel, which is equally severe as a broken control channel in traditional, non-distributed SDN architectures. On the contrary though, local control in our architecture increases fault-tolerance, as we show later.

4.3 Highly Flexible Control Plane Distribution

In this section, we make a solid case for rethinking the radical clean-slate approach most common SDN architectures follow by showing how lightweight controllets can bring back control onto the switch while still benefiting from the logically centralized paradigm of SDN. We also address drawbacks of control decentralization and challenges in practical deployments.

4.3.1 Augmented Fully Distributed Control

Most SDN architectures have abandoned fully decentralized network control based on a distributed control plane implemented solely by switches in favor of logically centralized control. While not strictly arguing for or against logically centralized control or fully distributed control, where control logic is distributed to both, controllers and switches, we observe that the strict notion of separating data plane elements from the logically centralized control plane limits the full potential of the SDN paradigm. For instance, "legacy" distributed control protocols, such as distance vector or link state routing protocols, have proven to be fault-tolerant and scalable. As found by [KZFR15], vigorous efforts have to be undertaken to provide the same fault-tolerance with a logically centralized SDN network. We stress the fact that on the one hand maintaining a global view and exerting logically centralized control comes at a cost [LWH+12] due to the inherent need for acquiring a global state, which gets costlier the stricter the consistency requirements are, and communication with a remote control entity, respectively. While this holds true also for the decentralized control model, we stress that on the other hand, a full global view is however not even needed for many control decisions, as we show later in Section 4.3.2. Hence, logically centralized control should not be the sole option. Consequently, we argue that true flexibility in network control implies to leverage the whole design space of control (de-)centralization and thus also includes the option for full distribution of network control, as depicted in Figure 4.3.

Recent developments in networking hardware enable switch-local control logic due to (a) increased computing performance and (b) programmability through open access to the switch's control plane. In particular, *white-box networking switches* (see Section 2.5) feature open, Linux-based switch operating systems as the control plane, running on increasingly powerful CPUs (see Section 4.5.3). Therefore, and in-line with recent research [SS13, BBCC14, BBBS16, CPSC15], our architecture encourages pushing lightweight controllets directly onto the switch, as illustrated in Figure 4.1. These switch-local

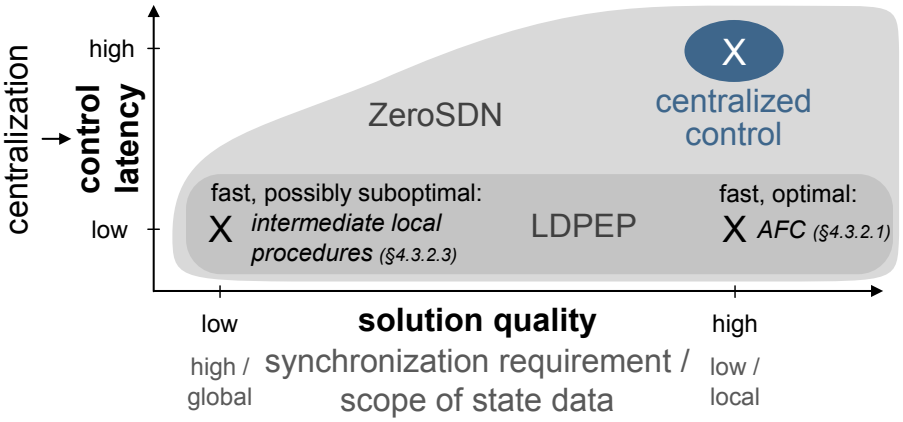


Figure 4.3. Design space and trade-offs in network control distribution. Opposed to logically centralized control with high control latency (dark blue), ZeroSDN covers the full spectrum (light-shaded area) and offers an additional degree of freedom by exploiting the trade-offs of state data scope and synchronization requirements against solution quality (dark-shaded area).

controllets can then execute the full spectrum from simple local logic to fully distributed network control protocols. Like any controllet, also switch-local controllets communicate through the message bus. Thus, we can implement distributed network control alongside logically centralized network control, or implement anything in-between (Figure 4.3, light-shaded area). This scheme allows for the best of both worlds—fully decentralized processing, yet being centrally coordinated, and logical centralization, which allows for trading-off *control latency* (latency of event processing and communication) against overheads of distribution and synchronization of controller state. For the synchronization of state among controllets (see Section 2.4.2), communication primitives of varying reliability offered by the message bus can be combined with additional methods to achieve a desired level of consistency and other properties, for instance by employing a 2-phase commit protocol for distributed transactions [MGBM17]. The selection of a suitable level of synchronicity (*synchronization requirement*; Figure 4.3, dark-shaded area) depends on the criticality of a network function to control. For instance, network operators could consider admission control more critical than monitoring or traffic engineering, where temporal inconsistencies are bearable, i.e., changes in these policies do not have to be enacted as quickly (eventual consistency).

Besides the partitioning of controller state data along network functions, state can additionally be partitioned by topological scope. Through the incorporation of (more)

global knowledge, i.e., state data of larger topological scope, we can thus additionally trade-off the *scope of state data* against *solution quality* of control decisions (Figure 4.3, dark-shaded area). As we show next, local knowledge can be augmented by partial caching or aggregation of (more) global knowledge upfront or by requesting remotely within a control decision process.

Potential control decision conflicts can be resolved by publishing all policy data and aggregating it locally alike. Local controllets decide which policy data is relevant for their control decisions, issue corresponding subscriptions, and cache received policy data.

The flexibility of our approach is to the best of our knowledge yet unmet and exploits the full conceptual range of SDN.

4.3.2 Local Data Plane Event Processing

We argue for placing control decision making as close as possible to the entities it is affecting, i.e., pushing down *decision making* instead of *decisions* (in form of forwarding entries) to the switches. We denote this concept as local data plane event processing (LDPEP). LDPEP allows for reacting most timely on data plane events, decreasing control latency. Another important advantage is that the state data of local scope naturally is most recent locally and thus has to be considered the ground truth for decision making [PZH+17]. Due to its locality, it neither has to be costly acquired nor has it to be consistently agreed upon. Furthermore, opposed to a non-local controllet, the total control load is inherently balanced to local controllets, relieving the message bus.

We apply a fast heuristic to quickly decide whether an event is to be processed locally or remotely. Therefore, we consider (i) the actual control application, (ii) the scope of the state data required for decision making, and (iii) the scope of the particular control decision. If the involved state data and decision are of limited scope and all necessary state data is locally available, the event is processed locally. Otherwise, the event is propagated over the message bus to be processed by remote entities in the control plane. Note that this decision is not exclusive and also the control scope is not necessarily limited to a single switch. Even with LDPEP, we still allow controllets to have forwarding rules being installed directly at the switch.

LDPEP not only decreases latency but also increases the network's failure resilience: it constitutes a stand-alone procedure in case an adequate remote controllet or the entire message bus is currently unavailable.

In the following, we show essential use cases enabled by LDPEP and elaborate on its design by example.

4.3.2.1 Autonomous Forwarding

A prime candidate that naturally lends itself to LDPEP is simple forwarding as, e.g., being implemented by the *MAC learning switch Nicira extension* [VMW] in the prominent SDN software switch implementation Open vSwitch (OVS) [PPK+15].

In the following, we present the concept of *Autonomous Forwarding*, which is illustrated in Figure 4.4, running on a typical switch hardware platform. Following standard OpenFlow behavior, packets (❶ from $Host_{src}$ destined to $Host_{dst}$) without matching forwarding rules in the fast-path ❷ are escalated over the slow-path to the switch's control plane (❸ `PACKET_IN`), where a forwarding decision is taken and applied by installing respective forwarding rules (❼ `FLOW_MOD`) for subsequent packets and sending the particular packet to a switch data plane egress port (❽ `PACKET_OUT`). Naively one could conclude the only state information needed for the forwarding decision was the host MAC to switch-port mapping, which is either passively learned ❹ from ingress packets or actively probed. However, the destination host might not be attached to a port of that switch. In addition, forwarding decisions might violate global network policies, such as firewall rules, ACLs, or tenant isolation.

To implement centrally coordinated control, preventing policy conflicts, and leverage global network view, the Autonomous Forwarding controllet (AFC) subscribes to relevant topology data and policy information on the message bus ①. Due to limited resources on the switch, the extent of local state caching has to be limited. Received publications about possibly interfering policies are thus aggregated ② into an *exception list*, storing hosts and local switch-ports that are affected by any policy and are thus being blacklisted (or whitelisted). Similarly, topology information is reduced to only relevant parts for local processing before being stored in the cache.

In the forwarding decision process, the MAC-switch-port mapping of $Host_{src}$ is learned and the Forward Information Base (FIB) cache is updated ❹. Note that FIB entries (tuples) may be arbitrarily extended, for instance to consider VLAN tags. Since the mapping constitutes topology information that in general is highly relevant for many other controllets as well, it is published to the message bus ④. Then, the cached topology data, i.e. the FIB, is queried for the switch-port associated with $Host_{dst}$ ❺. In case the data is not present locally, the `PACKET_IN` event can be escalated to the message bus to be processed by some remote controllet ⑤ or a request for the required data can be published. To evaluate whether autonomous local processing can be applied, the *exception list* is queried ❻. In case of a hit, the decision must not be taken locally and is thus escalated to the external control plane by a publication of the event to the message bus ⑥. Otherwise,

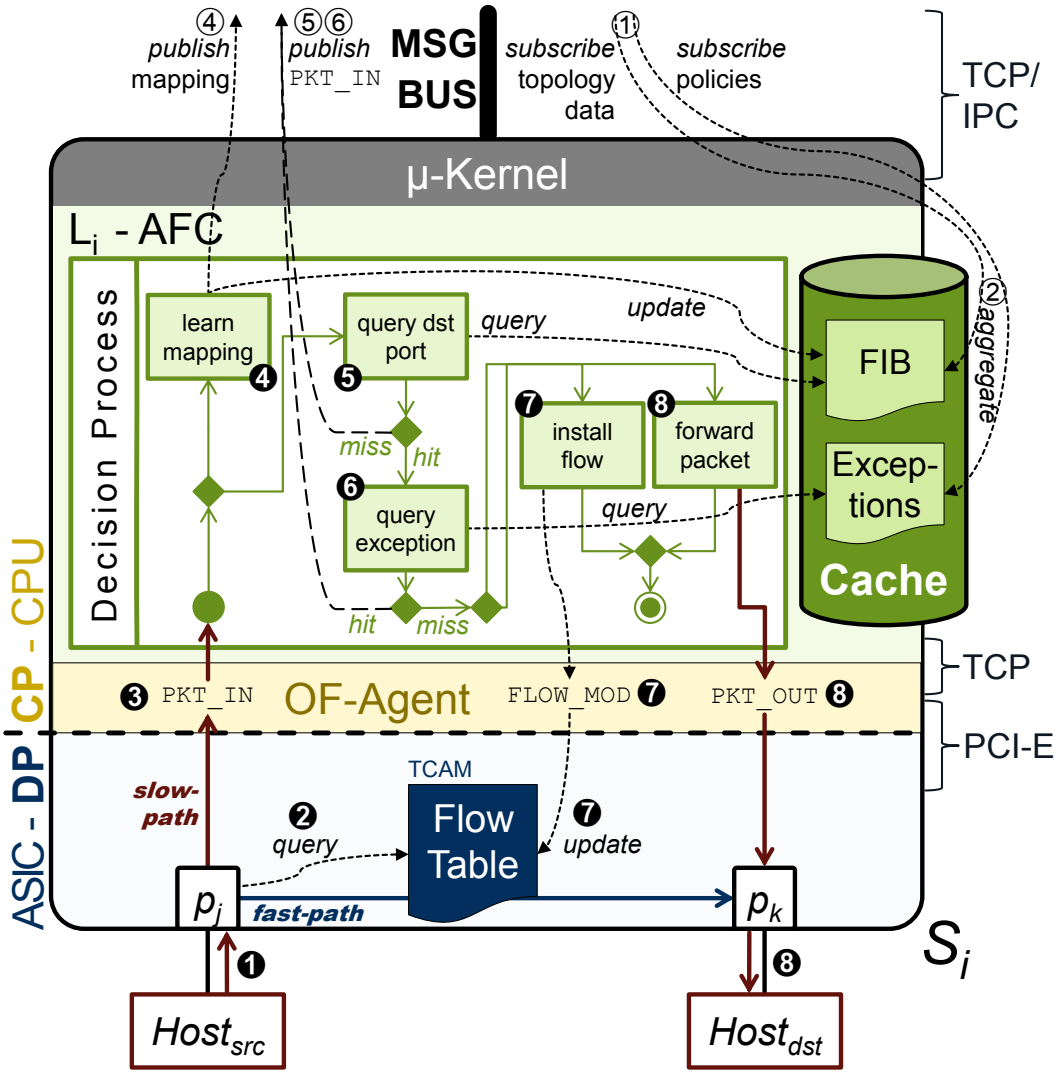


Figure 4.4. Schematic overview of the Autonomous Forwarding controllet and its processing of a local data plane event (forwarding of $Host_{src} \rightarrow Host_{dst}$) on a typical switch hardware platform. Data plane events and control plane events are routed over the message bus in order to enable coordination with other controllets and access to the global view.

local processing proceeds ❼❽. Note that for policies that can be translated directly into local drop-rules, such as admission control, affected DPEs with corresponding matches in the exception list can still be processed entirely locally.

While maintaining a local exception list is mandatory for policy adherence, the scope of non-local topology information to be locally cached can be chosen more fine-grainedly, considering the available resources on the particular switch and the desired data consistency. The scope of the local topology cache thus can range from purely local over regional (neighbor switches) to global view. This allows for trading off optimality of a control decision against resource consumption (memory, processing) and latency (for decision making and enacting). As mentioned above, data consistency is a crucial factor for the optimality and even validity of a decision. Typical cache invalidation and eviction strategies such as *least recently used* or *least frequently used* can be applied to optimize caching behavior. As a middle ground, instead of topology data itself, the cache can just store the primary source for that data—the controllet at which the data is local. Thus, in case such data is needed, the respective peer can be queried directly rather than publishing an uninformed query to the whole message bus.

4.3.2.2 ARP Handling

ARP is another essential networking mechanism, which has already been investigated in the context of local control and controller-function offloading [BBBS16, YZB+16]. Autonomous forwarding can be easily extended to include ARP handling. Additional to the link layer address data, ARP needs network protocol address data, which is passively or actively acquired, alike. Since ARP is a control protocol, we argue to employ a reactive control scheme, where all ARP requests are escalated to and handled in the control plane. Thus, at the cost of negligible memory consumption, ARP handling profits from decreased latency of LDPEP, while the remote controllets are effectively shielded from ARP control load that, in contrast to proactive flows, is to be fully handled by the control plane. Extensive evaluations of the quantitative impact of local ARP handling can be found in the mentioned literature.

4.3.2.3 Fast Failover & Adaptive Link Load Balancing

While decisions of the AFC and ARP LDPEPs are **permanent**, i.e., typically not challenged by external authorities (remote controllets), we now describe another class of LDPEP: **intermediate local procedures**. These allow for fast local reaction, while pos-

sibly compute-intensive and thus time-intensive centralized control decision is eventually determined and possibly replacing the local short-term procedure decision.

In our exemplary **local fast failover** procedure, a link failure (yet another type of data plane event) between a pair of adjacent switches (S_1, S_2) is detected at S_1 and propagated to a controllet running on S_1. A local procedure temporarily compensates the failure by steering the traffic over a link locally known[1] to belong to a redundant path to S_2. S_2 recovers analogously. Although being possibly suboptimal, local intermediate procedures provide a timely recovery, while the failure event is propagated to the message bus, where a remote controllet recalculates a globally optimal route that is ultimately deployed to the switches possibly overriding the decision of the local procedures. If S_1 and S_2 have broader cache scope, they could even avoid most suboptimal recoveries by coordinating their plans among each other using peer-to-peer communication, and adapt it in case of discovered sub-optimality. A related approach [AAK14] relying on pre-installation of failover flows and thus consuming additional scarce flow-table space shows that recovering through remote controllers is one order of magnitude slower than local procedures.

Instead of being applied to recover from (rare) failures, re-steering flows over redundant links according to the present link utilization can be a time-event-triggered (periodic) process, which we denote as **adaptive link load balancing**. This procedure is highly appealing for traffic engineering and more dynamic than traditional approaches, such as *Equal-cost Multipath Routing* (ECMP) [TH00]. Recent switch instrumentation technologies, like Broadcom's BroadView [Broa], even enable fine-grained access to hardware switch-port queue statistics, which allows for more detailed traffic analysis. Furthermore, adaptive link load balancing can be applied not only on local scopes, but rather on different levels of a whole control hierarchy, e.g., reflecting tiers on data center network topologies.

4.3.2.4 Control Plane Feedback Mechanism

Local controllets are the only entities that can *directly* access the switch's flow-table entries. Thus, any applied change to a flow table can be propagated to interested controllets, implementing a feedback mechanism that allows a controllet to verify whether its flow change has been successfully applied—a precursor for a transactional interface [CKLS15]. Although policy conflicts between controllets should be avoided by coordination upfront, with this mechanism, controllets are able to detect conflicts, e.g., when a rule, encoding a

[1] Switch to switch links can be discovered by employing active probing using the Link Layer Discovery Protocol (LLDP), as described in Section 4.4.

policy of one controllet CM_1 is modified by another controllet CM_2 such that the original policy of CM_1 is violated.

4.3.3 Relation to Data Plane Programming and Network Function Virtualization

In the following, we want to briefly discuss the relation of ZeroSDN in general and LDPEP in particular to data plane programming as well as to Network Function Virtualization.

As introduced in Section 2.6, P4 (Programming Protocol-independent Packet Processors) [BDG+14] specifies a high-level language for network programming that is not tied to fixed packet header definitions. The P4 compiler maps generic control programs to specific hardware or software platforms of target switches. Thus, P4 is able to fully exploit the capabilities of individual switch hardware, e.g., ASIC, NPU, or FPGA. However, switches in P4 do not take control decisions but merely execute control logic that has been compiled down from a high-level description, i.e., deploying control decisions instead of distributing decision making. Furthermore, P4 does not address the question where its compiler is actually executed, overall showing that control plane distribution is not yet considered.

Recent NFV-related SDN approaches typically focus on the distribution of network functions onto the switch data plane (see Section 2.6.2), like the generic frameworks OpenBox [BBHH16] and NetBricks [PHJ+16], or the management and orchestration of virtualized network functions (vNFs), like the E2 [PLH+15] framework, which handles the dynamism, placement, and chaining of vNFs. ZeroSDN is mostly complementary to NFV. While it also supports the implementation of dynamic network functions in the switch data plane[2], for instance implementing a stateless firewall with the AFC, with LDPEP, ZeroSDN rather focuses on the distribution of control plane functions for fast adaption, as shown with the intermediate local fail-over procedure or local load balancing. Contrasting OpenFlow and P4, ZeroSDN thus incorporates local control decision making, rather than a mere local installation of remotely taken control decisions.

4.3.4 Challenges of Deployment on Networking Hardware

In the following, we address challenges of deploying LDPEP on recent OpenFlow-enabled hardware switches. More specifically, we describe how switches without control plane

[2] In principle, LDPEP can implement arbitrary packet processing—in the control plane. For a detailed discussion on LDPEP's generalization to arbitrary slow-path packet processing, we refer to related work of ours [KDBR17].

access can be integrated in ZeroSDN nonetheless, as well as how we assure a safe control plane operation in case of an accessible control plane.

4.3.4.1 Migration and Closed Switch Hardware

In order to be able to run controllets locally, the switch's control plane has to be accessible, which is a defining property of white-box switches, as introduced in Section 2.5. For switches with an inaccessible control plane or insufficient resources, we provide a fallback mechanism that enables integration in our architecture. Such a switch is coupled with a dedicated SwitchAdapter, which instead of running locally is running on any other hardware, preferably in close proximity to the switch, via an OpenFlow connection and acts as a gateway to the switch in the message bus. Note that an external SwitchAdapter is still capable of executing local logic, yet additional network latency is incurred. We determine the penalty of externalizing the SwitchAdapter in Section 4.4.2.

4.3.4.2 Isolation and Resource Control through Lightweight Virtualization

The accessibility of the control plane is white-box switches' boon and bane: it allows arbitrary processes of different provenance to run in a less controlled environment, opposed to the closed switch model of traditional full-stack vendor implemented proprietary switch platforms. This raises concerns regarding security and reliability. (Unintentional) adverse behavior of control plane processes, including failures and excessive resource consumption, could starve other essential processes and thus poses a severe threat to its entire operation. For instance, in case of starvation of the OpenFlow agent, which is the sole interface to the underlying switch silicon in OF-switches, the control plane would be unable to detect and thus properly react to any data plane event, for instance, port state changes the in case of link failures. Also, the network's administrative domain might differ from the origin of the control application code. For instance in NFV, the network operator typically differs from the vendor of a virtualized network function (vNF), requiring trust in the code issuer and functional correctness of the VNF. In case of network virtualization, where tenants are provided logical partitions of network resources, the origin and behavior of control logic might not even be known to the network operator.

Consequently, we derive two requirements for the practical deployment of LDPEP: (1) **Isolation** to protect processes' data from each other and ensure data integrity, as illustrated in Figure 4.5(a), and (2) **Prioritization and resource control** to ensure liveness of control operation and thus ultimately network operation, as illustrated in Figure 4.5(b).

(a) Isolation of switch-local control processes (b) Fine-grained control resource control (RAM and CPU appropriation)

Figure 4.5. Requirements for safe control plane operation with multiple LDPEP controllets, ensuring data integrity and liveness of control plane applications.

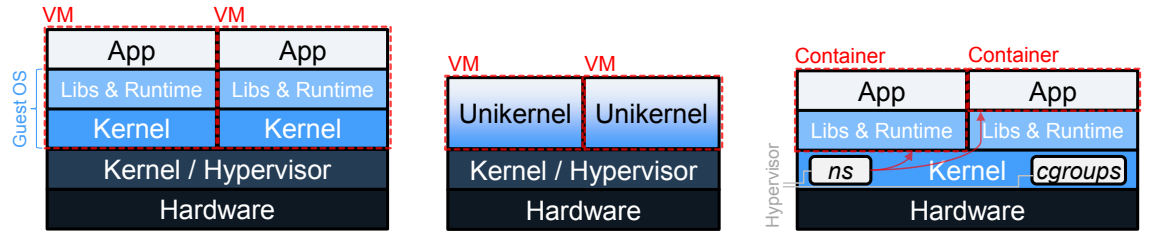
Moreover, the current white-box switch landscape exhibits a high heterogeneity with respect to hardware, i.e. switch silicon and control plane architecture (x86, ARM, PowerPC), and software, i.e. operating systems and forwarding agents, as described in Section 2.5.

We combine local controllets with lightweight virtualization to cope with white-box networking heterogeneity and to achieve required isolation properties.

In traditional *virtualization*, the emulation of resources provides isolation while their allocation to a VM depicts resource control. Since the large overhead of virtualizing a full OS along with an application, as employed with traditional virtual machines (see Figure 4.6(a)), counteracts the latency gains of local logic, we focus on using lightweight techniques. Two ways to counter virtualization costs (image size, memory footprint, and boot time) have been evolving: (1) stripping down the guest OS to a bare minimum, i.e., providing just the functionality the virtualized application needs for its operation (*library OS / Unikernel*, see Figure 4.6(b)) and (2) abandoning hardware emulation and full OS virtualization in favor of using isolation features of a shared kernel, providing multiple isolated user-space instances (*Container*, see Figure 4.6(c)).

Unikernels naturally lend themselves to cloud computing and NFV where they have been gaining importance in recent years, as for instance with ClickOS [MAR+14], a minimalistic, virtualized operating system for network processing. Unikernels have a single address-space. Kernel and application are a single, unified process. This eliminates the need for context-switches, but also prevents usage of multi-processing, signals, dynamic libraries, and virtual memory. Furthermore, application logic has to be ported to a particular Unikernel framework. However, Rump kernel [KC] uses NetBSD's kernel and libc. Thus, POSIX-compliant applications obeying these restrictions are supported without modifications.

In our scheme, the main goal of virtualization is to protect the control plane from unin-



(a) Traditional virtual machines based on a Type-1-Hypervisor

(b) Lightweight Unikernels/Library-OS approach

(c) Lightweight Linux Containers based on modern Kernel features

Figure 4.6. Comparison of traditional heavyweight and lightweight virtualization techniques, implementing required isolation and resource control requirements.

tentional adverse behavior of controllets which justifies lessening the isolation requirement to some degree. Containers allow for fine-grained control over the scope of isolation and resource allocation with almost no additional overhead. They rely on *namespaces*, a feature of the Linux kernel that isolates system resources, e.g., user and process IDs, IPC, filesystems and networking, of a set of processes whose resources are accounted using the control groups (*cgroups*) kernel feature. Two well-known implementations of container libraries are LXC and Docker. Besides the kernel, which is necessarily shared among all containers, a container configuration can share or isolate any combination of namespaces.

We evaluate the overhead of these techniques with given levels of isolation in our evaluations in Section 4.5.3.

4.4 Implementation

We have implemented an open-source prototype of our distributed SDN controller architecture, consisting of a modular execution framework (ZMF) running a distributed SDN controller application (ZSDN) with essential controllets atop. ZMF and most modules are written in C++, but we also provide a Java-based module framework (JMF). We provide build support for x86 and ARM architectures. This section presents the most important aspects of our implementation.

4.4.1 ZMF: The Zero Module Framework

Our micro-kernel implementation consists of two components, the `PeerDiscoveryService` and `MessagingService`. Module runtime environments are completely

decoupled and independent of each other. They run in dedicated processes, possibly on separate hardware. The `PeerDiscoveryService` implements module discovery with dependency and life-cycle management, enabling bootstrapping and peer dynamics. To this end, changes in a module's lifecycle state, such as joining/leaving the framework, are propagated using efficient UDP multicast. Furthermore, modules periodically confirm their state by multicasting heartbeat messages. Thus, with linear message complexity, each module knows the type and state of all other (relevant) modules.

For the message bus, we employ the production-grade low-latency communication middleware *ZeroMQ* (ZMQ) [iC]. Besides numerous communication patterns and transport mechanisms of varying reliability, ZeroMQ comes with a security framework implementing authentication, confidentiality, message integrity, etc. [iMa]. Access to the message bus is provided to ZMF modules through the `MessagingService`. We use TCP and IPC as reliable transport mechanisms. Later, we will show the mapping of data plane events and control plane events to pub/sub topics.

4.4.2 ZSDN: A Distributed SDN Controller

ZSDN consists of prototypical controllets for distributed SDN control. All controllets support OpenFlow (OF) 1.0 and 1.3. Common data structures like topology data are mapped to *Google Protocol Buffers* [Goo] definitions, providing language-independent module communication.

Figure 4.7 shows essential controllets and their logical interdependencies. The SwitchAdapter (SA) wraps an OF-enabled switch in an instance which is running locally on the switch, integrating it to and representing it within the framework.

State controllets acquire data plane state by passively reacting on subscribed events or

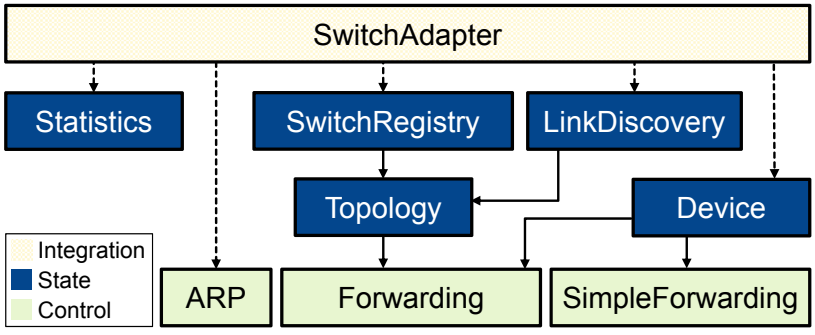


Figure 4.7. Dependency graph for essential controllets

active probing. For instance, the SwitchRegistry registers all available switches through subscriptions on changes of their representing SwitchAdapters, whereas the LinkDiscovery controllet detects switch to switch links by subscribing to LLDP (Link-Layer Discovery Protocol) data plane events and proactively injecting LLDP packets over the SA instances into the data plane. The Topology controllet subscribes to both, SwitchRegistry and LinkDiscovery events, such that eventually it holds complete topology knowledge, excluding end-systems, which are managed by the Device controllet. Topology information can be actively queried by controllets using req/rep. Topology changes are published through events, allowing for passive synchronization of controllet-local caches. Another module class provides control feedback to the data plane and thus closes the network control loop by modifying forwarding rules, such as the SimpleForwarding controllet.

4.4.2.1 Event Space – Topics Mapping

Due to the lack of practical high-performance content-based pub/sub middleware implementations, we use ZMQ's topic-based pub/sub implementation instead. We map the event space of both, data plane events (from SA) and control plane events (other controllets), to topics employing a hierarchical topic scheme which allows for fine-grained subscriptions. In the following, we describe the mapping, while illustrating its usage on the example of a SwitchAdapter.

Each controllet defines two sets of *topics*: Set `TO` defines which message types (topics) a controllet is able to process, i.e., which data plane events it wants to receive from the message bus. This set is mapped to corresponding subscriptions for event filtering. Set `FROM` defines the topics published by the controllet, i.e., events disseminated to the bus. Other controllets can subscribe to these advertised topics. Topic definition is strictly hierarchical. The first hierarchy layer defines the type of declaration (`TO` or `FROM`). The second layer comprises the identity of the controllet. All upper layers contain structure of controllet-type specific content. Attributes are encoded as a bit-sequence, with a specific length associated to each hierarchy layer, at a specific location within the topic-hierarchy. Wildcard matching ("`?`") is supported. For the SA, as shown in Listing 4.1, the semantics are as follows: Listens to Events (`TO`): The SA will receive any incoming message of these topics and forward it to the switch. Publishes Events (`FROM`): any OF message the SA receives from the switch is published using a corresponding topic within this set of topics.

Listing 4.1. Excerpt of the SwitchAdapter topic-hierarchy

```
TO=0x01
 SWITCH_ADAPTER=0x0000
  SWITCH_INSTANCE=0x???????
   OPENFLOW=0x00
    FEATURES_REQUEST=0x05
    PACKET_OUT=0x0D
    FLOW_MOD=0x0E
    ROLE_REPLY = 0x19
    METER_MOD = 0x1D
 ...
```

```
FROM=0x02
 SWITCH_ADAPTER=0x0000
  -
   OPENFLOW=0x00
    FEATURES_REPLY=0x06
    PACKET_IN=0x0A
     LB_GROUP=0x?? default=0x00
      IPv4=0x0800
       TCP=0x06
       UDP=0x11
    PORT_STATUS=0x0C
...
```

4.4.2.2 Partitioning & Load Balancing

Note that hierarchy layers are not tied to a fixed representation of the underlying event space, e.g., SA topics are not restricted to directly reflect OF-matching fields. Artificial hierarchy layers may be freely introduced between any layers. For instance, to enable load balancing of PACKET_IN messages, the SA artificially discriminates PACKET_INs by introducing an additional 1-Byte topic hierarchy layer (LB_GROUP) and disseminating such events in a round-robin fashion to the set of groups. Controllets participating in load balancing subscribe to a specific LB_GROUP, whereas controllets that want to receive all PACKET_INs apply a wildcard subscription on the LB_GROUP layer. This mechanism enables partitioning along the network topology where, for instance, Topology controllets refine their subscriptions to certain groups.

4.4.3 Integration Schemes for LDPEP

One way to implement switch-local control is to identify a set of essential controllets and run them locally on each switch. That way, full modularization is maintained and the controllets' code can be directly reused. While highly scalable, communication over the message bus, e.g., for querying topology data in case of the AFC (see Section 4.3.2.1), incurs higher latency compared to, e.g., direct memory access in case of a single-process integration. Although TCP connections over the local loopback interface are highly optimized in recent Linux kernels, micro-benchmarks [SMS+12] executed on our testbed

nodes[3] and switch[4] (see Sections 4.5.1 and 4.5.3) indicate higher throughput and lower latency when using inter-process communication (IPC) mechanisms.

When focussing on latency, LDPEP should be implemented by a fully integrated, monolithic controllet connected to the message bus in order to leverage the global view and central coordination, as explained for the AFC. Performance potential lays in a tighter coupling to the underlying switch hardware. Ideally, local logic would be pushed down to the data plane hardware using data plane programming, which however focuses on packet processing and thus is not suited to implement arbitrary control logic.

By supporting multiple integration schemes, our architecture offers great flexibility to network operators who have to compromise between performance and implementation efforts, based on the expected load. We have implemented the schemes modularized (*ZSDN-TCP*, *ZSDN-IPC*) and fully integrated (*ZSDN-AFC*) and compare their performance in the following.

4.5 Evaluation

In this section, we present the evaluation of our proposed distributed SDN controller architecture, consisting of a raw performance comparison, an analysis of the scalability of our approach, as well as results from the deployment on our white-box networking switch, including the overhead of virtualization.

4.5.1 Raw Controller Performance

First, we asses the raw performance of ZSDN and conduct a comparison with popular non-distributed or less distributed controller frameworks using the following **methodology**.

We use cbench [SY] for measuring controller throughput and latency. Cbench emulates switch behavior by sending `OF_PACKET_IN`s (triggers) to the connected controller. To measure throughput, cbench sends triggers as fast as possible and averages over the number of received `OF_FLOW_MOD` and `OF_PACKET_OUT` from the controller. To prevent double-counting, we modified the processing of controllers to respond with only one type of message. For sequential throughput, cbench waits for a response to a sent trigger, before sending the subsequent trigger. Hence, we approximate controller

[3] https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/details/tmpFxslu8.html

[4] https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/details/tmpgtzzD3.html

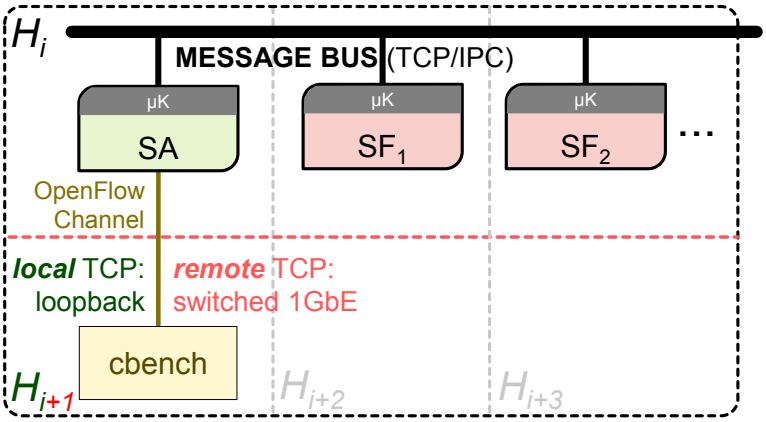


Figure 4.8. Evaluation setup of control plane (testbed) for raw controller performance evaluation (Section 4.5.1) and scalability evaluation (Section 4.5.2).

processing latency as the inverse of sequential throughput. Cbench and the controllers run on a testbed consisting of 12 nodes (Intel Xeon E3-1245v2 @ 3.4 GHz, 4 physical cores, 16 GB RAM) interconnected through a switched GbE network.

To investigate the impact of controller locality, as illustrated in Figure 4.8, we differentiate between the switch (cbench) and the controller (traditional SDN controller or ZSDN) running on the same node H_i (*local*; OpenFlow channel: TCP over loopback interface) and running on different nodes H_{i+1} and H_i (*remote*; OF Channel: TCP over switched Ethernet).

Each cbench run is averaging over 60 seconds in parallel on each of the 12 nodes (*local*) and 120 seconds on each of the 6 node pairs (*remote*), totalling in the aggregation of 12 minutes of observation time for each experiment.

We evaluate the following platforms:

(1) ZeroSDN ZSDN-TCP/IPC: modular controller framework using single-instance controllets with message bus communication using reliable (guaranteed and in-order message delivery) transport mechanisms TCP and IPC (UNIX domain sockets)

(2) ZeroSDN ZSDN-AFC: the Autonomous Forwarding controllet employing local data plane event processing (LDPEP), as introduced in Section 4.3.2.1, fully integrated (single process, see Section 4.4.3)

(3) NOX (verity) [GKP+08, Theb]: an early academic C++ implementation, popular for its performance

(4) ONOS [Thea]: Java-based, carrier-grade controller framework, modularized using OSGi [OSG]

(5) Floodlight [Big]: Java-based, production-grade controller framework, forked from

the academic Beacon controller

(6) Ryu [Ryu]: Python-based controller framework, popular for its support of most recent OF versions

Figure 4.9 shows the **results** of the controller comparison, where error bars depict the standard deviation. Regarding *local* **throughput** (Figure 4.9, top half), NOX performs best with $\approx 369 \pm 2$ msg/ms (messages per millisecond). The LDPEP of ZSDN-AFC results in similar figures with $\approx 260 \pm 1$ msg/ms.

The performance penalty of distribution shows to be bearable: distributed ZSDN throughput is about 53% of ZSDN-AFC ($\approx 138 \pm 28$ msg/ms), mainly dedicated to message passing. Note that here we ran only one instance of each controllet, thus measuring only the costs of distribution, not its benefits, which we measure in the next section. Interestingly, ZSDN throughput decreases by $1/3$ when using IPC instead of TCP. This contradicts expectations risen through the micro-benchmarks, where UNIX domain socket throughput was reported to be about 20% higher than TCP on these nodes (see Section 4.4.3). While Floodlight is close to ZSDN-IPC, ONOS performs slightly better. The Python-based controller Ryu is far off with ≈ 0.8 msg/ms. Overall, throughput penalties for a *remote* OF connection are moderate. Interestingly, the remote throughput of ONOS and Floodlight are measured to be higher than their local throughputs, which we could trace down to stem from their common Java-based network framework (Netty). Note that although the control load, i.e., event rate, in practical deployments can be expected to be smaller than in our maximum throughput evaluation, our results provide valuable insights in determining the upper performance bound.

Looking at **latency** (Figure 4.9, bottom half) however, *remote* latency is increased drastically compared to *local* latency with factors of 2 (ZSDN-TCP) to 6 (ZSDN-AFC). This is a strong argument for local processing, especially for the integrated LDPEP mode. On the other hand, when using modularized controllets, the penalty for running SAs remotely, e.g., for migration or inaccessible control planes (see Section 4.3.4.1) is bearable.

4.5.2 Scalability of Controllet Distribution

Next, we evaluate the benefits of distribution and replication of ZSDN controllets with the following **methodology**. As illustrated in Figure 4.8, we distribute the most compute-intense controllets SwitchAdapter (SA) and SimpleForwarding (SF) to dedicated nodes. For the moment, we use only a single SA instance (replication factor $k=1$) placed at H_i (*local*) or H_{i+1} (*remote*). Furthermore, we replicate the SF with a varying replication

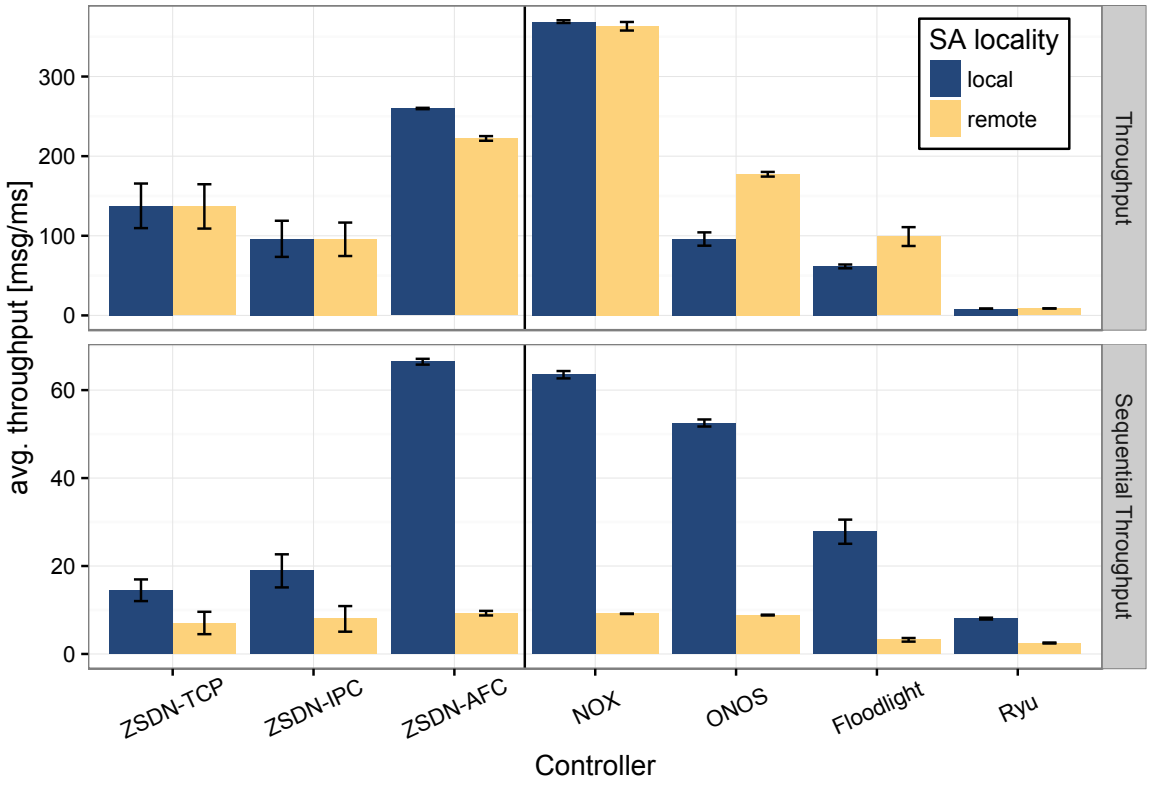


Figure 4.9. Raw controller performance: comparison of throughput and sequential throughput (inverse latency) for ZSDN and other popular controllers, drilled down by switch-controller locality (*local*/*remote*).

factor n. Each instance SF_j with $j \in [1,n]$ is placed on a dedicated node (H_{i+1+j}). The SA distributes the total load evenly to these instances (see Section 4.4.2.2). We additionally vary the number of switches s, cbench emulates. For each switch connection, the SA spawns 4 threads, dedicated to that connection.

The **results** are shown in Figure 4.10. Even for $n = 1$ (no SF replication), we achieve 15% higher **throughput** just by placing the SF instance on a dedicated node. For $s = 1$ and increasing n, throughput increases, but only sublinearly. In this setting, the SA constitutes a bottleneck. It maxes out 1 thread (per-core performance) and is not able to fire drastically more data plane events which the SF instances could process. If we increase s, the throughput increases almost linearly until the (single) SA instance maxes out (per-CPU performance (all cores)) at $s = 3$. Having $n > 3$ does not further improve performance, such that the overall peak performance is reached with $s=3, n=3$ at $\approx 280 \pm 16$ msg/ms. Note that maxing out introduces high indeterminism (e.g., apparent throughput drop in the graph). To investigate the scale-up behavior, we repeated the experiment using more powerful nodes (Xeon E5-1650v3 @ 3.5 GHz, 6 physical cores), where throughput peaks at $s=5, n=5$ with $\approx 670 \pm 20$ msg/ms. Note that more compute-intense processing, such as deep packet inspection, would much more benefit from distribution and replication.

For $n = 1$, **latency** increases by about 22% due to the physical separation of SA and SF and thus one additional hop over the control plane network. In contrast to *remote*, *local* latency increases with increasing s, since SA and cbench are running on the same host, thus sharing an increasingly loaded CPU, slowing down cbench's production of emulated data plane events and SA's distribution pace. With increasing n, *local* latency increases as well, due to increased splitting and merging efforts of events to be disseminated to the message bus and reactions received over the bus. *Remote* latency follows the same trend, but only slightly increases with increasing n since the impact on latency of the OpenFlow TCP connection over the physical network instead of the loopback interface is the dominating factor. In more practical scenarios with lower event rates but higher packet sizes, this effect is expected to be of a much lesser extent.

When using the full distribution capabilities by replicating both, SF and SA, i.e., increasing n as well as k, and keeping n/k balanced, we could verify **linear scalability**. Depending on the efficiency of group communication, which is very efficient in ZMQ due to filtering right at the publisher, network saturation limits scalability. For scenarios with such high event rates however, it is reasonable to employ 10GbE or higher on the control plane, counteracting network capacity bottlenecks.

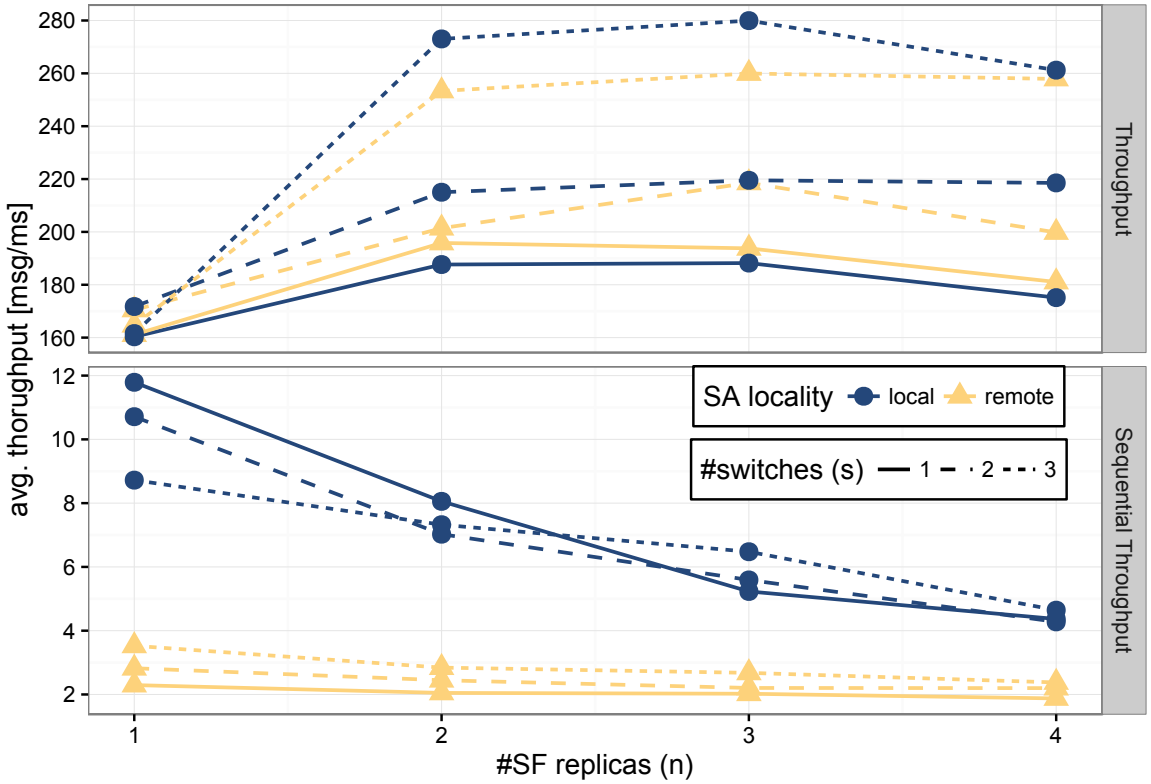


Figure 4.10. Scalability evaluation (scale-out): modular ZSDN controller with varying number of SimpleForwarding instances (n) and varying number of connecting switches (s).

4.5.3 Performance on White-box Networking Switch Hardware

In our last evaluation, we compare the controller performance on real white-box switch hardware, instead of emulating it. We compare ZSDN-IPC, ZSDN-AFC, NOX, and Ryu. The **device under test** is a typical top-of-rack white-box switch (cf. Figure 2.7, p. 55) Edge-Core AS5712-54X, whose hardware specification is publicly available under the Open Compute Project [Opea]. Its control plane comprises an x86 Intel Atom CPU (C2538) [Cor] with 4 cores at 2.4 GHz, 8 GB RAM, and a GbE NIC. Atop we run the operating systems *OpenNetworkLinux (ONL)* 2.0 with a 3.16.39-LTS kernel and *Pica8 PicOS* 2.8 with a 3.16.7 kernel. On the data plane, it features a Broadcom Trident II ASIC with 48 × 10GbE and 6 × 40GbE ports.

Our **methodology**, illustrated in Figure 4.11, is as follows. The switch runs a *network operating system (NOS)* on its control plane. For *remote* performance, the OpenFlow agent running atop of the NOS connects to a remote controller. For *local*, we deploy a controller directly on top the NOS, which the OpenFlow agent connects to. Later, we isolate the local controller using a hypervisor or container. We intentionally provoke that every ingress packet at a data plane port is processed in the control plane. To this end, the controllers run learning switches, but do not install flows. In the data plane, we connect two end-systems (hosts H_0, H_1) with 10GbE links to the switch. H_0 is sending packets to H_1 where they are reflected back. Packet identity is ensured through unique sequence numbers attached to the packets (as sole payload). Both, egress (t_{TX}) and

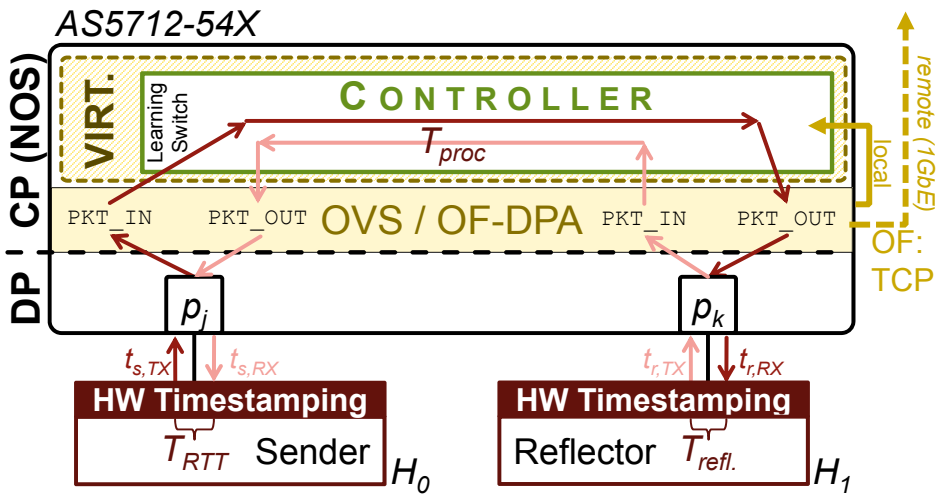


Figure 4.11. Setup of measuring control plane processing latency (T_{proc}) on a white-box switch (see Figure 2.7) using hardware-timestamping on end-systems.

ingress (t_{RX}) times are captured using hardware-timestamping. Thus we can measure the RTT at the sender ($T_{RTT} = t_{s,RX} - t_{s,TX}$) and the time spent for reflection at the reflector ($T_{refl} = t_{r,TX} - t_{r,RX}$) with high precision. We approximate the (one-way) switch processing latency as $T_{proc} = 1/2 * (T_{RTT} - T_{refl})$, neglecting transmission and propagation delay.

The **throughput results** show significant throttling. This is expected behavior, although the switch silicon is interconnected with the CPU over a PCI-Express bus with plenty of bandwidth. Switch vendors introduce limiting of traffic between the switch silicon (data plane) and the CPU (control plane) in their switch design, to prevent denial of service attacks on the control plane caused by (uncontrolled) data plane traffic. We observe that with ONL, throughput is capped at 1 kpps (1000 packets per second) with a low peak CPU utilization of the ONL's OF-DPA daemon of 50% of one core. This shows that the rate limit is clearly not caused by a CPU bottleneck. For switch-ingress rates $\geq$ 20 kpps on PicOS, we measure an egress-rate of about 7 kpps, while PicOS's Open vSwitch daemon consumes two cores. Note that due to isolation and resource control, our lightweight virtualization deployment effectively protects control plane operation, rendering additional safety mechanisms like the rate-limiting superfluous.

For evaluating switch control plane **processing latency**, we send with a rate of 100 kpps for 50 s. Through reflection, the effective packet rate (ingress rate at the switch) is doubled. We begin with a **comparison of controllers** (Figures 4.12 and 4.13) running bare-metal (no hypervisor/container) on ONL or remotely (Xeon E5-1650v3).

For switch-*local* controllers, NOX performs best with an average latency of $\approx 330 \pm 75\,\mu s$. ZSDN-AFC is consistently within 3% of NOX. The costs for module decoupling over the message bus of ZSDN-IPC evaluates to $\approx 767 \pm 112\,\mu s$, a factor of 2.3, clearly showing the superiority of LDPEP with the full integration scheme of ZSDN-AFC which still profits fully from centralized view and coordination. *Remote* execution increases latency by a factor of ≈ 1.8, for ZSDN-AFC and NOX and a factor of ≈ 1.4 for ZSDN-IPC. Note that our scenario of a one-hop switched GbE control network is almost ideal, providing a lower bound for switch control plane processing latency. For larger distances or WAN scenarios, remote control latencies are expected to be orders of magnitudes higher. Ryu (Figure 4.14) is expectedly performing worse than its C++ counterparts with $\approx 1795 \pm 225\,\mu s$ but surprisingly 20% faster remote execution.

We evaluate **virtualization overhead** of *Docker* (ONL only) as well as rump kernels (*rumprun*) and full VMs (*KVM*), both running on QEMU with KVM-acceleration enabled by the Atom's VT-x [UNR+05, Cor] support. The baseline is bare-metal execution (*none*). Since NOX and ZSDN are relying on Digital Shared Objects (DSO), we were not able to

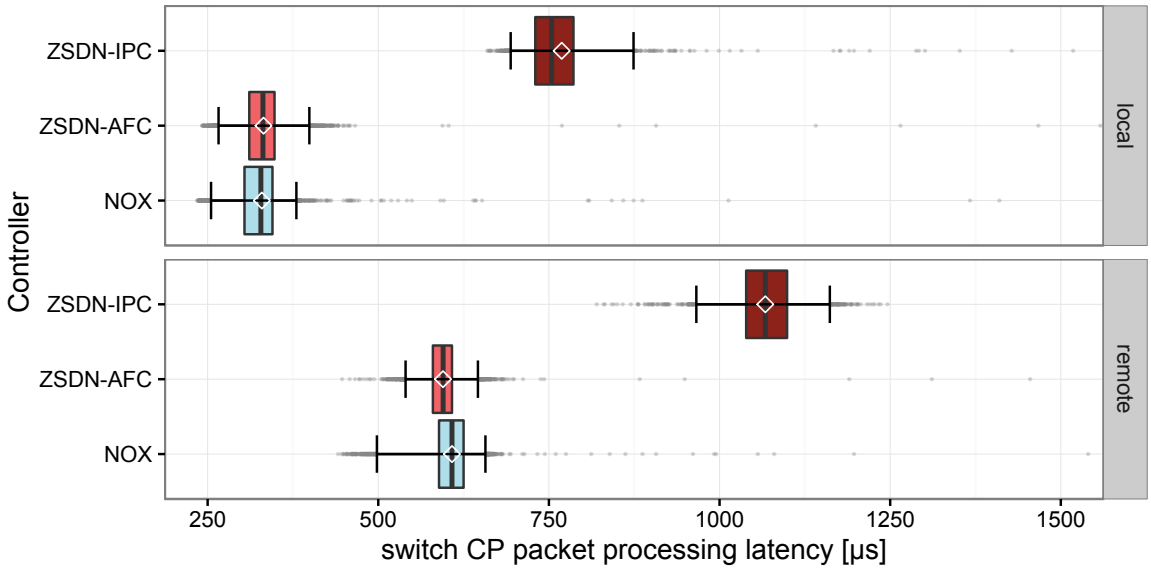


Figure 4.12. Evaluation on a white-box networking hardware switch: comparison of switch processing latencies (x-axes: medians (bars), averages (diamonds)) of modular ZSDN-IPC, fully integrated ZSDN-AFC, and NOX, drilled down by switch-controller locality.

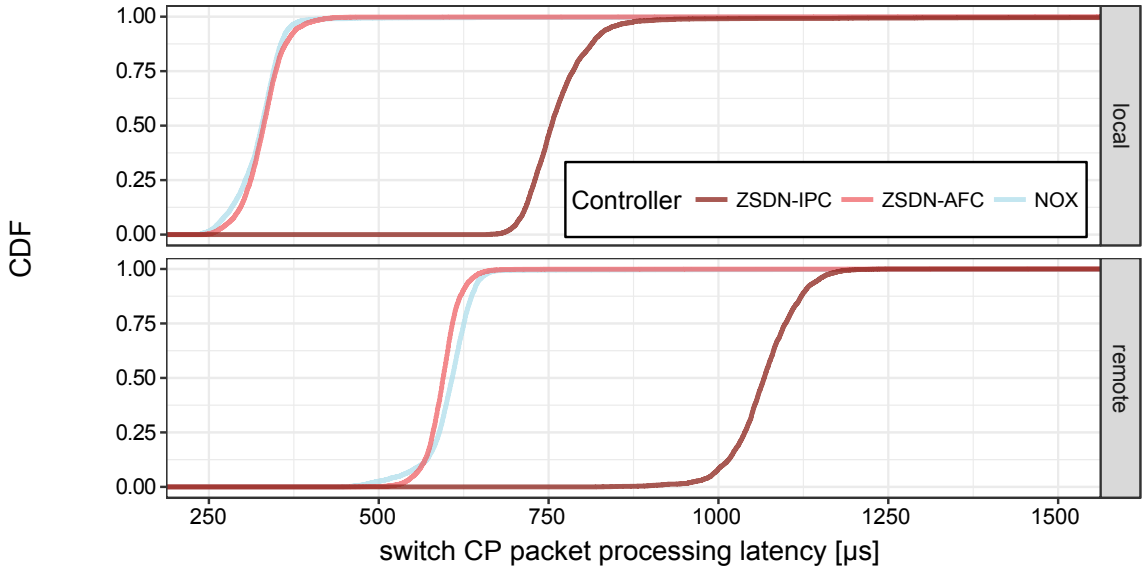


Figure 4.13. CDFs (y-axes) of switch processing latencies (x-axis) of modular ZSDN-IPC, fully integrated ZSDN-AFC, and NOX, drilled down by switch-controller locality.

port them to rumprun. The results are illustrated in Figures 4.14 and 4.15. **Docker** has the lowest overhead. With full isolation of all but the network namespace, Docker imposes almost no overhead for NOX/ZSDN. Latency and its deviation are within 1 µs to bare-metal execution. Ryu on Docker results in about 80 µs additional latency. Additionally isolating the network namespace (undepicted) incurs 100 µs additional latency which can be mostly attributed to the Linux network bridge, as we describe later.

Next, we measure the combined overhead of the hypervisor and the guest OS of virtualization variants, employing KVM's pseudo-paravirtualized network driver *virtio*. **Full virtualization** (*KVM*) adds large overhead. On average, 410 µs (factor 1.5) incur for ONL and 820 µs (factor 1.8) for PicOS, both slower than remote execution. Standard deviations are larger by factors 2 and 2.2, respectively. Ryu as a **Unikernel** (*rumprun*) is showing much better results. Compared to a full VM, latencies and deviations are greatly reduced by 220 µs and 285 µs for ONL, and 310 µs and 190 µs for PicOS. This is the result of the minimal guest OS and hence reduced OS overhead. Compared to bare-metal/Docker, factors of 1.2 and 1.5 for latency on ONL and PicOS are promising. We find that the standard Linux bridge in use at least partially accounts for the larger overhead. By using optimized software bridges, like macvlan, VALE or OVS in combination with hardware virtualization support such as SR-IOV [PLZ+15, KJ17] (paravirtualization), latencies as small as 45 µs [MAR+14] have been achieved—on server hardware, though.

Lastly, we evaluate the difference between the **NOSes**. For all measurements, compared to ONL, PicOS adds quite consistent latency of 200 µs on average for bare-metal and remote, 420 µs for NOX/ZSDN, and as high as 950 µs for KVM and rumprun. Especially for higher packet rates, we have observed instability of QEMU on PicOS. One explanation for the discrepancy may lay in the forwarding agent. While OF-DPA (used in ONL) tightly reflects the underlying switch silicon hardware pipeline, which is quite restrictive (see Section 2.2.1), OVS (used in PicOS) offers an abstracted and seemingly unrestricted multi-table pipeline as per the OpenFlow 1.3 specification. The implementation of this mapping is expected to cause additional latency as a rather static offset, as we observed.

We can conclude that containers provide isolation as needed at minimal cost. We could verify and quantify the benefit of reduced latency to be almost halved with containerized local control logic, despite isolation. Note, that our scenario of a one-hop switched GbE control network, is almost ideal, providing a lower bound for switch control plane processing latency. For larger distances or WAN scenarios, remote control latencies are expected to be orders of magnitudes higher, even compared to local yet sub-optimal virtualization variants.

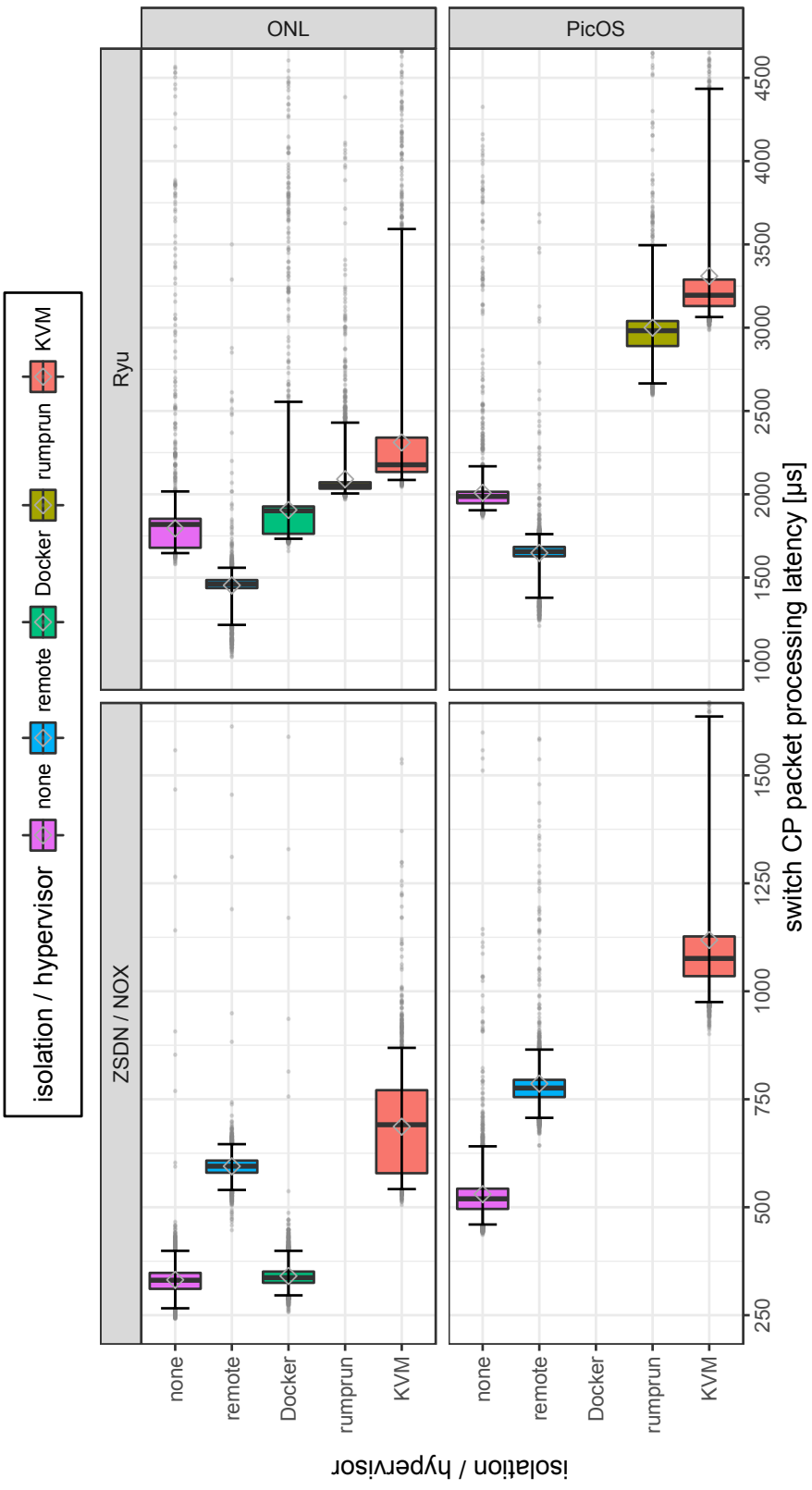


Figure 4.14. Processing latencies (x-axes: medians (bars), averages (diamonds)) of controllers (grid horizontal) running locally on network operating systems (grid vertical) with varying isolation mechanisms (y-axis). Bare-metal (*none*) and *remote* execution are given as baselines. Whiskers enclose 95% of measured values.

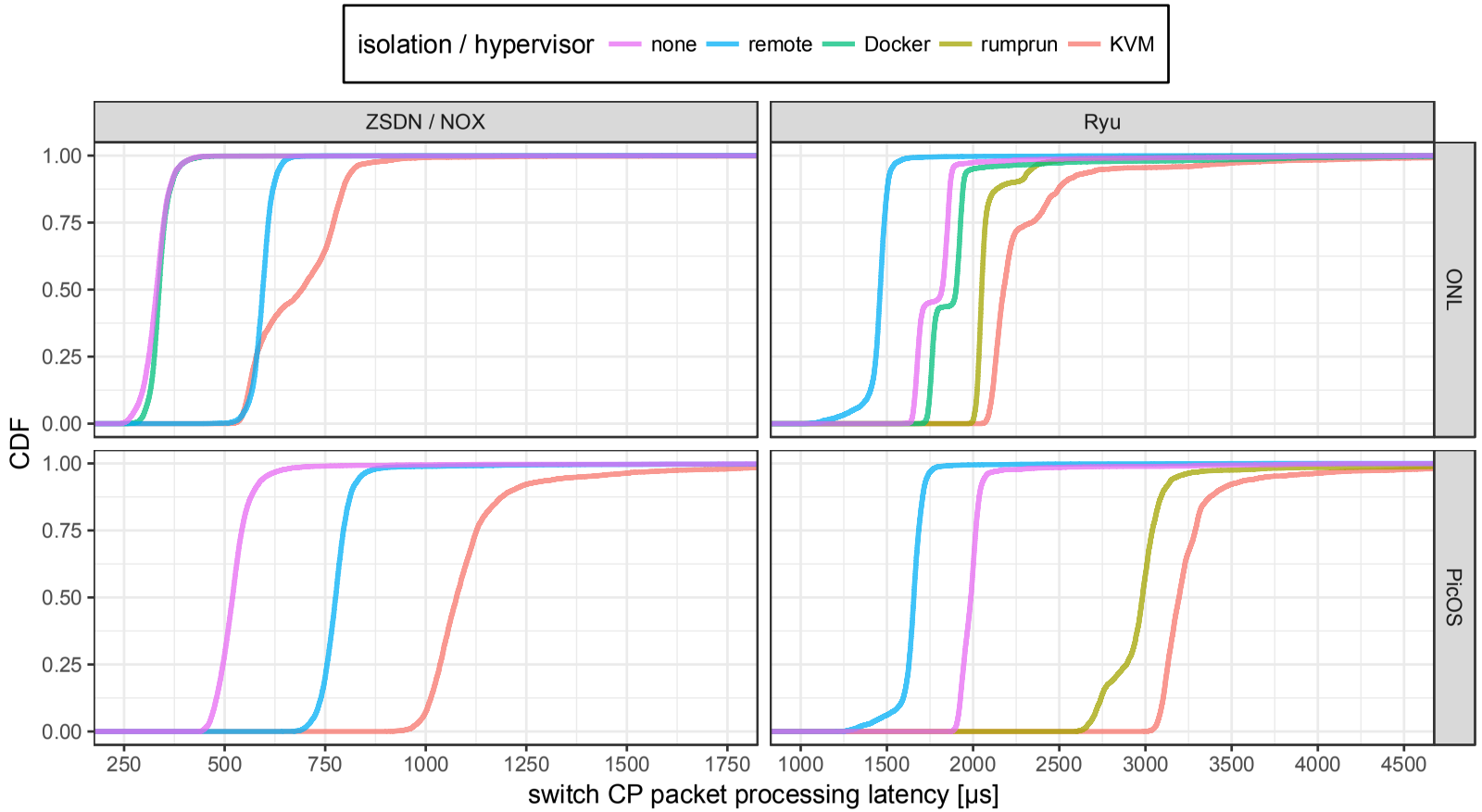


Figure 4.15. CDFs (y-axes) of processing latencies (x-axes) of controllers (grid horizontal) running locally on network operating systems (grid vertical) with varying isolation mechanisms.

4.5.4 Evaluation Conclusions

To conclude the results of our evaluations, we first can state that in a raw comparison to other controllers, our local data plane event processing (LDPEP) approach ZeroSDN-Autonomous-Forwarding-controllet (ZSDN-AFC) reaches a throughput of $\approx 260\,\mathrm{msg\,ms^{-1}}$ (messages per millisecond), which is about 70% of the best-in-class controller NOX. The costs for distribution with ZSDN was measured as a throughput penalty of 53% in comparison to ZSDN-AFC. Regarding latency, remote processing is drastically inferior to local processing, as expected. Local processing yields factors 2 for ZSDN-TCP and 6 for ZSDN-AFC of reduced latency compared to remote processing. Overall, this is a strong argument for local processing and LDPEP, which combines the advantages of local processing with aggregated global knowledge while having efficient access to remote controllets.

After observing ZSDN's throughput penalty due to the worst-case scenario, where all controllets are running on the same node, bearing just the cost but not using the benefit of the distributed architecture, we investigated the scalability of ZSDN when actually distributing. Thus, we distributed the most compute-intense controllets, switch adapter (SA) and forwarding (SF) to dedicated nodes. Not distributing the SA controllet in the beginning, we verified expectable scaling behavior (scale-up) of the SA controllet, scaling-up from $\approx 280\,\mathrm{msg\,ms^{-1}}$ to $\approx 670\,\mathrm{msg\,ms^{-1}}$. Distributing both, SA and SF (scale-out), we could verify linear scalability of throughput until saturation of the GbE control plane network.

The performance evaluation on actual switch hardware (a white-box switch) first of all revealed an excessive throughput throttling due to an intentional security mechanism to protect control plane operation. Interestingly, our concept of isolation and resource control renders this switch-vendor implemented mechanism superfluous. In terms of switch control plane processing latency, we could again verify the superiority of LDPEP. While NOX and ZSDN-AFC are about on-par with $\approx 330\,\mu\mathrm{s}$, the cost of decoupling controllets over the message bus evaluated to a factor of 2.3 (to $\approx 767\,\mu\mathrm{s}$) for ZSDN-IPC. Providing isolation and resource control using containers have shown to come at virtually no cost. The more heavyweight virtualisation techniques Unikernel and full hypervisor (KVM) introduce average factors of 1.2 and 1.5 additional latency on Open Network Linux (ONL), compared to containers and bare-metal execution.

4.6 Related Work

Many early approaches, including Onix [KCG+10], propose to externalize state storage, which incurs additional latency for lookups. In Onix, switches and controller instances are tightly coupled. While Onix limits the shared view onto network state information, HyperFlow [TG10], as our approach, holistically propagates all kinds of data plane events. HyperFlow also facilitates pub/sub to propagate events, event classification is however limited to three topics, whereas our approach leverages content-based filtering (mapped to a topic hierarchy in our preliminary implementation) to allow for fine-grained subscriptions. Furthermore, HyperFlow exclusively relies on passive synchronization of the locally cached network-wide view, while our approach offers maximum flexibility allowing both, local caches for fast access as well as access to highly consistent centralized storage.

DevoFlow [CMT+11] is the first SDN approach to allow for local decision making on the switch, however mandating changes of the switching ASIC. Kandoo [HYG12] proposes a two-layered controller hierarchy with a root controller maintaining network-wide state, and local controllers possibly running directly on switching hardware, only handling local events where no global knowledge is required. While this scheme allows for offloading of simple local logic, local controllers do not hold any state data, neither do they interact with each other at all. Our approach is not limited to such a strict hierarchical scheme and does not rely on a root controller instance, thus offering superior flexibility.

While these approaches exhibit a static switch-controller assignment, ElastiCon [DHM+14] allows for a dynamic switch to controller instance mapping. By periodic monitoring of controller load, the number of instances and the mapping is adapted for effective load balancing. Since switches are still tightly coupled to an instance, the authors introduce a switch migration protocol. A similar problem is addressed in [KCGJ14, BBHK15]. The decoupling of switch and controller offered by our approach eliminates the need for complex and costly migration mechanisms.

More recent approaches improve on fault-tolerance in control distribution. Beehive [YG16b] models control applications as centralized asynchronous message handlers featuring and thus focussing on application partitioning, exclusive handling of messages among a set of controllers, as well as consistent replication of control state information. Logical message propagation is dictated by map-functions that determine to which set of applications a specific message is to be sent to. Message passing is not addressed in detail. Furthermore, each Beehive controller instance contains all application logic in contrast to

our highly modular approach. Another work, Ravana [KZFR15], focuses on controller fault-tolerance. Ravana subsumes event dissemination from switches, their processing by a controller, and the resulting execution of controller commands at the switches in a transaction and guarantees that control messages are processed transactionally with exactly-once semantics. Message propagation and actual distribution schemes are not addressed.

Fibbing [VTVR15] exerts centralized control over routers that implement a legacy, non-SDN, control plane running fully decentralized routing algorithms, such as OSPF and IS-IS. The forwarding behavior of routers, i.e., their forwarding information base, is manipulated as to achieve desired network behavior by faking input messages to the distributed routing algorithms. Although being congruent in the notion of centralized control, unlike in our approach, Fibbing's control is solely indirect and thus inherently limited.

With respect to Network Function Virtualization, OpenBox [BBHH16], an SDN-based framework for NFs proposes to employ SDN-switches as packet processors in favor of middleboxes, however based on the assumption of a centralized controller. Moreover, E2 [PLH+15] proposes a framework for NFV applications, strengthening the role of SDN in NFV management through the unification of SDN and NFV in a single controller that automates NF-placement and service interconnection (management and orchestration). For extended network processing capabilities, they push richer programming abstractions into the network layer, however relying on software switches.

4.7 Conclusion

In this chapter, we presented a novel architecture for a highly flexible distributed SDN controller based on a message bus for communication and a micro-kernel design. Network control logic is split into control modules, called controllets, which can be freely distributed. Controllets communicate through the message bus and are decoupled from switches and other controllets using the publish/subscribe paradigm. The micro-kernel design only requires controllets to implement a small set of functions to connect to the message bus and participate in publish/subscribe communication. Consequently, controllets are extremely lightweight and can also be executed directly on white-box switches to enable fully distributed network control even without external SDN controller—a new level of flexibility in control plane distribution that so far is not possible with standard SDN controllers. Our evaluations showed the practicality of our architecture for both, full

distribution as well as the integration of controllets for fast local processing of data plane events, while still benefiting the from global view and centralized coordination. Through employing lightweight virtualization techniques, we cope with crucial challenges of practical deployment to ensure a safe operation of the control plane and thus continuous network operation.

5 P4CEP—A Data Plane Implementation of Complex Event Processing

As described in Section 2.6, data plane programming can be seen as an evolutionary step in SDN, introducing full protocol-independence and tremendously extending flexibility and capability with respect to packet processing, opposed to SDN's established de-facto standard data plane model of OpenFlow with its fixed parser and pipeline. This evolution has given rise to a recent trend in SDN, called *in-network computing* [SAA+17], which exploits data plane programming for offloading of application functionality from end-systems to programmable network elements while leveraging the performance of specialized forwarding hardware, capable of processing packets at line-rate throughput in orders up to Terabits per second with low latency. In-network computing has so far been proposed to support in particular distributed applications, ranging from consensus [DSC+15] and concurrency control [JdSM+18] over caching in distributed key-value stores [LLN+17], feedback control in a cyber-physical system [RGW+18] to aggregation functions in data-centric processing including machine learning and graph analytics [SAA+17]. It has been shown that in-network computing can yield significant performance improvements by increasing throughput, bandwidth-efficiency, or reducing latency.

In this chapter, we present how we employ in-network computing to offload *Complex Event Processing* (CEP), a representative of stateful processing from the domain of message-oriented middleware. Traditionally, CEP has been implemented as an overlay of software middleboxes (operators) inferring higher-level knowledge (complex events) by evaluating specific combinations of incoming information (basic events). Packets convey basic events, which are typically comprising structured low-dimensional data, such as sensor data, stock market values for high-frequency trading, or data of network management, such as intrusion-detection systems or anomaly detection. In-network computing is best suited for processing of small data encapsulated in packet headers. As described earlier in Section 2.6.2, the *middlebox model*, where packets are processed on remote

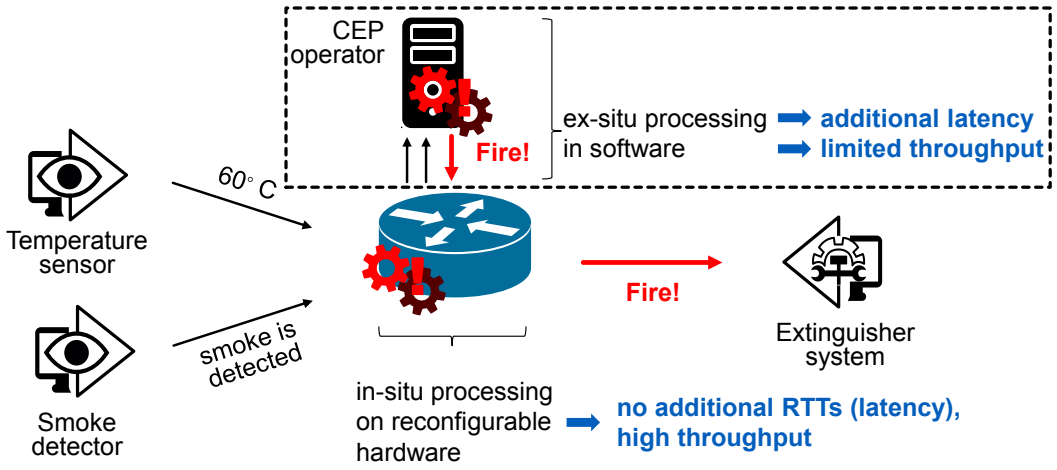


Figure 5.1. Comparison of *ex-situ* and *in-situ* processing in the case of Complex Event Processing in a fire-detection scenario. In-situ processing implements the CEP operator within programmable network elements, eliminating additional RTTs.

hardware appliances (middleboxes) or in virtualized environments on commodity server hardware (NFV), bears disadvantages: it increases network management complexity by the introduction of additional system components (remote, off-path entities) that can fail, and have to be managed (placement, dynamic configuration). By steering traffic through remote hardware, additional round trips are inherently inflicted, consequently increasing application latency. Thus, packets are ideally processed *in-situ* (see Figure 5.1) at high-performance network elements that they naturally traverse, consequently combining forwarding and processing, which resembles the rationale of in-network computing. Furthermore, the uniform interface of data plane programming, provided by the P4 language, greatly facilitates portability.

CEP is a well suited application to show challenges entailed in the distributed processing of in-network computing. Through the distribution of CEP operators, basic events may have to be propagated in a one-to-may pattern, raising the issue of update consistency of multicast trees, which we addressed in Chapter 3. Moreover, CEP heavily relies on state that is kept among arrivals of basic events (inter-packet state). In *stateful processing*, such an inter-packet state is used during processing as well as to store the output of the processing. Keeping inter-packet state data consistent is of utmost importance to guarantee the correctness of this class of applications. Stateful in-network computing is challenging since data plane programming is not yet particularly well-suited to ensure data consistency.

The contributions of this chapter are as follows: we present P4CEP—our work on an

in-network implementation of CEP, including a compiler from our P4CEP rule specification language to P4. Our design contains generic mechanisms for stateful processing, namely window-based aggregation functions (for reasoning over a window of events) and state machine logic (for event detection), which are novel and highly relevant in the context of in-network computation and can be reused for the in-network implementation of other stateful packet processing applications. We discuss requirements from the perspective of CEP applications addressing in particular distribution and consistency aspects. We show implications of stateful processing in in-network computing and provide feedback on related observed limitations in order to facilitate the further development of in-network computing for this class of applications. Lastly, we provide an evaluation of the performance properties of our implementation, which we deployed on programmable NIC hardware targets.

5.1 Background

In this section, we give a brief introduction to stateful data plane programming and describe challenges and limitations imposed by P4 and existing hardware targets, followed by an introduction to Complex Event Processing.

5.1.1 Data Plane Programming with P4

As introduced in Section 2.6, the paradigm of *data plane programming* subsumes the combination of (1) a quasi-standardized, hardware-agnostic domain-specific language (P4) implementing a uniform interface for defining the forwarding behavior of (2) emerging reconfigurable data plane hardware. Key elements of reconfigurable hardware are a parser defining header syntax and semantics and a match-action engine defining the semantics of processing. Both, parser and engine are software-definable, implementing a programmable multi-stage pipeline.

For P4CEP, we consider the following targets: hardware targets residing in end-systems, so-called *smart-NIC*s, such as (1) the Netronome Agilio NIC (NFP framework)[Ope18, Inc18b], (2) the NetFPGA platform [WSD+17, P4 18b], as well as (3) reconfigurable ASIC-based (RMT [BGK+13]) or FPGA-based (Corsa [Cor18]) data center switches. Furthermore, we consider (4) software-switch implementations, such as the P4 reference switch implementation bmv2 [P4 18a] and PISCES [SCP+16], and (5) extended Berkeley Packet Filter (eBPF) [Gun18], which provide fast in-kernel processing within end-systems.

In general, these hardware targets face inherent limitations [SAA+17, JYVM15]. (1) The size of both SRAM and TCAM memory is limited, which imposes bounds on the number of tables, their entries, and other held state. (2) Hardware switches are designed to unconditionally guarantee line-rate throughput. This places an upper bound on processing latency in the order of tens of nanoseconds, consequently bounding the number and complexity of packet operations in each pipeline stage. Hence, P4 models the control flow as an imperative program that specifies the execution sequence through the pipeline as a DAG, which rules out loops and thus renders P4 Turing-incomplete. (3) Stateful packet processing on programmable switches has been shown to be challenging [SCB+16]. Unsynchronized, concurrent access to state memory can lead to inconsistency effects, such as lost updates, which pose a severe threat for the correctness of stateful packet processing algorithms. The support of atomic memory operations is target-dependent and not mandated by P4. However, Netronome's NFP SDK provides a number of mechanisms for data consistency. Also for data center switches, hardware designs guaranteeing strong consistency semantics have been proposed [SCB+16]. We elaborate on consistency aspects while also referring to our concrete implementation later in Section 5.4.5.

While reconfigurable switching ASICs are primarily designed for network functions like forwarding, FPGAs are much more flexible as they allow for the implementation of custom logic in hardware. To be able to exploit the extended programmability of such targets, the P4 version $P4_{16}$ includes the `extern` primitive, which provides an interface to functions that are not part of the P4 specification, such as checksum computation and cryptographic operations. They can also be used for synchronization of memory access. For instance, the NFP framework allows referencing to external functions (`primitive actions`) written in a C-variant *Micro-C* [Net14] and executed in a Micro-C-sandbox running on NFP's micro-engines. Similar to $P4_{16}$'s `extern` concept, NFP's `primitive actions` can be used to process and manipulate packet headers and metadata fields as well as access stateful memory (registers). Micro-C natively supports efficient atomic arithmetic operations and has a built-in mutex and semaphore library. Although external functions are a very powerful concept, they break target-independence and possibly lead to unbounded processing latency.

5.1.2 Complex Event Processing

Complex Event Processing (CEP) [ABW06, Luc08, CM10, CM12b] is a paradigm to infer the occurrence of situations of interest from basic events. For instance, in the field of

algorithmic trading, a situation of interest can be the detection of a leading market signal, whereas the basic event streams contain stock quotes of a stock exchange [Hir12]. An example from the field of sensor fusion is detecting fire (the complex event) by reasoning over measurements of networked smoke and temperature sensors (basic events), as illustrated in Figure 5.1. In doing so, a CEP system deploys a distributed *operator graph* between event sources and sinks, where each operator detects a specific event pattern in its input streams and emits output events when instances of the corresponding event pattern have been detected.

The pattern to be detected by a CEP operator is typically defined in an event specification language [CM10] as a *continuous query*. Such a query consists of a number of matching expressions, such as Sequence, AND, OR, NOT, etc., that specify the conditions under which a sequence of input events matches the query. Furthermore, a query can contain aggregation operations such as MAX, MIN, AVG, etc., that are known from stream processing systems [ABW06]. In the fire detection example, these expressions are used to combine measurements of different sensor types (smoke, temperature) and allow the reasoning over their aggregated measurement values. Based on existing languages, we define a meaningful subset for in-network CEP, which we describe in greater detail in Section 5.4.2.

CEP operators are often stateful, i.e., the processing of one event may influence the internal state of the operator, which in turn influences the processing of subsequent events. Usually, the state relevant to a CEP operator is limited by a sliding window [ABW06, MST+17]. A sliding window restricts the infinite sequence of input events in an operator to a subsequence that can match the query. The extents of a sliding window are specified by a *window policy*, which defines the size of a window and its slide, i.e., by how much the window moves from one window instance to the next. In our running example, sliding windows, as illustrated in Figure 5.2, enable reasoning over time-series of measurements, e.g., allowing to infer trends. For instance, a fire can be defined to be inferred, when the averages over the last n measurements of the smoke and temperature sensors exceed a given threshold.

Thus, both pattern detection and sliding windows require holding state among the processing of incoming events. For an in-network implementation, this consequently mandates stateful processing of packets (events), holding and processing per-packet state as well as inter-packet state. Required consistency semantics on reliability (lost events) and ordering (out-of-order events) in event processing may differ depending on the CEP application, as we detail later.

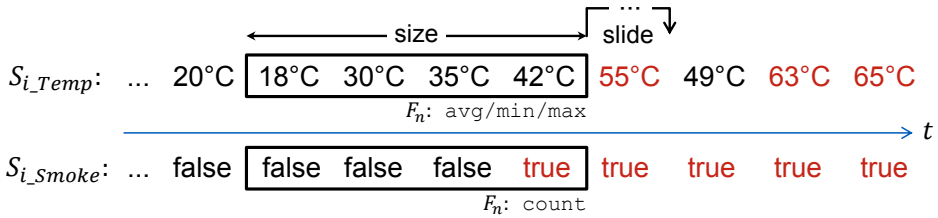


Figure 5.2. Illustration of a Sliding-window within Complex Event Processing in a fire-detection scenario. The window splits the infinite input event stream into finite-sized sub-sequences.

Typically, CEP operators are executed on end-systems. Typical performance figures show average processing latencies of about 200 µs, excluding the end-system's network stack latency, for detecting sequences of two states [CM12a], which is the simplest form of stateful processing. In terms of throughput, highly parallel implementations of CEP operators on multi-core CPUs can reach up to 218,000 events/second for more complex patterns [MST+17]. We provide a comparison with a state-of-the-art software-based CEP framework Apache Flink later in the evaluation.

5.2 System Model

The underlying system model for P4CEP, illustrated in Figure 5.3, assumes a set of **end-systems** that are interconnected by a set of programmable network processing elements (**P4CEP-targets**), forming a data plane topology.

End-systems that host event-based applications (CEP end-systems) are differentiated into **event sources**, which observe **basic events** and disseminate them, e.g., networked-sensors or server reporting performance metrics or log data, and **event sinks**, which receive and react to the delivery (also called notification) of **complex events**.

P4CEP-targets (listed in Section 5.1.1) implement two types of functions: (1) network functions, which co-exist with CEP (Co-NF), typically simple forwarding of packets, and (2) CEP functions, i.e., **CEP operators** implementing in-situ event detection. CEP operators and CEP end-systems comprise the **operator graph** spanning an overlay network.

The **P4CEP runtime component** implements a control plane interface to P4CEP-targets. Besides deploying compiled P4CEP programs, it handles all runtime tasks: updating P4 table entries and state transitions in the CEP engine as well as acquiring statistics and other monitoring data from the targets.

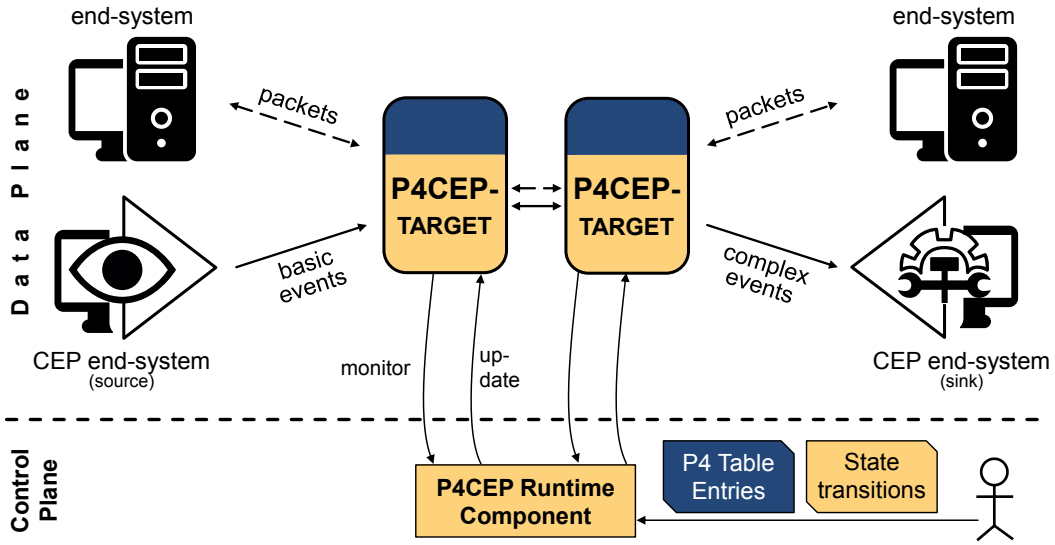


Figure 5.3. The P4CEP system model

5.3 Distribution and Consistency Aspects

Distribution is an inherent property of CEP systems due to the inherent distribution of sources, sinks, and operators across the overlay network. In this section, we discuss aspects of the distribution of CEP along with implications on consistency in light of its requirements, before proceeding with particular specifics and benefits regarding distribution in P4CEP.

5.3.1 Requirements from the Application's Perspective

From the perspective of CEP as an application, it is important that the detected complex events are a true representation of the monitored environment. Thus, they have to capture and process underlying basic events in a consistent way, ensuring that neither false-negatives, i.e., some events of interest were not regarded (*missed events*), nor false-positives, i.e., events that did not actually happen in the monitored environment (*"wrong" events*), do occur [May18]. On the strictest notion, this requirement would translate to strong requirements on *consistency*, i.e., exactly-once and in-order processing of basic events [CEF+17, KF19] ensuring the *correctness* of processing and delivery of events.

A CEP application may not only consider an event missed when a constituting basic event is is not regarded at all, for instance due to a packet drop in the overlay network. It may also consider an event missed if it is delivered too late. Hence, the *timeliness* requirement stipulates the importance of latency in event detection, requesting upper

bounds for the time span from the detection of a complex event at on operator to the delivery of its notification to the event sinks. The severity of delayed event delivery, and possibly an effective reordering of events (cf. in-order processing above), ranges from obsoleteness of events, when late events are not relevant any more, up to actual false positives, when late events cannot be discriminated from and are hence interpreted as subsequently detected events.

Another requirement of CEP is *scalability*. The distribution of operators improves scalability and performance of event detection. Concepts for distribution have been proposed in great numbers in the CEP literature and can be categorized by (1) placing the operator ideally in the network (operator placement) [Ac04, PLS+06, RDR10, CGLPN16], (2) distributing the operator (disaggregation of event detection to multiple operators) [SMMP09, KKR10], and (3) parallelization of the operator (data parallelization) [HSS+14, BMK+11, MKR15].

5.3.2 Control Plane: Operator Placement and Control under Global View

The global view on the network as part of the SDN paradigm simplifies and improves the quality of placement of P4CEP operators. Fully decentralized placement algorithms, as described above, can be replaced by algorithms implemented in a logically centralized controller (the P4CEP runtime component) leveraging global knowledge or use a full-range distribution model, i.e., anything in between fully decentralized and centralized, as described in Chapter 4. The control plane data required for the control and management of a distributed CEP (control state) is topological information, i.e., the operator graph and overlay network (information about event sources and sinks, placement of operators) as well as the mapping of complex event patterns to operators. For instance, in contrast to decentralized placement algorithms, the controller can directly use its global knowledge to evenly distribute the CEP load in order to avoid missed events due to network congestion.

5.3.3 Data Plane: Consistent One-to-many Event Propagation and CEP Operation

In P4CEP, the data plane is used for (1) the propagation of events, i.e., the forwarding of CEP (and non-CEP) packets, and (2) the implementation of CEP operators within P4CEP targets, as we describe in Section 5.4.4. Regarding data plane consistency we thus

differentiate between consistent event propagation (network-centric) and consistent CEP operation (processing-centric), which we describe individually in the following. Atomicity of data plane updates triggered by the P4CEP runtime component can be guaranteed by *atomic update blocks* as per the $P4_{16}$ specification [P4.18].

When a basic event is required by more than one operator, i.e., the distribution of operators is not disjoint with respect to the complex event patterns, it has to be replicated and propagated to multiple operators in the overlay network. In other words, a one-to-many communication pattern, such as the efficient *multicast*, is required. Operator placement typically is dynamic in the sense that it reacts on changes to the overlay network or the pattern-mapping, ultimately resulting in updates of the multicast distribution tree. As described in Chapter 3, this kind of update will inevitably cause inconsistency effects in the data plane in form of duplicates (possibly causing wrong events) or drops (possibly causing missed events), which our approach for correct multicast tree updates allows to control.

Consistency regarding the actual processing of events, i.e., pattern detection, is determined by the processing engine of a target. Due to the targets' predominant execution model of pipeline processing on highly parallel execution units, the main challenge lays within keeping the stateful data consistent. We describe mechanisms for data consistency in our concrete implementation of P4CEP's stateful processing in detail later in Section 5.4.5.

5.3.4 Stateless In-network Filtering of Events

The early filtering of unnecessary events, i.e., events that are not part of any complex event pattern mapped to downstream operators or that do not have downstream sinks, leads to reduced load in the event detection engine and to increased bandwidth-efficiency, overall benefiting all stipulated CEP requirements.

While not part of this work, we want to highlight the opportunity to leverage the distribution inherent to CEP by embracing in-network computing also for in-network filtering. Compared to P4CEP's stateful processing, filtering does not require holding state, except the filter rules, which are however not changed during or as a result of processing. Thus, limitations stemming from stateful processing do not apply, overall simplifying implementation.

Bhowmik *et al.* [TKBR14, BTK+17, Bho17] have previously proposed concepts and provided an implementation for offloading of content-based filtering in publish/subscribe middleware systems from dedicated middleboxes to OpenFlow SDN switches, enabling

line-rate filtering. Since OpenFlow switches use a fixed pipeline and do not have a flexible packet header parser, the content of a message has to be mapped into static header fields, limiting expressiveness. Thus, we expect an in-network implementation to be more expressive while having reduced implementation complexity.

5.4 P4CEP: Design & Implementation

In this section, we first describe the processing semantics and expressiveness of P4CEP as defined by our P4CEP rule specification language, before we describe the concrete mapping to a P4 pipeline along with a description of the P4CEP workflow and compiler, present details of the pipeline implementation, and discuss inherent limitations of (i) the P4 language and (ii) stateful data plane programming in general. We illustrate our design by an accompanying example running throughout the entire section.

As stated in the system model, we assume only in-situ operators residing on P4CEP targets, rather than incorporating ex-situ operators residing on dedicated end-systems, for instance traditional software middleboxes. Although the design space could be in principle easily extended to such a hybrid scenario where CEP functionality is distributed to both programmable switches and dedicated end-systems, in this chapter we focus on the data plane implementation of CEP using P4, while the hybrid scenario would mainly increase complexity in the control part. Furthermore, for the sake of illustration, the descriptions in the subsequent sections take the perspective of just a single P4CEP-target, without loss of generality.

5.4.1 Abstract Event Processing Model

In a nutshell, P4CEP implements the following event processing model (for details see Section 5.4.4). Each event is processed in the order of ingress at a P4CEP target. The processing of an individual event in the incoming event stream consists of (1) *window operators* employing aggregation functions on windows storing values of historic events and (2) the *detection of patterns (complex events)* in the incoming event stream. While window operators operate directly on stored historic events, pattern detection is implemented by finite state machines (FSMs), which encode the rules that specify complex events.

P4CEP supports the parallel detection of multiple patterns (encoded in multiple FSMs), either of different patterns or multiple detection of the same pattern (in the case of interleaving basic events in the event stream).

5.4.2 The P4CEP Rule Specification Language

As described above, the specification of the CEP-functionality in P4CEP is split into the definition of window operators and event definition rules, which are encoded into finite state machines (FSMs) implementing pattern detection. The P4CEP rule specification language comprises both, the definition of window operators and event definition rules.

Listing 5.1 shows our exemplary rule definition whose complex event definition is translated into a FSM shown in Figure 5.4. Note that for the sake of illustration, we simplified the example by considering a network-centric CEP application that considers arbitrary network traffic as events, evaluating headers of common layer 3 and 4 network protocols. For a CEP application that considers other types of basic events, such as readings from temperature sensors, as in the initial generic CEP example (see Figure 5.2), event type and data can be encoded in custom headers (CEP event header) for instance using type-value pairs (see Section 5.4.3 “CEP Design Config: Event Header Definitions”). Since P4 allows for an arbitrary interpretation of packet headers and fields, both options, existing or custom headers, are functionally equivalent for P4CEP.

As a side effect, the example shows that some co-NFs, such as anomaly/intrusion-detection, typically relying on sums or counts of specific (sequence of) packets, can be mapped directly to CEP-functionality. The illustrated example enables the detection of the following anomaly pattern: a large IPv4 packet and an HTTP-packet, followed in sequence by an UDP-datagram or the sum of total lengths over all last $n = 8$ seen IPv4 packets exceeding 6 KB.

We describe the formal elements of the P4CEP rule specification language which are used to express such patterns in the following.

5.4.2.1 Specification of Windows

The definition of a **`window`** consists of a name, a window **`size`**, and a field reference whose **`value`** is to be stored within that window. There might be multiple window specifications. Windows can be referenced by name within a pattern of a complex event definition or as its return value.

In P4CEP, field references are simple references to P4 headers or packet metadata that must have been parsed by the P4 program and thus be defined either as a CEP event header or as a co-NF header.

The example defines a window named `sample_wnd` that stores the last $n = 8$ values of the IPv4 header `Total Length` field of the last n ingress IP-packets (basic events).

Listing 5.1. Exemplary P4CEP-rule definition of a window and a sequential pattern, composed of predicates on simple L3/L4-packets and on a window.

```
var sum_threshold = 6000;
window sample_wnd {
  size 8
  value ipv4.totalLen
}
complex_event sample_evt {
  value sum(ipv4.totalLen)
  strategy skip-till-next-match
        instances 1
  pattern ([ipv4.totalLen > 500] && [tcp.dstPort == 80]) ;
          ([sum(sample_wnd) > $sum_threshold] ||
           [ipv4.protocol == 17])
}
```

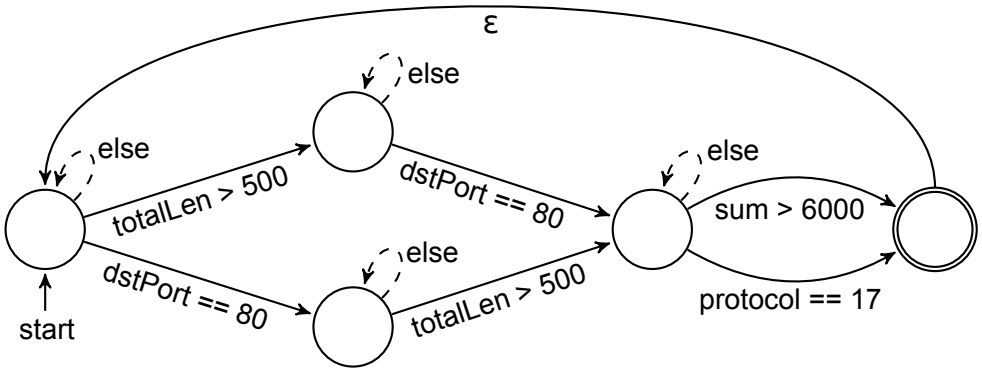


Figure 5.4. Generated finite state machine (FSM) detecting complex event patterns as specified in Listing 5.1.

Table 5.1. Representation of the FSM (see Figure 5.4) as a P4 match-action table as used in the P4CEP pipeline (see Section 5.4.4).

Keys		Values	
State	Match (predicate ID)	Next State	Accept. State
0	totalLen > 500	1	false
0	dstPort == 80	2	false
1	dstPort == 80	3	false
2	totalLen > 500	3	false
3	sum > 6000	4	true
3	protocol == 17	4	true

5.4.2.2 Specification of Complex Events

The definition of a **`complex_event`** is structured as follows. There might be multiple complex events defined in which case an ingress event is evaluated in the detection of all defined complex events.

(1) A return **`value`** to be set in a complex event packet sent in case of detection. This can be (1a) any valid P4 expression (static expression, field reference), or (1b) a reference to an aggregation function over a window (e.g., `sum(sample_wnd)`), or over a header field. In the example, the return value of the complex event named `sample_evt` is set as an aggregation function (the `sum`) over all `Total Length` fields of all processed IPv4 packet since the start of the pattern detection.

(2) A **`strategy`** specifying the state transition of the state machine that is encoding the pattern if an incoming basic event does not match any predicate:

(2a) `skip-till-next-match` performs a transition to the same state, i.e., ignores the event (`else`-branches in Figure 5.4 of the FSM-representation of the example).

(2b) `strict` resets the state machine by setting the next state to the initial state.

For the strategy `skip-till-next-match`, an additional parameter **`instances`** can be set. It enables multiple parallel detections of this complex event when the constituting basic events are interleaved in the incoming event stream. More specifically, it defines the number of matchers that are active in parallel. For instance, for a rule with the pattern `[ A ; B ]` and `instances = 1` an incoming event stream of <A, A, B> would result in just a single detection.

(3) A **`pattern`** of basic events defining a complex event. Basic events are specified by simple or compound predicates P_x. Predicates are demarcated by square brackets and combined to patterns using the following logical operators.

(3a) Sequence `;`: the left predicate must hold true before the right.

(3b) Conjunction `&&`: both predicates must hold true (in any order).

(3c) Disjunction `||`: one of the predicates must hold true.

In P4CEP, a predicate can be any valid P4 condition on one or more field references, or a condition on an aggregation function over a window or over a header field. A simple predicate consists of a single term, i.e., a comparison operator with two operands, (1) a window reference or a P4 expression and (2) a reference value

to compare the first operand to. A compound predicate consists of multiple terms combined by the logical operators conjunction (`&&`) and disjunction (`||`).

As described above, the example defines a pattern consisting of a sequence of event `A` and `B`. `A` is satisfied iff both predicates P_{x1} =`ipv4.totalLen > 500` and P_{x2} =`tcp.dstPort == 80` are true for two ingress events in any order, leading to alternative paths branching from the initial state in the corresponding FSM (see Figure 5.4). `B` is satisfied iff predicate P_{x3} =`sum(sample_wnd) > $sum_threshold` or predicate P_{x4} =`ipv4.protocol == 17` is true for any subsequent event. That is an ingress event that satisfies P_{x4}, i.e., any UDP datagram, or the satisfaction of P_{x3}, which is true iff the `sum` over the defined window `sample_wnd` (see above) is larger than the value of the variable of `$sum_threshold` (which is set to 6000 in its declaration (see Section 5.4.2.4)).

5.4.2.3 Aggregation Functions

The following aggregation functions on windows (window form) or field references (free form) are currently supported: `sum`, `min`, `max`, and `count`. If applied on a field reference, i.e., a header field, the aggregation function is applied on each basic event while the aggregate value is stored in a register. If applied on a window, as in the example, the function is used to build the aggregate over the entire window within the window evaluation.

`Count` differs from the other functions, as it does not operate on a field reference, but expects a boolean expression which is evaluated while the count of how many times the expression has been true is stored in a register. In its free form, its expression is evaluated upon event ingress and if true the counter is incremented. The actual counter value can be used within a predicate. For instance, P_x = `count(ipv4.totalLen) >= 500`. In its window form, it counts how often the expression is true for the values within the current window. A special variable `$value` allows referencing the current value of a window iteration within a predicate. For instance, P_x = `count(sample_wind, $value >= 500) >= 10` counts how many elements of the window have a value ≥500 and compare the counter value with the threshold of 10 in the evaluation of the predicate.

Implementing `average` is not straightforward due to P4's missing float support and lack of a division operator, but can be approximated by fix-point and bit-shift operations on windows of sizes 2^n.

5.4.2.4 Declaration of Variables

P4 expressions, which among other purposes are used for the definition of predicates, are hard-coded within P4 program code by the P4CEP compiler. In order to allow the change of expressions at runtime, variables can be used. They are declared using the keyword **`var`** by name and initial value. They are referenced by prefixing the name with `$`.

5.4.3 The P4CEP Workflow and Compiler

In the following sections, we describe the mapping of CEP functionality to P4 and discuss implementation details. P4CEP's design-workflow is illustrated in Figure 5.5. It is mainly composed of our P4CEP compiler and an unmodified **P4 compiler** chain, consisting of a target-independent and target-dependent compiler, as well as **target-dependent toolchains**. Currently, we support target-specific external functions for the Netronome NFP target. The user-input to the P4CEP compiler (**CEP design config**) consists of P4-definitions of header fields and parser instructions for packets that are to be interpreted and processed as basic events as well as declarations of window operators and event definition rules, which describe how complex events are derived from basic events, as defined by the P4CEP rule specification language (see Section 5.4.2).

From these definitions, the **P4CEP compiler** creates corresponding P4 source code,

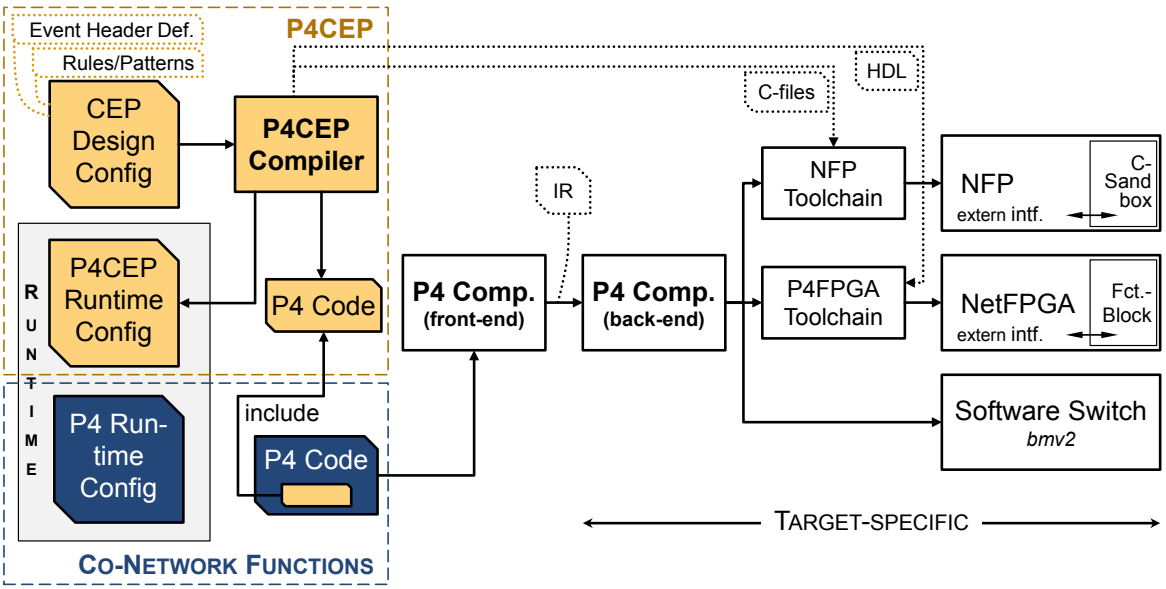


Figure 5.5. P4CEP workflow: design-time components and source files involved in building P4CEP for different targets

supporting both P4 specifications, i.e., versions $P4_{14}$ [P4 18c] and $P4_{16}$ [P4 18d, BD17]. It comprises definitions of registers (holding inter-packet state), metadata structures (holding per-packet state), auxiliary tables for (multiple) windows, implemented as register ring buffers, and (multiple) state machines, each associated with a complex event pattern to detect. The P4 code generated by the P4CEP compiler is merged with the user-provided P4 program source file, which implements co-NF functionality, using P4's `include`-primitive. Additionally, runtime configuration files hold table entries and can be (re-)deployed at runtime by the P4CEP runtime control plane component. They are created in a target-compatible format by the P4CEP compiler and given as user-input for the co-NF part, respectively.

5.4.4 The P4CEP Processing Pipeline

Figure 5.6 illustrates P4CEP's pipeline for processing packets and events within a P4-target. Upon packet ingress, a classifier discriminates CEP-related events, i.e., basic and complex events, from non-CEP related, i.e., other network traffic. We encode events in packet headers, leveraging P4's flexible and powerful parser, which maps the abstract bit-vector resembling an ingress packet to a concrete interpretation as given by the header definition, which we specify as part of the CEP design config. Non-CEP traffic is routed to the **Co-NF** P4 control flow (dark-shaded). CEP traffic is handled by the **CEP** ingress control flow (light-shaded). Note that although events are conceptually considered to be CEP-traffic, the target's responsibility is also the propagation of both simple and complex events to other operators or end-systems over the network, i.e., propagation in the overlay graph and hence forwarding in the underlay network. Thus, they may additionally be handled by the Co-NF control flow. When a complex event is created within a target, we send it to the classifier for further handling through the P4's `resubmission` mechanism.

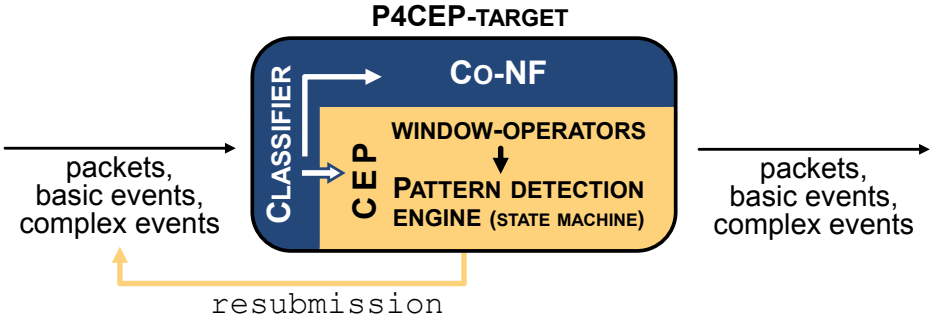


Figure 5.6. P4CEP's pipeline, depicting processing of packets and events within a P4-target

The main CEP logic is implemented in the CEP ingress control flow, which processes a packet that transports a basic event. It can be divided into **window operations**, which store the n last values of header fields in a FIFO-manner and offer aggregation functions over these values, and the **event detection engine**, which detects complex events based on a state machine implementation.

In the first step of **window operations**, the current instance count (ring buffer head pointer) of the window is read from a register and incremented with overflow handling. One drawback of P4 (***Limitation 1***) is that registers cannot be directly referenced in arithmetic operations or as table keys. Thus, register values have to be copied into dedicated intermediate metadata fields and back, which bloats code space and execution overhead. Then, the header field value of the current event and instance count are stored in registers, i.e., are persisted in the window. For applying the aggregation function on the window, our compiler has to unroll the window iteration, due to P4's lack of loops (***Limitation 2***). The aggregate value is stored in a metadata field m_{aggr}, as is the iteration counter m_{iter}. For each value r_i in the window, r_i has to be copied from the window register to a metadata field m_i. Then, the aggregation function is applied on m_{aggr}, referencing m_i. This procedure is repeated for all windows.

We define a complex event as a pattern of basic events which is formally defined by the P4CEP rule specification language. The detection of patterns within the **pattern detection engine**, can be modelled as a deterministic finite **state machine** $C = (\Sigma, S, s_0, \delta, F)$, as illustrated in Figure 5.4 on p. 146. It consists of a sequence of basic events (input symbols $x \in \Sigma$), where typically a basic event is specified by **predicates** P_x (simple or compound) on packet header fields (see Section 5.4.2.2). For each pattern, the following actions are executed sequentially: all packet predicates are evaluated. If P_x evaluates to true, an id associated to that predicate ($P_x \rightarrow x$) is stored in a metadata field m_x. Then, the state machine is executed by first acquiring the current state $q \in S$ by copying from a register to a metadata field m_q, followed by performing a lookup with the key-pair $<m_q, m_x>$ on the **transition table** Δ—a P4 table encoding δ (see Table 5.1 on p. 146). Upon a match, the returned value pair $<q_n = \delta(q, x),\ b_{is\_accepting}>$ is written to registers if q_n is not an accepting state ($\neg b_{is\_accepting} \equiv q_n \notin F$). If it is an accepting state, the state machine is reset, i.e., q_n is set to the initial state s_0, and the return value for the complex event is set, encoded in a header field, before the packet is sent to registered CEP sinks or other operators using the P4-`resubmission` mechanism. P4CEP allows the detection of multiple complex events by sequential execution of the corresponding state machines.

5.4.5 Data Consistency in P4CEP's Stateful Processing

In the following, we address data consistency in stateful processing in P4 in general, as well as mechanisms specifically used in our implementation. As described in Section 5.3.3, the main challenge here is to keep the stateful data consistent while it is being processed and changed by highly parallelized processing units[1].

P4CEP holds inter-packet state in registers. Inter-packet state state in P4CEP comprises window ringbuffers (event header values, head index, window aggregate value), the current state q_i of each state machine C_i, and possibly additional instance counter variables for the count aggregation function and for multiple instances of event detectors. Consequently, inconsistency effects may lead to missed events, for instance when values are written into the same window ringbuffer slot because the head pointer has not yet been updated (*lost update*), or even produce wrong events, for instance when a new state q_{new} due to a triggered transition has not yet been persisted in the register, while another arriving event of the same type is being processed with the FSM still being in the old state q_{old} (*dirty read*). In order to prevent data inconsistency and to ensure correctness (cf. Section 5.3.1), write access to registers or more general, execution of critical code sections, have to be isolated. It is obvious, that this partial linearization of processing is mandatory for correctness but leads to degradation of performance (***Limitation 3***).

For the expression of atomic execution of code blocks, $P4_{16}$ provides the `@atomic` annotation, which can be applied to block statements, parser states, control blocks, or whole parsers [P4 18d, P4.18]. If a target is not capable of executing a block atomically, its compiler back-end rejects the program, such that atomicity is either clearly ensured or clearly rejected. $P4_{14}$ lacks native primitives for atomic execution.

Regarding our primary target for P4CEP, the Netronome NFP SDK provides several mechanisms for data consistency which can augment a $P4_{14}$ program or which are directly mapped to by the NFP $P4_{16}$ back-end compiler. External C functions may employ a simple spin lock algorithm which makes use of the atomic memory access feature provided by the NFP's C-API. The P4CEP compiler inserts this function call wherever a critical region is required. As a more transparent mechanism, the NFP SDK provides annotations on register and action objects within P4 source code. In P4CEP, every register definition is annotated with the `@pragma netro reglocked` pre-processor pragma, which will create a mutex for each register that grants exclusive access within all actions that reference that register. Hence, this pragma is used to indicate that all accesses to the given register

[1] see Section 5.5.1 for details on the microarchitecture of our primary target

should occur under mutual exclusion. Furthermore, to prevent the NFP from caching register contents, which would also jeopardize data consistency, the P4CEP compiler prepends the `@pragma netro no_lookup_caching` pragma to each action that accesses a register. Despite full protection and thus ensured correctness, we reach practical performance results on the NFP, as we show later.

5.4.6 Limitations for Stateful Processing

Here, we discuss the encountered limitations of P4 and their implications for stateful packet processing. Additional to the aforementioned Limitation 1 (no direct operations on registers), Limitation 2 (lack of a loop construct), and Limitation 3 (synchronizing access to inter-packet state, atomic code blocks), another limitation lays in the fact that conditions in P4 can be *only* used within the control flow, not within actions (Limitation 4). Moreover, $P4_{14}$-actions cannot be directly executed within a control flow (Limitation 5). Instead, they have to be indirectly executed by using P4's `apply`-primitive to perform a lookup on an empty dummy-table where the action to be executed is specified as the default action. This limitation clearly shows that P4 is centered around the match-action paradigm which in particular traditional targets have been following. For large portions of P4CEP-generated code, this workaround has to be applied by expressing it's processing semantics in match-action semantics, which is feasible but inefficient. We realize that some of these limitations are inherent design trade-offs in creating P4, which seemed to be driven by satisfying the intricate requirements of switch hardware architectures [JYVM15] to maintain line-rate processing, for instance ruling out loops (Limitation 2), rather than having stateful packet processing in mind. However, we observe that the evolution of P4 with $P4_{16}$ reflects the evolution of P4 targets in terms of extended capabilities and different processing semantics. Consequently, $P4_{16}$ better facilitates stateful packet processing, e.g., by the introduction of the atomicity primitive (Limitation 3) and corrects other seemingly unnecessary limitations like the action indirection (Limitation 5).

5.5 Evaluation

In this section, we evaluate P4CEP's practicability on state-of-the-art P4 targets and compare its performance with a popular software-based CEP system.

5.5.1 P4CEP Hardware and Software Targets

As a **hardware target platform** for P4CEP, we use a Netronome Agilio smart-NIC with 2 × 10GbE ports (ISA-4000-10-2-2), based on Netronome's mid-range NFP-4000 Flow Processor [Inc18a, Inc18b] as part of their Agilio CX series.

The NFP-4000 clocks at 1.2 GHz and features 60 eight-way multi-threaded Flow Processing Cores (FPCs) based on a RISC design, a hierarchical memory subsystem, and a programmable PCIe interface, interconnected via a 9.6 Tbps switch fabric, achieving 100 Gbps stateful packet processing throughput at 148 million packets per second (Mpps). Its hierarchical memory subsystem consists of an internal memory unit (IMU) with 4 MB internal SRAM with a latency of ≤ 25 cycles and external memory units (EMU) providing additional 6MB SRAM increasing the latency to ≤ 500 cycles though. Furthermore, the EMUs facilitate the integration of external DDR3 DRAM memory (up to 8 GB) usable as flow-table memory or as a packet buffer. As we show later, program memory, stored in SRAM, constitutes a bottleneck for the scalability of parallel event processing and window operators.

An optimization of the NFP's memory management as to leverage additional EMU SRAM memory to shift program memory limitations is thus of particular interest. A further optimization would be to leverage NFP's massively-parallel processing architecture by deliberate placement of independent CEP operators on the NFP's highly-parallel processing units. However, in order to stay target-independent, we refrained from implementing such NFP-specific optimizations, except the Micro-C-sandbox implementation for the window iteration and the Micro-C spin-lock implementation to isolate access to critical sections. On our NFPs with NFP SDK version 6.0.4, we run a pure P4 implementation of P4CEP (**NFP**) and the optimized version employing NFP's Micro-C-sandbox for window iteration (**NFP-C**).

As a **software target platform** for P4CEP, we use the popular P4 reference switch implementation **bmv2** [P4 18a], implemented in C++, at version 1.12.0. One should note that bmv2 is not optimized for performance but to cover the P4 specification in terms of functionality. Since it is often used to showcase other P4 programs, we consider it meaningful for comparison though.

5.5.2 A Software-framework for CEP: Apache Flink

Apache Flink [CKE+15] is an open-source software-based stream processing system implemented in Java. It has been heavily optimized for performance and is hence being used

as a CEP framework for wide-spread real-world applications. Due to its high performance [KRK+18] in terms of latency and throughput, we use Flink as a reference state-of-the-art software-based CEP system to compare P4CEP against. In order to provide a fair comparison, we tried to keep the semantics of CEP processing between P4CEP and Flink as similar as possible. Keeping in mind that results are hence not directly comparable, they nonetheless provide a fair approximation of performance comparison.

As described in greater detail in [CKE+15], Flink differentiates APIs for batch processing (DataSet API) and real-time processing (DataStream API), where we use the latter, reflecting CEP's execution model. Data processing is modelled as DAGs (Dataflow Graphs), encoding data flows between sources, operators, sinks, and external data storage devices. The provided primitives allow to closely mimic P4CEP's processing semantics. Flink's internal buffer management allows for trading-off latency and throughput. Buffers, containing payload data, are forwarded to a subsequent operator in the DataFlow graph either as soon as they are full or upon a timeout. Hence, the timeout parameter constitutes a lower latency bound and is consequently considered a crucial parameter for our evaluation.

5.5.3 Methodology

For evaluating latency and throughput of P4CEP and Flink, we use a setup as illustrated in Figure 5.7. We employ a pair of end-systems (Intel Xeon E5-1650v4 @ 3.6 GHz, 6 physical cores, 32 GB RAM), running CentOS 7.6 with a 4.9.75-29.el7 kernel and being interconnected by two 10GbE links.

The *CEP operator end-system* hosts the CEP operator that is implemented either in hardware by P4CEP running on the Netronome Agilio NFP (**NFP** and **NFP-C**) or in software by P4CEP running on **bmv2** version 1.11.0 or by Apache **Flink** version 1.4.2 on OpenJDK 1.8.0_161. Depending on the evaluation scenario, the NFP's physical ports (p_0, p_1) are either used directly by the NFP engine (for P4CEP on NFP) or are transparently exposed to the operating system as ordinary network interfaces (for P4CEP on bmv2 and Flink), using the NFP's virtual ports ($v_{0.0}$, $v_{0.1}$). The latter scenario is equivalent to using an ordinary NIC, instead of the NFP—we could not measure any latency nor throughput penalty.

The *CEP src/snk end-system* is responsible for the production of basic events (CEP source) and consumption of complex events (CEP sink), separated by dedicated network namespaces (with a shared clock), each being assigned one 10GbE network interface. Basic events and event definition rules at the operator are aligned such that each basic

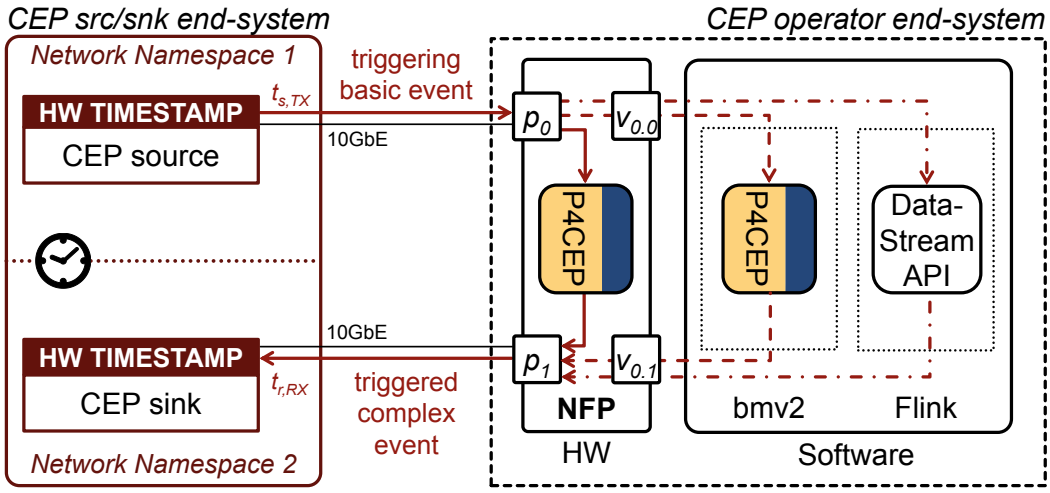


Figure 5.7. Evaluation setup for measuring latency and throughput of the P4CEP hardware target (NFP) and software target (bmv2), as well as of the Apache Flink software framework.

event triggers a complex event detection. Thus, packet rate and event rate are equivalent. For **latency evaluations**, basic events are sent at an egress rate of 2 kpps for a 30 seconds period to the operator system, resulting in a sample size $N = 60{,}000$ events. An event is encoded as a UDP datagram with an 8-byte payload to carry a unique sequence number for packet identity. We approximate the latency for CEP processing (end-to-end event detection latency) l_p by hardware-timestamping the egress of basic events ($t_{s,TX}$) and the ingress of consequently detected complex events ($t_{r,RX}$) as: $l_p = t_{r,RX} - t_{s,TX}$. Propagation and serialization delay are negligible. For **throughput evaluations**, we send basic events as denoted above, at an egress rate equivalent to full line-rate and measure the ingress rate of complex events (B_i) by counting respective packets. Hardware-timestamping is not used. We assess the relative throughput $B_p = B_i/B_{lr}$, where B_{lr} is the line-rate throughput.

5.5.4 Baseline Performance

To give an impression of the P4CEP target's absolute performance, we provide a minimal baseline analysis of a simple P4 program, implementing stateless forwarding based on parsing layer 2–5 headers of smallest-sized packets. We measured a baseline latency including serialization delay of 6.8 µs for NFP and NFP-C and 475 µs for bmv2, respectively. Baseline throughput is full line-rate (≈14.88 million packets per second (Mpps) for 10GbE) for NFP and NFP-C, where bmv2 shows $B_p \approx 0.08\%$ (≈ 12 kpps).

For Flink, we mimic the P4 target's baseline behaviour by defining a minimal dataflow graph consisting of a `DataStreamSource` encapsulating each received datagram and a `SinkFunction` that directly forwards it. We measured Flink's baseline latency as 205 µs along with a baseline throughput of $B_p \approx 5\%$ (≈ 744 kpps) with timeout parameter set to 0, effectively removing the latency lower bound at the cost of decreased throughput.

5.5.5 Scalability Evaluation

In our main evaluation, we extensively assess the scalability of P4CEP in terms of window size, pattern complexity, predicate complexity, number of variables, number of complex events, and number of event instances. For the window size scalability, we provide a comparison with Apache Flink and bmv2.

The results are summarized in Table 5.2 and described in the subsequent sections.

Table 5.2. Summary of P4CEP scalability results on the NFP target

complexity	limitation/ test. until	space impact	latency impact	throughput impact
window size	≤ 1000	ring buffer, linear increase	+1 µs, lin.	sublin. decr.
pattern	≤ 1000	transition table, linear increase	*	*
predicate	≤ 1000	expression validation, linear increase	8 ns, lin.	*
# variables	≤ 240	dummy table, stepwise-linear increase	+500 ns, const.	*
# complex events	≤ 4	tables, expression valid., registers	+1.3 µs, lin.	−1.5 Mpps, lin.
# instances	≤ 5	tables, expression valid., registers	+1.5 µs, const.	*

* not measurable

5.5.5.1 Window Size

To assess P4CEP's scalability with respect to window size, we use a rule definition as depicted in Listing 5.2. It consist of a window definition of varying size n where $0 \leq n \leq 1000$ along with a simple pattern that triggers the evaluation of the window and evaluates a sample predicate, which is always true in this case.

Figure 5.8 shows the mean performance over $N = 60{,}000$ samples within the interval $0 \leq n \leq 20$. We measured a processing latency of $9.8\,\mu s \leq l_p \leq 29.5\,\mu s$ and relative throughput of $56\% \geq B_p \geq 16\%$ for NFP-C. The pure P4 implementation (NFP) performs

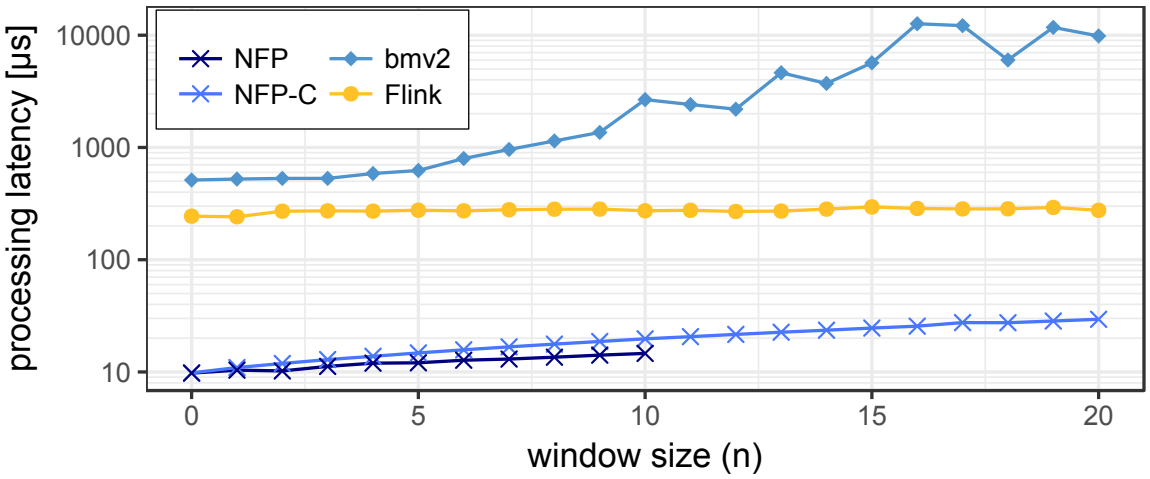
Listing 5.2. Rule definition for the scalability evaluation, where the window size parameter (grey box) is varied. The pattern is used to ensure triggering of the window evaluation.

```
window sample_wnd {
    size 2
    value ipv4.totalLen
}
complex_event sample_evt {
    value event.id
    strategy skip-till-next-match
    pattern [event_id.id >= 0] || [sum(sample_wnd) > 0]
}
```

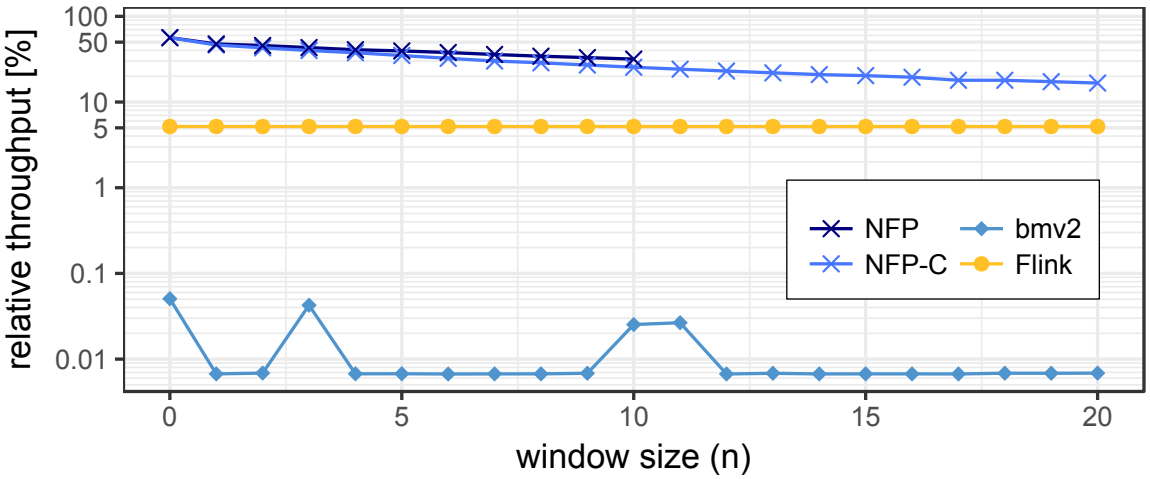
slightly better, showing low overhead for the `extern`-mechanism. However, for $n > 10$, the size of the generated P4 code grows too large, due to manual loop-unrolling, exceeding the program size limit of our NFP-4000 smart-NIC. With window operations implemented in the C-sandbox, NFP-C scales linearly with $\Delta l_p \approx 1\,\mu s$ per iteration, up to $l_p \approx 969\,\mu s$ for $n = 1000$, whereas B_p drops sublinearly down to 0.4% (≈60 kpps). Considering the NFP's baseline latency of 6.8 µs, the overall latency penalty that the P4CEP pipeline imposes is quite moderate with 3 µs. While throughput penalty is more drastic, one should keep in mind that the NFP-4000 is a mid-range smart-NIC for which we employed no optimizations, whereas the performance of other targets like FPGA-based smart-NICs or programmable data center switches can be expected to be much higher. The NFP(-C)'s standard deviation of l_p (jitter) is overall low (tens to hundreds of nanoseconds), as is its deviation of throughput (≈0.02%).

While bmv2 has no code size restrictions, its performance is significantly worse. Starting with $l_p \approx 512\,\mu s$ and $B_p \approx 0.05\%$, it also shows inferior scalability properties, exceeding $l_p \approx 10$ ms for $n > 15$. Jitter is also of much greater extent.

For a comparison with Flink, we implemented a dataflow graph, shown in Listing 5.3, that tightly mimics the semantics of P4CEP. Similar to the baseline evaluation, we use a `DataStreamSource` (`ids`) for the generation of an event from each received datagram and a `SinkFunction` (`IdPacketSink`) that creates the complex event notification, encapsulated in an UDP datagram. The `countWindowAll` method instantiates a count-based sliding window, of varying size n (with $n = 2$ in the example) that is evaluated on each event ingress. The `SumWindowFunction` aggregates the values in the window using a SUM operator, as the name suggests. While Flink mainly focusses on time-based windows, its DataStream API also allows for the implementation of a count-based sliding window that is semantically equivalent to P4CEP's sliding-window implementation. Again,



(a) End-to-end event detection latency / CEP processing latency l_p depicted on a log scale



(b) Operator throughput B_p relative to line-rate depicted on a log scale

Figure 5.8. P4CEP's performance for increasing window sizes n on an NFP smart-NIC where window operators are implemented natively in P4 (**NFP**) or within the NFP's C-sandbox (**NFP-C**) and on the P4 reference software switch implementation **bmv2**. Apache **Flink** is added as a reference for state-of-the-art software-based CEP frameworks.

Listing 5.3. Functional Flink equivalent to the P4CEP rule definition for the scalability evaluation (see Listing 5.2). The window size parameter (grey box) is varied.

```
ids.countWindowAll( 2 , 1)
    .process(new SumWindowFunction())
    .addSink(new IdPacketSink(<...>));
```

we set the buffer timeout parameter within the `StreamExecutionEnvironment` to 0, thus optimizing for latency. As shown in Figure 5.8, Flink does not show significant performance impact when scaling the window size. Both latency and throughput stay constant even within $0 \leq n \leq 1000$ with $l_p = 270\,\mu s$ and $B_p = 3.5\%$ (≈ 520 kpps), respectively. Due to software processing, the standard deviation of latency (jitter) and throughput are with $\approx 62\,\mu s$ and $\approx 1\%$ significantly higher than for NFP(-C). While both, latency and throughput of Flink are an order of magnitude worse than for P4CEP on NFP(-C) for small n, Flink outperforms P4CEP for $n \gtrsim 280$.

5.5.5.2 Pattern complexity

We denote pattern complexity as the number of distinct predicates n that comprise the pattern in a sequential combination. As shown in Listing 5.4, we increase the number of distinct predicates n within a pattern. In order to enforce that each predicate P_x is actually evaluated, they must not be equal but they all have to evaluate to false. Due to their sequential composition within the pattern, the number of states $|S|$ and consequently the transition table Δ increases linearly with n. The transition table Δ is implemented by a match-action table within the P4 pipeline. The NFP implements match-action tables in content-addressable memory (CAM), which have O(1) memory access. Thus, l_p is expected to grow linearly. However, since the memory access time is several orders of magnitude lower than l_p, we could not measure any impact on l_p and on B_p within $0 \leq n \leq 1000$.

Listing 5.4. Rule definition for the scalability evaluation, where the complexity of a pattern (grey boxes) is varied by increasing the number of sequentially-combined distinct predicates.

```
complex_event sample_evt {
    value event.id
    strategy skip-till-next-match
    pattern [ethernet.etherType == 2049] ; [ethernet.etherType == 2050]
}
```

5.5.5.3 Predicate complexity

We denote predicate complexity as the number of simple predicates n within a compound predicate. As shown in Listing 5.5, we increase the number of distinct simple predicates n that comprise a compound predicate, where the pattern consists of just that one compound predicate. The simple predicates can be combined by conjunctions or disjunctions. Since the validation of expressions, to which simple predicates are mapped, are executed sequentially for each ingress event, the number of validations increases linearly with n. Overall, we have measured minimal performance impact. For $n = 1000$, l_p is increased by $\approx$8 µs, whereas B_p stays constant. We can calculate a mean linear increase of 8 ns, which meets approximately the resolution of our hardware-timestamping mechanism.

Listing 5.5. Rule definition where the complexity of a single predicate (grey boxes) is varied by increasing the number of its constituting simple predicates.

```
complex_event sample_evt {
    value event.id
    strategy skip-till-next-match
    pattern [ ethernet.etherType == 2049  || ethernet.etherType == 2050 ]
}
```

5.5.5.4 Number of variables

As shown in Listing 5.6, we increase the number of variables n by adding additional definitions of variables along with a reference within a predicate. Since variable assignment has to be implemented within a dummy table as a workaround in $P4_{14}$ (table indirection, Limitation 5), additional latency of $\approx$500 ns incurs for the traversal of an additional table in the pipeline. We could not measure any impact on throughput. The number of variables per table is limited to 15, overall limiting n to 240 for the NFP Agilio CX.

Listing 5.6. Rule definition where the number of variables (grey boxes) is varied by adding additional definitions of variables along with a reference within a predicate.

```
var var_0 = 0;
var var_1 = 1;
complex_event sample_evt {
    value event.id
    strategy skip-till-next-match
    pattern  [ipv4.protocol == $var_0]  ; [ipv4.protocol == $var_1]
}
```

5.5.5.5 Number of complex event definitions

As shown in Listing 5.7, we increase the number of complex events n that are to be detected by adding additional definitions of complex events (of the same complexity) in the rule. Performance scales linearly with n with a latency increase of $\approx$1.3 µs and throughput decrease of $\approx$ 1.5 Mpps per additional complex event. Due to limitations of program memory and pipeline capacity of the Agilio CX, n is limited to ≤ 4.

Listing 5.7. Rule definition where the number of complex events to be detected (grey boxes) is varied by adding additional definitions of complex events.

```
complex_event sample_evt_0 {
    value event.id
    strategy skip-till-next-match
    pattern [ethernet.etherType == 2049]
}
complex_event sample_evt_1 { ... }
```

5.5.5.6 Number of event instances

In our last evaluation, we increase the number of event instances n (interleaved patterns to be detected in parallel), as shown in Listing 5.8. In order to ensure that all instances are in a state different from the starting state s_0. We measured a static latency offset of 1.5 µs for the second instance $n = 2$ and constant l_p for $n > 2$ as well as constant throughput.

Due to limitations of program memory and pipeline capacity of the Agilio CX, n is limited to ≤ 5.

Listing 5.8. Rule definition where the number of engine instances (grey boxes) is varied.

```
complex_event sample_evt {
    value event.id
    strategy skip-till-next-match
    instances 2
    pattern [ipv4.protocol == 17] ; [ipv4.protocol == 1234]
}
```

5.5.6 Evaluation Conclusions

We conclude from the evaluations that P4CEP achieves good performance, rendering our approach practical. On the NFP hardware target, it outperforms sate-of-the-art software-

based CEP systems by an order of magnitude with respect to latency and throughput for viable window sizes. In absolute terms, P4CEP on the NFP tremendously outperforms existing CEP systems with 9.8 µs end-to-end event detection latency at around 8.3 million events per second, when not using windows operations. Our scalability evaluations have shown, that overall, the extent of stateful processing incorporating registers, in particular window operations, has a negative impact on performance on the NFP. The main reason for this performance degradation can be attributed to the overhead due to the synchronization of register access and atomic code blocks (Limitation 3). Furthermore, due to hardware limitations, scalability with respect to the number of complex events to detect as well as the number of instances is limited, whereas in particular mechanisms with low register involvement, such as pattern and predicate complexity, show great scalability.

Given the impediments that come with the early stage of in-network computing and data plane processing, we identify performance potential in eliminating workarounds due to the experienced limitations of $P4_{14}$, for instance for the table indirection, greatly limiting the scalability of variables. Moreover, it should be noted that P4CEP's lack of optimizations of the compiler and specific hardware platform, except the window iteration, bears further potential for performance improvement. For instance, we expect an extension of the NFP's program memory by using its external memory unit to significantly increase the scalability of complex events and instances, as well as throughput improvements with optimized operator placement. Lastly, we expect great performance improvement on more optimized implementations and targets like the high-end Agilio NFP series, NetFPGA, or programmable data center switches.

5.6 Related Work

In this section, we briefly discuss related work other than already mentioned.

SNAP [AKG+16] is a network-centric, high-level language for network programming, extending stateless packet processing with primitive stateful operations. It offers a stateful network-wise abstraction for packet processing by enabling access to a persistent global array in control programs, while making the distribution of that state in the data plane transparent to the programmer. Packet processing in a SNAP program depends on the current state of the network, which is held in variables within the global array and which is possibly changed as a result of processing. SNAP considers events as non-frequent changes in the network, such as traffic changes and failures, that trigger recompilation of the network program in the control plane.

Stateful NetKAT is a language for event-driven network programming [MHFe16], extending the NetKAT language by mutable state. It is similar to SNAP, however, it rather focuses on applying consistent network updates where consistency properties hold during the transition between two network configurations, which is triggered in response to events.

While both approaches enable network-centric stateful packet processing, P4CEP is tailored for complex event processing with a generic notion of events that includes but is not limited to network events. Since P4CEP is based on P4, it is more lightweight while still leveraging the full expressiveness of P4.

OpenState [BBCC14] is an implementation of a generic state machine in the data plane of an OpenFlow switch. It maps the state machine execution to a fixed match-action pipeline consisting of two tables, holding the current state and transitions on a per-flow basis. Since state is held in flow-table entries, OpenState requires a custom OpenFlow instruction to be able to update the state after a transition. While the authors provide a modified implementation of an OpenFlow software switch, there are no hardware implementations. P4CEP uses P4 to implement its state machine logic without the need for modifications of software or hardware.

5.7 Conclusion

In this chapter, we presented P4CEP, an in-network implementation of Complex Event Processing. As a representative of the class of stateful processing applications, we showed the implications of CEP's distributed processing in our data plane implementation with regard to data plane consistency, where we discussed the need for a mechanism for correct multicast tree updates as well as the central role of data consistency with respect to inter-packet state and related capabilities of today's data plane programming. We conclude however, that some encountered limitations of P4 that limit its applicability for stateful processing have already been addressed in the evolution of the P4 language or are mitigated by target-specific extensions for data center switches or smart-NICs.

In the evaluation, we showed that P4CEP offers practical performance and good scalability, despite enforced partial linearization of the processing in order to ensure application correctness. On a programmable NIC, P4CEP outperforms the state-of-the-art software CEP system Apache Flink by an order of magnitude in latency (tens of microseconds range) and throughput (million events per second range) for a viable degree of stateful operations. We identified constrained scalability of simultaneous pattern detection due to

target-dependent resource constraints on the one hand but also identified a large potential for target-specific optimizations on the other hand.

6 Conclusion and Future Work

This chapter closes the thesis by summarizing its presented main contributions and providing an outlook on possible future work and research directions.

6.1 Conclusion

Recent years have seen a tremendous proliferation of Software-defined networking that has elevated SDN from a theoretical concept to large-scale adoption in real-world networks such as Data Center Networks and their interconnections through WANs of big industrial players like Google, Facebook, and Microsoft. SDN is a key enabler for high dynamics in terms of changes of the network configuration, for instance in the cloud computing context, and in terms of a high rate of innovation allowing experimentation and deployment of new network protocols and substantially expanding the networking paradigm by moving applications into the network at unprecedented pace and with small effort. Centered around the separation of the inherently distributed data plane from the logically centralized and hence possibly physically distributed control plane, aspects of consistency and distribution are naturally of utmost importance for SDN, especially when facing requirements on scalability, availability, and resiliency raised through practical adoption at scale. This thesis addresses various aspects of distribution and the problem space of consistency in Software-defined Networking. In particular, it focusses on the three areas of update consistency, flexibility in control plane distribution, and the in-network implementation of a distributed application, making the following contributions.

1. As a consequence of high network dynamics, the networks have to be frequently reconfigured. Reconfiguration of a network on the one hand profits from logically centralized control and global network view, on the other hand, it has shown to be an intricate and crucial process which may severely degrade the performance of a network and break network invariants, stipulated to ensure certain security or connectivity functions for instance. We present a management architecture that is

aware of expectable inconsistency effects due to reconfiguration and hence allows for an appropriate selection of an update mechanism and its parameters in order to prevent those effects.

2. A thorough investigation of update consistency in multicast networks leads us to the introduction of a novel correctness property, duplicate-freeness. When using a lightweight update mechanism, we show that it is impossible to avoid both dropping and duplication of packets. However, we present an analytical framework that allows identifying crucial update steps and enables feeding back those steps to the management architecture. Our evaluations show the practical relevance of the addressed problem even for moderate degrees of dynamics in real-world mid- to large-scale WANs.

3. We introduce a lightweight update mechanism for multicast route updates that allows for the selection of an update strategy preventing either dropped packets or duplicated packets. Evaluations show that prevention of drops results in 5% duplication rate and prevention of duplicates results in 9% drop rate on the aforementioned topologies. Furthermore, we present a mitigation approach implementing in-network filtering of duplicates. If both effects are to be prevented accepting higher cost, we provide an optimization of an existing powerful but resource-intensive update approach.

4. To provide flexibility with respect to distribution schemes and consistency in synchronization of control plane state in order to increase control responsiveness, we present ZeroSDN, an event-based architecture for flexible full-range control plane distribution. Our architecture is centered around a message bus concept decoupling controller functionality from each other and from switches through so-called control plane events and data plane events, respectively. The flexible dissemination scheme for those events, based on the content-based publish/subscribe paradigm, allows for a free combination of replication and partitioning of control functionality with different consistency semantics. Our implementation covers a large set of mechanisms for improving consistency and scalability of the control plane, such as the inherently implemented load-balancing, detection of policy conflicts, and a feedback mechanism for data plane updates. Evaluations show linear scalability when distributing compute-heavy control functions.

5. To further decrease control latency, we enable switch-local control decision making.

By placing control logic onto switches in combination with our message bus, we expand SDN's control paradigm and enable the full range from fully decentralized control, over local control still profiting from global view up to SDN-typical fully (logically) centralized control. Allowing to limit the scope the local logic operates on from full global view to solely local view allows to trade-off scope of state, synchronization overhead (and hence control latency), and quality of control decisions. We provide use cases for all control schemes, for instance presenting autonomous local procedures that without switch-external control allow for temporary fast yet possibly sub-optimal control decisions, for instance swiftly recovering local link failures, as well as "autonomous forwarding" featuring control coordination, reconciliation of policy conflicts, and fine-granular local aggregation of the global view. A performance comparison with established controller frameworks shows raw throughput without distribution of about 70% of the best-in-class controller. Local processing yields factors 2 to 6 of reduced latency compared to non-local processing. In absolute terms, the control latency with local logic on a recent top-of-rack 10GbE hardware switch has been measured to be as low as $\approx 330\,\mu s$.

6. In order to ensure a safe and steady control plane operation in the face of arbitrary switch-local control logic, we present mechanisms for enforcing isolation and fine-grained resource control for local control applications, based on lightweight virtualization techniques. Evaluations show that adequate isolation and resource-control can be reached at virtually no overhead using container technology.

7. In order to address SDN's latest evolution, we present P4CEP, an implementation of a distributed application from the domain of message-oriented middleware. We implement Complex Event Processing (CEP) on top of programmable network devices using data plane programming, in particular, the P4 language. Enabled by P4's flexible and powerful programming model, we present a data plane implementation of CEP that yields greatly reduced latency and increased throughput due to hardware-based packet processing. Since we do not want to swap a dedicated CEP middlebox for a dedicated network device solely implementing CEP, we implement a mechanism to allow the co-existence of CEP along arbitrary other network functions on the same device. Our implementation comprises a compiler that compiles patterns for the detection of complex events specified in our rule specification language to data plane programs, consisting of a state machine and operators that process so-called windows containing historic events.

8. We discuss challenges entailed in distributed data plane processing and address aspects of distribution and consistency in particular for stateful data plane programming, where packet processing changes internal state, which in turn changes the processing of subsequent packets. In summary, we identified the following limitations of P4 for stateful in-network computing: (1) the overhead of ensuring data consistency by synchronizing register access and atomic code blocks, (2) the inability to directly handling inter-packet state in registers, and (3) the indirection of action invocations. Although we understand some limitations as deliberate decisions in P4's design, we see great potential for constructs like bounded loops or more efficient primitives for synchronization. Evaluations on a programmable NIC show excellent performance of 9.8 µs end-to-end event detection latency at around 8.3 million events per second when not employing window operators, outperforming the software-based Apache Flink CEP system which yields about l_p =270 µs and B_p =3.5% (≈520 kpps) for windows sizes up to 1000. However, the programmable NIC shows limited scalability properties. Due to a limitation on the size of a P4 program, window sizes > 10 preclude loop-unrolling and hence require P4-extern mechanisms, for instance, a C-based sandbox in case of the evaluated programmable NIC. Overall, scalability evaluations show significant performance penalties when handling state data under tight consistency requirements, as (1) suggests.

6.2 Future Work

The research conducted in this thesis and its resulting approaches can be potentially extended by several subsequent research directions and refinements of approaches, respectively. In the following, we give a short list of the most promising future work and discussions in the context of consistency and distribution in Software-defined Networking.

- In the context of the update consistency, we argued for the incorporation of update inconsistency effects in the reconfiguration process triggered by a network manager. While we have shown specific relevance and approaches for the multicast paradigm, our generic update awareness architecture can be applied to a plethora of other applications and network functions, by investigating on application-specific optimization of update processes and possibly designing specific hybrid update approaches, combining stateless and stateful update mechanisms. For instance, more complex network functions from the virtualization within the domain of mobile networks

such as cloud RAN (cRAN) [CCY+15] or mobile edge computing (MEC) [MYZ+17] can be expected to be particularly susceptible to inconsistency effects.

- More specifically for our multicast update approach, while designing our rule update generator and update executor, we identified several opportunities to leverage parallelization of update execution which could be addressed in future work. Moreover, a combination of update scheduling with timed updates and high-precision time synchronization looks promising to further mitigate update inconsistency effects due to propagation delay.

- We also identified further improvements and extended scope of the presented concept of event-driven distributed network control. Our distributed SDN controller is based on content-based filtering of events, in particular, the filtering of data plane events based on header field matching. In larger networks, event notifications might arrive at a high rate, which makes content-based message filtering by the message bus challenging. In our prototype, we used a high-performance topic-based messaging system (ZeroMQ) as a workaround by mapping match fields onto a topic hierarchy. However, such a topic mapping also comes with inherent problems. More specifically, the discretization of the event space incurs lack of expressiveness. Also, attributes have to be specified according to the order given by the topic hierarchy. Irrelevant hierarchy levels can be wildcarded, however, efficient wildcard topic matching at high event rates is hard to implement in software. To solve this problem, an in-network implementation of content-based routing [BTK+17] could be employed. Furthermore, the evaluation of complex data plane events and complex data plane events could be offloaded to the network, using data plane programming techniques as proposed with P4CEP, however being deployed in the control plane network rather than the data plane network.

- As another extension of ZeroSDN, we can further leverage the publish/subscribe paradigm to build a *holistic distributed system controller* not limited to controlling the network elements but to include virtual network functions, end systems (including virtual machines), applications (e.g., client and server processes on the application layer), etc. In other words, we can extend the network control plane to a *holistic system control plane* implemented by a set of distributed controllets, which communicate indirectly through events including not only simple and complex data plane events but any event relevant for controlling and managing the holistic system. As a simple example, consider the migration of a virtual machine (VM), which

might also require the migration of virtual network functions like firewalls, and the adaptation of routes for chaining services. Using event-based communication, we can trigger actions to implement an event-triggered workflow defining the sequence of actions necessary to migrate the VM. For instance, as soon as the VM has been suspended by a VM controllet, an event could be fired that triggers the migration of network functions, which then trigger the adaptation of routes in the network through further events. This way, complex *system management workflows* can be implemented in a decentralized fashion.

- Based on our experiences gained from designing our in-network CEP implementation P4CEP, we argue that in-network computation, in particular for stateful processing, poses an interesting research question regarding the trade-off between portability (target-independence) and leveraging programmability, including application-specific custom functions (introducing target-dependence). For instance, while it was our design goal to stay target-independent through the exclusive use of a uniform data-plane programming language (P4), implementing custom functions enabled mitigation of current limitations of P4 and enriched functionality at the cost of becoming target-dependent. To further explore this trade-off, more powerful operators from CEP and stream processing can be adopted, including application-specific custom functions (introducing target-dependence). One example is a CEP operator for face recognition in a stream of image data that might be implemented on an FPGA. Another subsequent step is the implementation and investigation of target-specificity of P4CEP on data center switches with reconfigurable ASICs following the P4 Portable Switch Architecture (PSA), such as the Barefoot Tofino, as well as on hardware models specifically designed for a high degree of data consistency, offering *packet transactions* [SCB+16].

- Extensions more specific to our approach are time-based windows and other window semantics, e.g., tumbling windows. Since timestamps for packet ingress and egress are available as P4 intrinsic metadata, we expect their implementation with relative ease. Moreover, additional aggregation operators (or their approximation) such as average could be implemented.

- The combination of high-precision switch clock synchronization [KJC19] and very small jitter of processing latency due to hardware processing could furthermore lead to an adoption of in-network CEP in the domain of time-sensitive networks (TSN) [NDR16], providing real-time CEP with very tight jitter and latency bounds.

BIBLIOGRAPHY

[AAK14] N. L. M. v Adrichem, B. J. v Asten, and F. A. Kuipers. Fast Recovery in Software-Defined Networks. In *2014 Third European Workshop on Software Defined Networks*, pages 61–66, September 2014.

[AB14] R. Ahmed and R. Boutaba. Design considerations for managing wide area software defined networks. *IEEE Communications Magazine*, 52(7):116–123, July 2014.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[Ac04] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 456–467. VLDB Endowment, 2004.

[AFG+14] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[AKG+16] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[AM16] M. Aslan and A. Matrawy. Adaptive consistency for distributed sdn controllers. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 150–157, Sep. 2016.

[ARTN13] A. Arefin, R. Rivas, R. Tabassum, and K. Nahrstedt. OpenSession: SDN-based cross-layer multi-stream management protocol for 3d teleimmersion. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, October 2013.

[BAM09] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association. event-place: Boston, Massachusetts.

[BAM10] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[Bäu16] Christian Bäumlisberger. Plattformunabhängige lokale SDN-Controller auf offener Weiterleitungshardware durch Anwendung von Containertechnologie. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, November 2016.

[BBBS16] Roberto Bifulco, Julien Boite, Mathieu Bouet, and Fabian Schneider. Improving SDN with InSPired Switches. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM.

[BBCC14] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014.

[BBHH16] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 511–524, New York, NY, USA, 2016. ACM.

[BBHK15] A. Basta, A. Blenk, H. Belhaj Hassine, and W. Kellerer. Towards a dynamic SDN virtualization layer: Control path migration protocol. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 354–359, November 2015.

[BD17] Mihai Budiu and Chris Dodd. The p416 programming language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017.

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[BGH+14] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[BGK+13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[Bho17] Sukanya Bhowmik. *Content-Based Routing in Software-Defined Networks*. Doctoral thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, December 2017.

[Big] Big Switch Networks. Floodlight: An Open SDN Controller. `http://www.projectfloodlight.org/floodlight/`.

[BJ87] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[BMK+11] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with stream mapreduce. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 48–58, Nov 2011.

[BRKB13] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani. On the feasibility of a consistent and fault-tolerant data store for sdns. In *2013 Second European Workshop on Software Defined Networks*, pages 38–43, Oct 2013.

[Broa] Broadcom. BroadView: Analytics-Driven Dynamic Path Optimization. `https://www.broadcom.com/collateral/tb/BroadView-TB301-RDS.pdf`.

[Brob] Broadcom. Open Network Switch Layer (OpenNSL). `https://github.com/Broadcom-Switch/OpenNSL`.

[Broc] Broadcom. OpenFlow Data Plane Abstraction (OF-DPA). `https://github.com/Broadcom-Switch/of-dpa`.

[Brod] Broadcom. OpenFlow Data Plane Abstraction (OF-DPA™): Abstract Switch Specification Version 2.01. `https://github.com/Broadcom-Switch/of-dpa/blob/master/OFDPAS-ETP100-R.pdf`.

[BSA14] Alysson Bessani, João Sousa, and Eduardo Alchieri. State machine replication for the masses with bft-smart. pages 355–362, 06 2014.

[BSM18] F. Bannour, S. Souihi, and A. Mellouk. Distributed sdn control: Survey, taxonomy, and challenges. *IEEE Communications Surveys Tutorials*, 20(1):333–354, Firstquarter 2018.

[BTK+15] Sukanya Bhowmik, Muhammad Adnan Tariq, Boris Koldehofe, André Kutzleb, and Kurt Rothermel. Distributed control plane for software-defined networks: A case study using event-based middleware. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 92–103, New York, NY, USA, 2015. ACM.

[BTK+17] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Dürr, T. Kohler, and K. Rothermel. High Performance Publish/Subscribe Middleware in Software-Defined Networks. *IEEE/ACM Transactions on Networking*, PP(99):1–16, 2017.

[CCY+15] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann. Cloud ran for mobile networks—a technology overview. *IEEE Communications Surveys Tutorials*, 17(1):405–426, Firstquarter 2015.

[CEF+17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[CGLPN16] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 69–80, New York, NY, USA, 2016. ACM.

[CGSW14] Krzysztof Ciebiera, Piotr Godlewski, Piotr Sankowski, and Piotr Wygocki. Approximation Algorithms for Steiner Tree Problems Based on Universal Solution Frameworks. *arXiv:1410.7534 [cs]*, October 2014. arXiv: 1410.7534.

[Cha04] David Chappell. *Enterprise service bus*. " O'Reilly Media, Inc.", 2004.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[CKLS13] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. Software Transactional Networking: Concurrent and Consistent Policy Composition. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 1–6, New York, NY, USA, 2013. ACM.

[CKLS15] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *Proceedings of INFOCOM'15)*, April 2015.

[CM10] Gianpaolo Cugola and Alessandro Margara. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[CM12a] Gianpaolo Cugola and Alessandro Margara. Complex Event Processing with T-REX. *J. Syst. Softw.*, 85(8):1709–1728, August 2012.

[CM12b] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[CMT+11] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[Con] The P4 Language Consortium. Industrial Contributors to the P4 Language. https://p4.org/contributors/.

[Cor] Intel Corporation. Intel Ark - Intel Atom® Processor C2538. https://ark.intel.com/content/www/us/en/ark/products/77981/intel-atom-processor-c2538-2m-cache-2-40-ghz.html.

[Cor18] Corsa Technology Inc. Corsa DP6420 OpenFlow data plane. http://www.corsa.com/products/dp6420, August 2018.

[CPSC15] C. Cascone, L. Pollini, D. Sanvito, and A. Capone. Traffic Management Applications for Stateful SDN Data Plane. In *2015 Fourth European Workshop on Software Defined Networks*, pages 85–90, September 2015.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, 1996.

[Dee89] S.E. Deering. Host extensions for IP multicasting. RFC 1112 (Internet Standard), August 1989. Updated by RFC 2236.

[DHM+14] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. ElastiCon: An Elastic Distributed Sdn Controller. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 17–28, New York, NY, USA, 2014. ACM.

[Don] John Donovan. Setting the Pace with Our Next-Gen Network. http://about.att.com/innovationblog/121514settingthepace.

[DSC+15] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[DSH+10] A. Doria (Ed.), J. Hadi Salim (Ed.), R. Haas (Ed.), H. Khosravi (Ed.), W. Wang (Ed.), L. Dong, R. Gopal, and J. Halpern. Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810 (Proposed Standard), March 2010. Updated by RFCs 7121, 7391.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[Eur12] European Telecommunications Standards Institute (ETSI). Network Functions Virtualisation - An Introduction, Benefits, Enablers, Challenges and Call for Action. `https://portal.etsi.org/NFV/NFV_White_Paper.pdf`, October 2012.

[FBC+15] Yonghong Fu, Jun Bi, Ze Chen, Kai Gao, Baobao Zhang, Guangxu Chen, and Jianping Wu. A Hybrid Hierarchical Control Plane for Flow-Based Large-Scale Software-Defined Networks. *IEEE Transactions on Network and Service Management*, 12(2):117–131, June 2015.

[FBMP13] P. Fonseca, R. Bennesby, E. Mota, and A. Passito. Resilience of SDNs based On active and passive replication mechanisms. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 2188–2193, December 2013.

[Fen97] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236 (Proposed Standard), November 1997. Updated by RFC 3376.

[Fet16] Matthias Fetzer. Local Data Plane Event Handling in Software-defined Networking. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, April 2016.

[FGR+13] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, February 2013.

[FHF+11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[FLMS18] Klaus-Tycho Foerster, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Loop-free route updates for software-defined networks. *IEEE/ACM Trans. Netw.*, 26(1):328–341, February 2018.

[FMW16] Klaus-Tycho Foerster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9, 2016.

[FRZ14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.

[FSV19] K. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Communications Surveys Tutorials*, 21(2):1435–1461, Secondquarter 2019.

[GKP+08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38(3):105–110, July 2008.

[Goo] Google LLC. Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[Gre15] Albert Greenberg. Sdn for the cloud. In *Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[Gun18] Gunawardena, Dinan and Kicinski, Jakub. P4, eBPF and Linux TC offload. `https://open-nfp.org/the-classroom/p4-ebpf-and-linux-tc-offload/`, March 2018.

[GVHM13] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design Principles for Packet Parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 13–24, Piscataway, NJ, USA, 2013. IEEE Press.

[Hew16] Hewlett Packard Enterprise. HP Switch Software OpenFlow v1.3 Administrator Guide nl K/KA/KB/WB 15.18. `http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c04777809`, December 2016.

[HHLY14] Liang-Hao Huang, Hui-Ju Hung, Chih-Chung Lin, and De-Nian Yang. Scalable Steiner Tree for Multicast Communications in Software-Defined Networking. *arXiv:1404.3454 [cs]*, April 2014.

[Hir12] Martin Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–317, 1983.

[HSS+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014.

[HYG12] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[HYS13] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 43–48, New York, NY, USA, 2013. ACM.

[iC] iMatix Corporation. ZeroMQ: Distributed Messaging. `http://zeromq.org/`.

[IKM14] A. Iyer, P. Kumar, and V. Mann. Avalanche: Data center Multicast using software defined networking. In *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–8, January 2014.

[iMa] iMatix Corporation. CurveZMQ - Security for ZeroMQ. `http://curvezmq.org/page:read-the-docs`.

[Inc18a] Netronome Systems Inc. Netronome NFP-4000 Flow Processor – Product Brief. `https://www.netronome.com/m/documents/PB_NFP-4000.pdf`, August 2018.

[Inc18b] Netronome Systems Inc. NFP-4000 – Theory of Operation. `https://www.netronome.com/m/documents/WP_NFP4000_TOO.pdf`, August 2018.

[JdSM+18] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 26–32, New York, NY, USA, 2018. ACM.

[JKM+13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[JLG+14] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic Scheduling of Network Updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM.

[JRS11] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 245–256, 2011.

[JYVM15] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 103–115, Berkeley, CA, USA, 2015. USENIX Association.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[KC] Antti Kantee and Justin Cormack. Rump kernels: No os? no problem! In *;Login: USENIX Magazine, October 2014, Vol. 39, No. 5*. USENIX, USENIX Association.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[KCG+10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[KCGJ14] Anand Krishnamurthy, Shoban P. Chandrabose, and Aaron Gember-Jacobson. Pratyaastha: An Efficient Elastic Distributed SDN Control Plane. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 133–138, New York, NY, USA, 2014. ACM.

[KD17] Murat Karakus and Arjan Durresi. A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN). *Computer Networks*, 112:279–293, January 2017.

[KDBR17] Thomas Kohler, Frank Dürr, Christian Bäumlisberger, and Kurt Rothermel. InFEP - Lightweight Virtualization of Distributed Control on White-box Networking Hardware. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–6, Nov 2017.

[KDR15] Thomas Kohler, Frank Dürr, and Kurt Rothermel. Update Consistency in Software-defined Networking based Multicast Networks. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 177–183, November 2015.

[KDR16] Thomas Kohler, Frank Dürr, and Kurt Rothermel. Consistent Network Management for Software-Defined Networking Based Multicast. *IEEE Transactions on Network and Service Management*, 13(3):447–461, Sep. 2016.

[KDR17] Thomas Kohler, Frank Dürr, and Kurt Rothermel. ZeroSDN: A Highly Flexible and Modular Architecture for Full-range Network Control Distribution. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, ANCS '17, pages 25–37, Piscataway, NJ, USA, May 2017. IEEE Press.

[KDR18] Thomas Kohler, Frank Dürr, and Kurt Rothermel. ZeroSDN: A Highly Flexible and Modular Architecture for Full-Range Distribution of Event-Based Network Control. *IEEE Transactions on Network and Service Management*, 15(4):1207–1221, Dec 2018.

[KF13] Hyojoon Kim and N. Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, February 2013.

[KF19] Asterios Katsifodimos and Marios Fragkoulis. Operational stream processing: Towards scalable and consistent event-driven applications. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 682–685, 2019.

[KJ17] Patrick Kutch and Brian Johnson. SR-IOV for NFV Solutions - Practical Considerations and Thoughts. `https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf`, February 2017.

[KJC19] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 8–20, New York, NY, USA, 2019. ACM.

[KKR10] Gerald G. Koch, Boris Koldehofe, and Kurt Rothermel. Cordies: Expressive event correlation in distributed systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 26–37, New York, NY, USA, 2010. ACM.

[KMD+18] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. P4CEP: Towards In-Network Complex Event Processing. In *Proceedings of the ACM SIGCOMM 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 33–38, New York, NY, USA, 2018. ACM.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Kol14] Bikash Koley. Software defined networking at scale. `https://static.googleusercontent.com/media/research.google.com/de/pubs/archive/42948.pdf`, 2014.

[KPK15] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What You Need to Know About SDN Flow Tables. In Jelena Mirkovic and Yong Liu, editors, *Passive and Active Measurement*, number 8995 in Lecture Notes in Computer Science, pages 347–359. Springer International Publishing, March 2015.

[KPKC18] Maciej Kuzniar, Peter Peresíni, Dejan Kostic, and Marco Canini. Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches. *Computer Networks*, 136:22–36, 2018.

[KREV+15] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, January 2015.

[KRG+15] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, May 2015. USENIX Association.

[KRK+18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[KRW13] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental Consistent Updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 49–54, New York, NY, USA, 2013. ACM.

[KZFR15] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller Fault-tolerance in Software-defined Networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 4:1–4:12, New York, NY, USA, 2015. ACM.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[LHG+15] Y. Liu, A. Hecker, R. Guerzoni, Z. Despotovic, and S. Beker. On optimal hierarchical sdn. In *2015 IEEE International Conference on Communications (ICC)*, pages 5374–5379, June 2015.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[LLN+17] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 795–809, New York, NY, USA, 2017. ACM.

[LRFS14] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 15:1–15:7, New York, NY, USA, 2014. ACM.

[Luc08] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, pages 3–3, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[LWH+12] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically Centralized?: State Distribution Trade-offs in Software Defined Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[LWZ+13] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 411–422, New York, NY, USA, 2013. ACM.

[LYYW11] Dan Li, Jiangwei Yu, Junbiao Yu, and Jianping Wu. Exploring efficient and scalable multicast routing in future data center networks. In *2011 Proceedings IEEE INFOCOM*, INFOCOM '11, pages 1368–1376, April 2011.

[Maa18] Marius Maaß. In-network complex event processing using advanced data-plane programming. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, June 2018.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[Mar] Markets and Markets. Whitebox Server Market Report. `https://www.marketsandmarkets.com/Market-Reports/white-box-server-market-219773580.html`.

[MAR+14] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

[May18] Ruben Daniel Mayer. *Window-Based Data Parallelization in Complex Event Processing*. Doctoral thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, March 2018.

[McG12] Rick McGeer. A Safe, Efficient Update Protocol for Openflow Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 61–66, New York, NY, USA, 2012. ACM.

[MFC14] Jedidiah McClurg, Nate Foster, and Pavol Cerny. Efficient Synthesis of Network Updates. *arXiv:1403.5843 [cs]*, March 2014.

[MGBM17] A. S. Muqaddas, P. Giaccone, A. Bianco, and G. Maier. Inter-controller traffic to support consistency in onos clusters. *IEEE Transactions on Network and Service Management*, 14(4):1018–1031, Dec 2017.

[MHFe16] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. Event-driven Network Programming. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 369–385, New York, NY, USA, 2016. ACM.

[MHS11] Z. Milosevic, M. Hutle, and A. Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 235–244, 2011.

[MK17] O. Michel and E. Keller. Sdn in wide-area networks: A survey. In *2017 Fourth International Conference on Software Defined Systems (SDS)*, pages 37–42, May 2017.

[MKR15] R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, 2(4):274–286, Aug 2015.

[MRF+13a] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, April 2013. USENIX Association.

[MRF+13b] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *USENIX NSDI*, NSDI'13, page 2, 2013. Pyretic.

[MSM16] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates in software-defined networks. *IEEE/ACM Transactions on Networking*, 24(6):3412–3425, December 2016.

[MST+17] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. Spectre: Supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 161–173, New York, NY, USA, 2017. ACM.

[MW13] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 20:1–20:7, New York, NY, USA, 2013. ACM.

[MYZ+17] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys Tutorials*, 19(4):2322–2358, Fourthquarter 2017.

[nan17] nanomsg Documentation. Differences between nanomsg and ZeroMQ. `http://nanomsg.org/documentation-zeromq.html`, January 2017.

[NDR16] Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. Time-sensitive software-defined network (tssdn) for real-time applications. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 193–202, New York, NY, USA, 2016. ACM.

[Net14] Netronome Inc. The Joy of Micro-C. `https://open-nfp.org/media/documents/the-joy-of-micro-c_fcjSfra.pdf`, December 2014.

[Ng] Eugene Ng. Maestro: A System for Scalable OpenFlow Control.

[NGN16] Mehdi Nikkhah, Roch Guerin, and Mehdi Nikkhah. Migrating the internet to ipv6: An exploration of the when and why. *IEEE/ACM Trans. Netw.*, 24(4):2291–2304, August 2016.

[NHS12] Yukihiro Nakagawa, Kazuki Hyoudou, and Takeshi Shimizu. A Management Method of IP Multicast in Overlay Networks Using Openflow. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 91–96, New York, NY, USA, 2012. ACM.

[OF] OpenDaylight Foundation. OpenDaylight: Open Source SDN Platform. `https://www.opendaylight.org/`.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[OO14] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. pages 305–319, 2014.

[Opea] Open Compute Project. Networking Specs And Designs. `https://www.opencompute.org/wiki/Networking/SpecsAndDesigns`.

[Opeb] Open Networking Foundation. OpenFlow Switch Specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf.

[Ope18] Open-NFP.org. Open-Netronome Flow Processor (NFP) - Netronome. https://open-nfp.org/resources, March 2018.

[OSG] OSGi Alliance. Open Services Gateway initiative. https://www.osgi.org/.

[P4 18a] P4 Language Consortium. Behavioural Model (bmv2) – P4 software switch implementation. https://github.com/p4lang/behavioral-model, August 2018.

[P4 18b] P4 Language Consortium. P4→NetFPGA: A low-cost solution for testing P4 programs in hardware. https://p4.org/p4/p4-netfpga-a-low-cost-solution-for-testing-p4-programs-in-hardware.html, August 2018.

[P4 18c] P4 Language Consortium. The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf, November 2018.

[P4 18d] P4 Language Consortium. The P416 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf, November 2018.

[P4.18] P4.org Architecture Working Group. P416 Portable Switch Architecture (PSA). https://p4lang.github.io/p4-spec/docs/PSA-v1.1.0.pdf, November 2018.

[P4.19] P4.org API Working Group. P4Runtime Specification. https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0/P4Runtime-Spec.pdf, January 2019.

[PHJ+16] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, 2016. USENIX Association.

[PIC18] PICA8 Inc. Flow Scalability per Broadcom Chipset. https://docs.pica8.com/display/picos2102cg/Flow+Scalability+per+Broadcom+Chipset, March 2018.

[PKCK14] Peter Pereíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. Espres: Transparent sdn update scheduling. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 73–78, New York, NY, USA, 2014. ACM.

[PLH+15] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[PLS+06] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, April 2006.

[PLZ+15] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015.

[PPK+15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The Design and Implementation of Open vSwitch. pages 117–130, 2015.

[Pro] Open Compute Project. Switch Abstraction Interface (SAI). https://github.com/opencomputeproject/SAI.

[PZH+17] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. Scl: Simplifying distributed sdn control planes. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 329–345, Berkeley, CA, USA, 2017. USENIX Association.

[RDR10] S. Rizou, F. Durr, and K. Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *2010 Proceedings of 19th Inter-*

national Conference on Computer Communications and Networks, pages 1–6, Aug 2010.

[RFR+12] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[RFRW11] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent Updates for Software-defined Networks: Change You Can Believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 7:1–7:6, New York, NY, USA, 2011. ACM.

[RGW+18] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. Towards in-network industrial feedback control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 14–19, New York, NY, USA, 2018. ACM.

[RHC+17] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frederic Giroire, Joanna Moulierac, Dino Lopez Pacheco, and Guillaume Urvoy-Keller. Minnie: An sdn world with few compressed forwarding rules. *Computer Networks*, 121:185 – 207, 2017.

[RMF+13] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *;login:*, 38, 2013.

[RSU+12] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, PAM'12, pages 85–95, Berlin, Heidelberg, 2012. Springer-Verlag.

[Rus15] Mark Russinovich. Report from Open Networking Summit: Achieving Hyper-Scale with Software Defined Networking. `https://azure.microsoft.com/de-de/blog/report-from-open-networking-summit-achieving-hyper-scale-with-software-defined-networking/`, 2015.

[Ryu] Ryu SDN Framework Community. Ryu: Component-based Software Defined Networking Framework. `https://osrg.github.io/ryu/`.

[SAA+17] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[SCB+16] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM.

[SCF+12] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. PAST: Scalable Ethernet for Data Centers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[SCP+16] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[SK18] E. Sakic and W. Kellerer. Impact of adaptive consistency on distributed sdn applications: An empirical study. *IEEE Journal on Selected Areas in Communications*, 36(12):2702–2715, Dec 2018.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[SMS+12] Steven Smith, Anil Madhavapeddy, Christopher Smowton, Malte Schwarzkopf, Richard Mortier, Steven Hand, and Robert N. M. Watson. Draft: Have you checked your IPC performance lately? 2012.

[SOA+15] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[Son13] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[SS13] Stefan Schmid and Jukka Suomela. Exploiting Locality in Distributed SDN Control. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 121–126, New York, NY, USA, 2013. ACM.

[Str15] Jan Strauß. Control-plane Consistency in Software-defined Networking: Distributed Controller Synchronization using the ISIS2 Toolkit. Bachelor Thesis: University of Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems, May 2015.

[STT03] Ed Spitznagel, David Taylor, and Jonathan Turner. Packet classification using extended tcams. In *Proceedings of the 11th IEEE International Conference on Network Protocols*, ICNP '03, pages 120–, Washington, DC, USA, 2003. IEEE Computer Society.

[SY] Rob Sherwood and Kok-Kiong Yap. Cbench: an Open-Flow Controller Benchmarker. `http://www.openflow.org/wk/index.php/Oflops`.

[TCCP15] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou. Adaptive Resource Management and Control in Software Defined Networks. *IEEE Transactions on Network and Service Management*, 12(1):18–33, March 2015.

[TG10] Amin Tootoonchian and Yashar Ganjali. Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX Association, 2010.

[TH00] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991 (Informational), November 2000.

[Thea] The Linux Foundation. ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out. `http://onosproject.org/`.

[Theb] The NOX Controller. NOX Network Control Platform. `https://github.com/noxrepo/nox`.

[Thec] The Open Networking Foundation (ONF). Our Mission. `https://www.opennetworking.org/mission/`.

[TKBR14] Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, and Kurt Rothermel. PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 217–228, New York, NY, USA, 2014. ACM.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *Comm. Mag.*, 35(1):80–86, January 1997.

[ubA] u-blox AG. GPS-based Timing - Application Note. `https://www.u-blox.com/sites/default/files/products/documents/Timing_AppNote_%28GPS.G6-X-11007%29.pdf`.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[vADK14] N.L.M. van Adrichem, C. Doerr, and F.A. Kuipers. OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8, May 2014.

[VCB+15] S. Vissicchio, L. Cittadini, O. Bonaventure, G. G. Xie, and L. Vanbever. On the co-existence of distributed and centralized routing control-planes. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 469–477, April 2015.

[VMW] VMWare Inc. Open vSwitch – An Open Virtual Switch (Nicira Extensions). `https://git.io/vgTKL`.

[VTVR15] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central Control Over Distributed Routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 43–56, New York, NY, USA, 2015. ACM.

[WIS39] J. WISHART. Statistical Tables for Biological Agricultural and Medical Research. *Nature*, 144(3647):533–533, September 1939.

[WSD+17] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[WV18] Jim Wanderer and Amin Vahdat. Google Cloud using P4Runtime to build smart networks. `https://cloud.google.com/blog/products/gcp/google-cloud-using-p4runtime-to-build-smart-networks`, April 2018.

[XYL+17] Hongli Xu, Zhuolong Yu, Xiang-Yang Li, Liusheng Huang, Chen Qian, Taeho Jung, Hongli Xu, Zhuolong Yu, Xiang-Yang Li, Liusheng Huang, Chen Qian, and Taeho Jung. Joint route selection and update scheduling for low-latency update in sdns. *IEEE/ACM Trans. Netw.*, 25(5):3073–3087, October 2017.

[YDAG04] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and Control Element Separation (ForCES) Framework. RFC 3746 (Informational), April 2004.

[YG16a] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 4:1–4:12, New York, NY, USA, 2016. ACM.

[YG16b] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Second ACM Symposium on SDN Research (SOSR '16)*, Santa Clara, CA, March 2016.

[YTG13] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, February 2013.

[YZB+16] Ji Yang, Zhenyu Zhou, Theophilus Benson, Xiaowei Yang, Xin Wu, and Chengchen Hu. Focus: Function offloading from a controller to utilize switch power. 2016.

[Zel93] A. Z. Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9(5):463–470, May 1993.

[ZLT+18] J. Zheng, B. Li, C. Tian, K. Foerster, S. Schmid, G. Chen, and J. Wux. Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 12–21, July 2018.

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(Ort, Datum)	(Thomas Kohler)