Content-Based Routing in Software-Defined Networks

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

Sukanya Bhowmik

aus Kolkata, India

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. Thomas Plagemann
	Dr. rer. nat. Muhammad Adnan Tariq
Tag der mündlichen Prüfung:	01.12.2017

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2017

Acknowledgments

First and foremost, I would like to thank my doctoral advisor and mentor, Prof. Dr. Kurt Rothermel, without whom this research would not have been possible. He gave me an opportunity to be a part of his research group and continues to inspire me to keep working in academia. His constant support and guidance has been paramount throughout this journey, and I am extremely grateful for that.

I would like to offer my sincerest gratitude to Prof. Dr. Thomas Plagemann for kindly agreeing to be a part of my PhD committee and reviewing my thesis.

I would, also, like to thank my project supervisor, Dr. Muhammad Adnan Tariq, who has played a major role in quickly familiarizing me with this research area. I have greatly benefited from his keen scientific insight, will to excel, technical assistance, and outstanding mentoring. Special thanks go to Dr. Boris Koldehofe and Dr. Frank Dürr who have always inspired me and from whom I have had the privilege of learning a lot about research.

During my doctoral studies, I have had the privilege of interacting with my esteemed colleagues who have helped build a very friendly environment in the department. I wish to thank each one of them, especially, Naresh Nayak and Thomas Kohler for their valuable support and technical assistance in the field of software-defined networking. Special thanks go to Eva Strähle without whom I would be lost while dealing with administrative matters.

Finally, I would like to thank my family for supporting me during this process. My parents, Nandini Bhowmik and Milan Bhowmik, and my grandparents, Samir Mohan Chakravarty and Lily Chakravarty, have always supported me and encouraged me to pursue my goals. Special thanks go to my sister, Ananya Bhowmik Mitra, and my brother, Suman Mitra, for always motivating me to strive for excellence. Lastly, I would like to thank my husband, Sukalyan Roy, for his unending support and outstanding patience through the course of my doctoral journey. Truly, the contributions of all these people cannot be expressed merely in words.

Contents

Abstract **13**

Zusammenfassung **15**

1 Introduction **17**
1.1 State-of-the-Art Content-Based Pub/Sub Systems 18
1.2 Software-Defined Networking 20
1.3 Research Statement 22
1.3.1 Provision of In-network Content-based Filtering 22
1.3.2 Addressing Data Plane Limitations 22
1.3.3 Handling Control Plane Overhead 23
1.4 Contributions 23
1.5 Structure of the Thesis 25

2 In-network Content-based Filtering **27**
2.1 The PLEROMA Middleware 28
2.2 Content Representation 29
2.2.1 Spatial Indexing 29
2.2.2 Mapping a dz 31
2.3 Topology Reconfiguration 31
2.3.1 Maintenance of flow tables 33
2.3.1.1 Advertisements and Subscriptions 33
2.3.1.2 Flow installation 34
2.3.1.3 Unsubscriptions and Unadvertisements 38
2.4 Performance Evaluations 38
2.4.1 Experimental Setup 38
2.4.2 End-to-End Latency 40
2.4.3 False Positive Rate 42
2.4.4 Control Overhead 42
2.4.5 Discussion 43

2.5 Related Work 44
2.6 Conclusion 45

3 Expressive Mapping of Content Filters 47
3.1 Limitations of Content Representation 48
3.2 Workload-based Indexing 49
3.2.1 Selective Indexing 49
3.2.2 Adaptive Spatial Indexing 53
3.3 Dimension Selection 54
3.3.1 Event Variance 54
3.3.2 Subscription Matching 56
3.3.3 Correlation 58
3.3.3.1 Calculating Covariance Matrix 59
3.3.3.2 Performing Principal Component Analysis 60
3.3.4 Evaluation-based Techniques 61
3.4 Handling Dynamic Network Updates 62
3.4.1 Data Plane Consistency in PLEROMA 63
3.4.2 Light-Weight Approach 64
3.5 Performance Evaluations 67
3.5.1 Experimental Setup 67
3.5.2 Workload-based Indexing 69
3.5.3 Dimension Selection 69
3.5.3.1 False Positive Rate 71
3.5.3.2 Runtime Overhead 72
3.5.4 Combining Approaches 74
3.5.5 Handling Dynamics 74
3.5.6 Discussion 76
3.6 Related Work 77
3.7 Conclusion 78

4 Expressive Filtering by Combining Application Layer 79
4.1 System Architecture 80
4.2 Filter Selection Problem 81
4.3 Filter Benefit and Penalty Calculation 82
4.3.1 Benefit 83
4.3.2 Penalty 85
4.4 Selection Algorithms 85
4.4.1 Switch Selection Algorithm 85
4.4.2 Cluster-based Selection Algorithm 88
4.4.3 Network Updates 91
4.5 Further Optimizations 91

4.6 Performance Evaluations 92
4.6.1 Experimental setup 92
4.6.2 Comparing with State-of-the-Art 93
4.6.3 Impact of Threshold Factor 94
4.6.4 SSA vs CSA 95
4.6.5 Discussion 96
4.7 Related Work 97
4.8 Conclusion 98

5 Addressing TCAM Limitations 99
5.1 Impact of TCAM Limitations on PLEROMA 100
5.2 Filter Aggregation Problem 101
5.2.1 Problem Statement 101
5.2.2 Problem Analysis 102
5.3 Filter Aggregation Algorithm 104
5.3.1 Filter Aggregation on a Switch 104
5.3.1.1 Determining Possible Flow Merges 105
5.3.1.2 Selecting Flow Merges on a Switch 106
5.3.2 Aggregation Cost at a Merge Point 107
5.3.2.1 Incoming Traffic 109
5.3.2.2 False Positives on Downstream Paths 111
5.3.3 Resolving Dependencies Between Switches 113
5.3.4 Handling Dynamics 113
5.3.4.1 Basic Local Aggregation (LA-B) 114
5.3.4.2 Cost-based Local Aggregation (LA-C) 114
5.3.5 Ensuring Data Plane Consistency 115
5.4 Performance Evaluations 116
5.4.1 Experimental Setup 117
5.4.2 Comparing Network False Positive Rate 117
5.4.3 Comparing Runtime Overhead 120
5.4.4 Impact of Sampling Factor 121
5.4.5 Dynamic Behavior 122
5.4.6 Discussion 124
5.5 Related Work 124
5.6 Conclusion 125

6 Scaling the Control Plane 127
6.1 Distributed Control Plane - System Architecture 128
6.2 Control Plane Consistency in Pub/Sub 130
6.3 Scaling Approaches 133
6.3.1 Shared Everything Approach 133

6.3.2 Shared Nothing Approach . . . 137
6.3.2.1 Topology Reconfiguration . . . 138
6.3.2.2 Adaptive Load Balancing . . . 138
6.4 Keeping DP-config Consistent with CP-config . . . 140
6.5 Reducing Flow Operations . . . 141
6.6 Performance Evaluations . . . 143
6.6.1 Experimental Setup . . . 143
6.6.2 Vertical Scaling . . . 144
6.6.3 Horizontal Scaling . . . 146
6.6.4 Reducing Flow Operations . . . 148
6.6.5 Discussion . . . 149
6.7 Related Work . . . 150
6.8 Conclusion . . . 151

7 Summary and Future Work **153**
7.1 Summary . . . 153
7.2 Future Work . . . 155

Bibliography **159**

List of Figures

1.1 Broker-based Pub/Sub System 18
1.2 SDN Architeture 20

2.1 PLEROMA Middleware 28
2.2 Spatial Indexing 30
2.3 Forwarding in the switch network. Match fields of flows in R_1, R_2, R_4-R_6 are shown as dzs. Flows follow the notation MF $\rightarrow$ IS : PO 35
2.4 Flow maintenance on the arrival of S_3. 36
2.5 Flow maintenance on the departure of S_3 37
2.6 Testbed Topology 39
2.7 End-to-End Latency 41
2.8 False Positive Rate 42
2.9 Average Processing Latency 43

3.1 Limitations of Content Representation 48
3.2 Avoiding Empty Subspaces 50
3.3 Selective Indexing 51
3.4 Adaptive Spatial Indexing 53
3.5 Effects of event distribution 55
3.6 Event-based Selection 55
3.7 Subscription-based Selection 57
3.8 Light-Weight Approach 65
3.9 Versioning vs light-weight approach (LWA) 66
3.10 Performance Evaluations: Workload-based indexing 68
3.11 Performance Evaluations: Dimension Selection - False Positive Rate . . 71
3.12 Performance Evaluations: Dimension Selection - Runtime Overhead . . 73
3.13 Performance Evaluations: Combined Approaches 75
3.14 Performance Evaluations: Handling Dynamics 76

4.1 Hybrid Content-based Routing 80

4.2 False Positive Detection 83
4.3 Partial Overlap 83
4.4 Switch Selection 87
4.5 Cluster-based Selection 89
4.6 Performance Evaluations : Comparing with State-of-the-Art 93
4.7 Performance Evaluations: Impact of Threshold Factor 95
4.8 Performance Evaluations: SSA vs CSA 96

5.1 Importance of upstream switch filters 102
5.2 Merge Point Tree for Incoming Port 1 105
5.3 Cost Calculation 110
5.4 Performance Evaluations: False Positive Rate 118
5.5 Performance Evaluations: Runtime Overhead 120
5.6 Performance Evaluations: Impact of Sampling Factor 121
5.7 Performance Evaluations: Dynamic Behavior 123

6.1 System Architecture 129
6.2 Concurrent Access to CP-config and DP-config 130
6.3 Control Plane Inconsistency 132
6.4 Shared Everything Approach 134
6.5 Shared Nothing Approach 137
6.6 Reducing Flow Operations 142
6.7 Performance Evaluations : Vertical Scaling 145
6.8 Performance Evaluations : Horizontal Scaling 147
6.9 Performance Evaluations : Reducing Flow Operations 149

List of Abbreviations

API	Application Program Interface
ASI	Adaptive Spatial Indexing
BRS	Brute-Force Selection
CORBA	Common Object Request Broker Architecture
CS	Correlation-based Selection
CSA	Cluster-based Selection Algorithm
FA	Filter Aggregation Algorithm
FA-LB	Filter Aggregation Algorithm: Load-Based Method
FA-PB	Filter Aggregation Algorithm: Pattern-Based Method
FPR	False Positive Rate
FNR	False Negative Rate
DC	Data Center
DHT	Distributed Hash Table
EMCS	Event Match Count-based Selection
EVS	Event Variance-based Selection
GS	Greedy Selection
IP	Internet Protocol
LWA	Light-Weight Approach
NS	Network State
NYSE	New York Stock Exchange
PCA	Principal Component Analysis
P2P	Peer-to-Peer
RI	Regular indexing
RPC	Remote Procedure Call
RS	Random Dimension Selection
SDN	Software-Defined Networking
SEA	Shared Everything Approach
SI	Selective Indexing
SNA	Shared Nothing Approach

SNA-LB	Shared Nothing Approach with Load Balancing
SSA	Switch Selection Algorithm
TCAM	Ternary Content Addressable Memory

Abstract

Content-based routing, as provided by publish/subscribe systems, has emerged as a universal paradigm for interactions between loosely coupled application components, i.e., content publishers and subscribers, where published content is filtered and forwarded by content filters to interested subscribers. Over the past few decades, content-based publish/subscribe has been primarily implemented as an overlay network of software brokers. Even though these systems have proven to efficiently support content-based routing between a large number of distributed application components, such broker-based routing and content filtering in software results in performance (w.r.t. end-to-end latency, throughput rates, etc.) that is far behind the performance of network layer implementations of communication protocols.

As a result, the goal of this thesis is to develop methods that enable content-based filtering and routing at line-rate in the network layer by exploiting the capabilities of Software-Defined Networking (SDN). In particular, this thesis focuses on realizing a high performance SDN-based publish/subscribe middleware, called PLEROMA, while addressing major obstacles raised by data (forwarding) plane and control plane limitations of software-defined networks. More specifically, the following contributions are made in this thesis.

Our first contribution is to provide methods to fulfill the functional requirements of the content-based publish/subscribe paradigm on the network layer in order to enable line-rate filtering and forwarding of published content in the data plane. We propose methods to establish paths between publishers and their relevant subscribers by installing content filters directly on hardware switches in the data plane. While the developed methods result in a publish/subscribe middleware whose performance (w.r.t. end-to-end latency, throughput rates, etc.) is significantly better than state-of-the-art solutions, a network layer implementation faces some serious challenges due to inherent limitations of software-defined networks.

In fact, our next three contributions focus on addressing the problems associated with expressive filtering of content in the network layer, i.e., on hardware switches in the

data plane, in the presence of hardware limitations. In particular, we address limitations w.r.t. limited flow table size and limited number of bits available for filter representation in hardware switches that curtail the expressiveness of content filters. Our contributions include various methods that use the knowledge of workload in the system to mitigate the adverse effects of these data plane limitations, thus improving bandwidth efficiency in the system.

Not just the data plane, but also the control plane can have its own limitations (w.r.t. scalability in the presence of dynamically changing subscription requests) which can pose as a significant bottleneck for content-based routing on software-defined networks. As a result, our final contribution is to provide methods that enable concurrent and consistent control distribution, thus paving the way for a scalable and distributed control plane solution to high dynamics in an SDN-based publish/subscribe system.

Zusammenfassung

Inhaltsbasiertes Routing (Content-based Routing), wie es etwa von Publish/Subscribe-Systemen bereitgestellt wird, hat sich zu einem universellen Paradigma für Interaktionen zwischen lose-gekoppelten Anwendungskomponenten, den Inhaltsherausgebern (Publisher) und Inhaltsbeziehern (Subscriber), entwickelt, in dem veröffentlichte Inhalte durch Inhaltsfilter gefiltert und an interessierte Bezieher weitergeleitet werden. In den vergangen Jahrzenten wurde inhaltsbasiertes Publish/Subscribe überwiegend durch ein Overlaynetzwerk aus softwarebasierten Vermittlern (Broker) umgesetzt. Obwohl diese Systeme inhaltsbasiertes Routing zwischen einer großen Anzahl von verteilten Anwendungskomponenten effizient umsetzen können, liegt die Softwareimplementierung von vermittlungsbasiertem Routing und Inhaltsfilterung bezüglich Ende-zu-Ende Latenz und Durchsatz weit hinter der möglichen Leistung einer Implementierung von Kommunikationsprotokollen direkt auf der Netzwerkschicht zurück.

Dementsprechend ist es das Ziel dieser Dissertation, Methoden zu entwickeln, die inhaltsbasierte Filterung und –Routing bei vollem Leitungsdurchsatz der Netzwerkschicht durch Ausnutzung der Fähigkeiten der softwaredefinierten Vernetzung (Software-Defined Networking, SDN) ermöglichen. Diese Dissertation konzentriert sich dabei auf die Umsetzung einer hoch performanten SDN-basierten Publish/Subscribe-Diensteschicht (Middleware) namens PLEROMA und die Lösung der durch Einschränkungen der Datenweiterleitungsschicht (Data Plane) und Kontrollschicht (Control Plane) von softwaredefinierten Netzwerken bestehenden Herausforderungen. Insbesondere werden die folgenden Kernbeiträge im Rahmen der Dissertation vorgestellt.

Ein erster Beitrag besteht in der Ausarbeitung von Methoden zur Erfüllung der funktionellen Anforderungen an inhaltsbasiertes Publish/Subscribe auf der Netzwerkschicht, um Filterung und Weiterleitung von veröffentlichten Inhalten in der Datenebene bei vollem Leitungsdurchsatz zu erreichen. Dabei werden Methoden vorgestellt um Kommunikationspfade zwischen Herausgebern und dessen Beziehern aufzubauen, basierend auf der Einrichtung von Inhaltsfilterung direkt auf den Hardwareswitches in der Datenebene. Die entwickelten Methoden resultieren in einem Publish/Subscribe-System dessen Performanz bezüglich Ende-zu-Ende-Latenz und Durchsatz signifikant besser

sind als moderne, vergleichbare Lösungen, dessen Implementierung auf der Netzwerkebene aber aufgrund inhärenter Einschränkungen von softwaredefinierten Netzwerken große Herausforderungen birgt.

Dementsprechend konzentrieren sich die drei darauffolgenden Beiträge auf Probleme bezüglich der Ausdrucksmächtigkeit von Filtern auf der Netzwerkebene, also auf Hardwareswitches in der Datenebene, unter Berücksichtigung der vorherrschenden Einschränkungen. Im Speziellen wird auf die hardwarebedingten Limitierungen bezüglich der begrenzten Größe der Flow-Tabelle und der begrenzten Anzahl an für die Filterung zur Verfügung stehenden Bits, was die Ausdrucksmächtigkeit der Inhaltsfilterung schmälert, eingegangen. Die geleisteten Beiträge beinhalten vielfältige Methoden, die Wissen über die Arbeitslast im System nutzen um zuwiderlaufende Effekte aufgrund der genannten Einschränkungen in der Datenebene abzumildern, wodurch die Bandbreiteneffizienz des Systems verbessert wird.

Zusätzlich zur Datenebene, weist auch die Kontrollebene spezifische Einschränkungen auf, etwa bezüglich der Skalierbarkeit unter Vorherrschen von sich dynamisch ändernden Bezugsanfragen, was ein bedeutender Kapazitätsengpass für inhaltsbasiertes Routing in softwaredefinierten Netzwerken darstellen kann. Deswegen stellt der letzte Beitrag Methoden zur Verfügung, die eine nebenläufige und konsistente Kontrollverteilung ermöglichen, was den Grundstein für eine skalierbare und verteilte Kontrollebene unter hoher Dynamik in einem SDN-basierten Publish/Subscribe-System legt.

Chapter 1

Introduction

In today's fast-paced world, modern applications have very demanding requirements in terms of the manner in which they connect and communicate with each other. These applications include stock exchange [EFGK03, Bet00], instant news delivery [CLS03], online gaming [KTKR10], network monitoring [MLJ10], road traffic monitoring [MC02], workflow management [CDNF01], etc. The sheer amount of distributed components communicating in such applications, their dynamic behavior patterns, the growing amount of information exchanged between them, and the need for this exchange to be performed in a timely and bandwidth-efficient manner demand a high performance communication middleware. This is where classical abstractions of paradigms based on a request/response model of interaction (e.g., RPC [TA90], CORBA [tOMG], etc.) fall short, thus prompting the emergence of a more efficient and flexible paradigm, namely, the publish/subscribe communication paradigm.

Over the past few decades, the publish/subscribe or pub/sub paradigm has been widely adopted by modern applications to perform one-to-many or many-to-many communication. The foremost strength of the pub/sub paradigm lies in its ability to provide an efficient platform that enables asynchronous and decoupled interactions between multiple distributed components. More specifically, a pub/sub system consists of primarily two participating components—publishers (producers of information) and subscribers (consumers of information)—where the communication between the participants is not dictated by their identities but by the content of published information itself. So, publishers, which have no information on existing subscribers, simply publish information within the network in the form of *events*. The subscribers, also oblivious of the existence of the publishers, express their interest in receiving specific events with the help of *subscriptions*. Similar to subscriptions, publishers may, additionally, express the content they intend to publish with the help of *advertisements*. To maintain anonymity and to ensure that each event published by a publisher is delivered to all its interested subscribers, a logical intermediary, i.e., the pub/sub infrastructure, receives

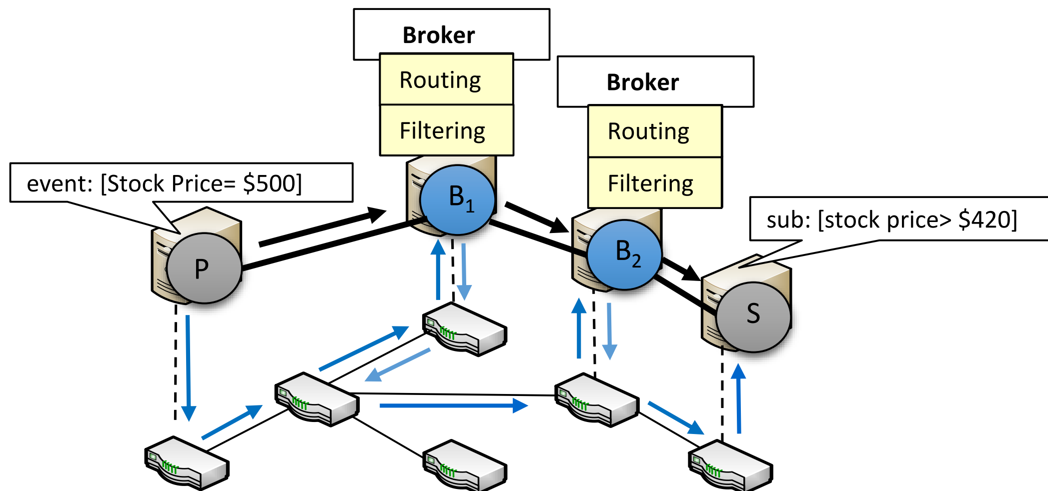


Figure 1.1: Broker-based Pub/Sub System

events from publishers and notifies relevant subscribers based on the content of published information. Clearly, the performance of the logical intermediary is crucial in determining the efficiency of a pub/sub system. However, before discussing the manner in which the logical intermediary has been realized over the past few decades (cf. Section 1.1), it is important to discuss the manner in which subscriptions, advertisements, and events are expressed in a pub/sub system, as this forms the basis of the pub/sub paradigm.

Depending on the degree of expressiveness of subscriber interests, pub/sub has been broadly classified into variants. However, the focus of this thesis is the most expressive variant of all—the very popular content-based pub/sub—which enables subscribers to specify powerful and expressive *subscriptions* that are then used as complex content filters for events. In a content-based model, an event is represented as attribute-value pairs, while an advertisement or subscription (i.e., content filter) is represented as a conjunction of constraints over these attributes. For example, Figure 1.1 depicts a simple stock quote dissemination system where an event e, published by publisher P, is represented as an attribute-value pair, i.e., [stock price= $500], and a subscription sub, issued by a subscriber S, is represented as a constraint over the attribute, i.e., [stock price $>$ $420]. As e satisfies the constraint of the filter sub, i.e., the event is relevant to the subscriber S, the logical intermediary should route e to S.

1.1 State-of-the-Art Content-Based Pub/Sub Systems

Over the past couple of decades, content-based filtering and routing of events to interested subscribers has been largely implemented on an overlay network of software servers, more commonly known as *brokers* [JCL+10, CDNF01, CRW01, CRW00, CS04,

Müh02, MFB02]. More specifically, a dedicated set of brokers adopt the role of the logical intermediary. Brokers collect subscriptions (representing content filters) from the subscribers in the network and compose routing tables with this information. So, when a publisher publishes an event, it gets filtered against the routing state maintained at the brokers and forwarded accordingly along the paths between the publisher and all subscribers interested in the published event. Referring again to Figure 1.1, the depicted stock quote dissemination system is a simple example of a broker-based network where brokers B_1 and B_2 collect subscriptions and maintain routing state. In this example, a path through the network of brokers is established between P and S for all events satisfying the content filter *sub*. So, the event e, first, gets filtered against the routing state at B_1 and on account of a match gets routed to B_2. Similarly, based on its maintained routing state, B_2 filters and routes e to the interested subscriber S. The recent past has also seen the advent of P2P-based (peer-to-peer) publish/subscribe systems [TKKR09, TBF⁺03, Tar13, BDFG07] where the role of the brokers is adopted by the participants of the system themselves, i.e., publishers and subscribers, that are organized into forwarding overlays.

However, whether in an overlay network of brokers or in a P2P-based architecture, both content-based filtering and routing of events are performed in software, i.e., in the application layer. As a result, their performance is far behind the performance of communication protocols implemented on the network layer w.r.t. throughput rates, end-to-end latency, and bandwidth efficiency. This is because broker-based pub/sub implementations are unable to exploit the performance benefits of standard multilayer switches or hardware routers capable of forwarding packets at line-rate and achieving data rates of 10 Gbps and more using dedicated hardware such as Ternary Content Addressable Memory (TCAM). Moreover, routing on overlay networks may not be bandwidth-efficient as the routing tables at brokers take the overlay topology into consideration which typically differs significantly from the underlying topology. This results in the dissemination of the same packet multiple times over the same physical link being shared by multiple logical links. For example, in Figure 1.1, the event e needs to traverse the same physical links multiple times while being routed by the brokers from the publisher to the interested subscriber. This is in sharp contrast to routing in the network layer.

Recent analysis has shown that addressing non-functional requirements of modern applications, such as increased throughput rates, reduced latency, increased scalability, increased bandwidth efficiency, etc., is of immense value. For example, according to InformationWeek [Mar07], electronic trading now makes up around 70% of the daily volume on the NYSE of which close to half is algorithmic trading. Studies further show that even a one-millisecond advantage in trading applications can be worth $100 million a year to a major brokerage firm [Mar07]. Not only in algorithmic trading but also in the online gaming industry, latencies larger than 50 milliseconds hurt user experience [PW02]. Moreover, recent studies on the online industry also show the importance

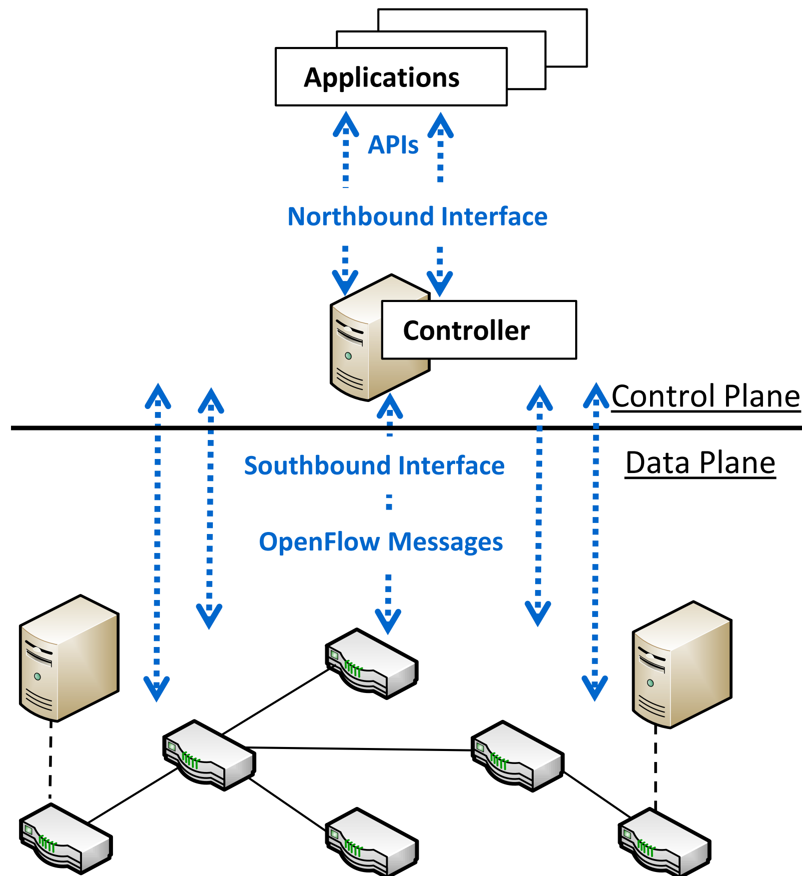


Figure 1.2: SDN Architeture

of high performance communication. For example, a 100 millisecond latency penalty implies a 1% sales loss for Amazon [Hof09]. Considering such massive demands of modern applications, it would be highly attractive to implement content-based routing directly on the network layer. However, even till the recent past, changes to existing standard network protocols and hardware seemed to be unrealistic and most research refrained from network layer implementations. This, however, has changed with the advent of software-defined networking (SDN), which provides the possibility to go beyond the limitations of traditional network architectures by allowing software to flexibly configure the network.

1.2 Software-Defined Networking

Software-Defined Networking (SDN) has significantly impacted programmable and active network evolution. In fact, SDN has been extensively considered for automated interconnection, dynamic resource sharing, WAN VPN, etc., across data centers over

the past few years. For example, for almost a decade, Google has been exploiting the benefits of SDN to power one of the world's biggest WANs, i.e., Google's data center (DC) WAN B4 [JKM+13]. Google uses SDN to connect 12 DCs while increasing the capacity of a single DC by more than 100x. Microsoft, too, has been harnessing the power of SDN to flexibly and reliably operate one of the largest public clouds in the world, i.e., Microsoft Azure [Azu15].

The main advantage of SDN is that it allows the abstraction of lower-level functionalities and presents them as network services. This is made possible through the establishment of a clear separation between the control plane and the data (forwarding) plane. More specifically, SDN enables the extraction of all control logic from network (forwarding) devices (such as switches) and hosts this control logic on a logically centralized server called the *controller*. In the traditional approach, each network device maintains information about its neighboring devices and makes forwarding decisions based on this information. However, SDN allows the logically centralized controller to capture a global view of the entire network and use this view to make efficient centralized decisions while updating the network.

The control plane in software-defined networks provides two interfaces—the northbound interface and the southbound interface. The northbound interface acts as the connection between the SDN controller and the applications and services running over the network. So, the northbound application program interfaces (APIs) can be used to orchestrate the network via the SDN controller such that it is aligned to the needs of various applications. Meanwhile, it is the southbound interface that connects the controller to the actual physical network. With the help of standards like OpenFlow [Com12], a popular southbound interface to the network, the controller has the ability to collect and process information (e.g., network statistics, application-specific requests) from the data plane and perform network updates accordingly by modifying the state of network devices (i.e., switches). A brief overview of the SDN architecture is illustrated in Figure 1.2.

Given the flexibility of SDN, the possibility of exploiting the technology to design a high performance content-based publish/subscribe middleware on the network layer seemed very promising and plausible but was yet to be explored in literature. A network layer implementation implies that the expressive filtering of events, which has been done at the application layer over the past two decades, should now be performed on the Ternary Content Addressable Memory (TCAM) of OpenFlow-enabled switches (in the date plane) at line-rate. Moreover, since the logically centralized controller has a global view of the underlying topology, it should also be possible to install a network topology for forwarding information between publishers and subscribers in a bandwidth-efficient manner. As a result, in this thesis, we exploit the capabilities of SDN to realize high performance content-based routing on software-defined networks.

1.3 Research Statement

This thesis primarily focuses on three problem areas with respect to the design of a high performance content-based publish/subscribe middleware.

1.3.1 Provision of In-network Content-based Filtering

The first problem that this thesis focuses on is the provision of in-network content-based filtering and routing of events in a publish/subscribe middleware such that line-rate performance can be achieved. As discussed previously, in order to achieve line-rate forwarding of events, content-based routing needs to be implemented directly on the network layer such that the filtering of events can be pushed to the data plane. This implies that content filters, which traditionally reside on the application layer, need to, now, be mapped to content filters that are capable of being installed in the TCAM of switches. In fact, content filters need to be represented by forwarding rules or flow entries in switches. This can prove to be extremely challenging.

Moreover, it is crucial that, in the presence of dynamic subscription requests, necessary paths are deployed between publishers and relevant subscribers such that no *false negatives*, i.e., events that are not forwarded to subscribers interested in receiving them, occur in the system. Also, necessary paths need to be removed from the network when subscribers unsubscribe. Performing topology reconfiguration in a resource-efficient manner to deploy or remove paths in the network can prove to be very challenging in software-defined networks. So, the first step towards high performance content-based routing would be to provide the basic functionalities of publish/subscribe in the network layer.

1.3.2 Addressing Data Plane Limitations

Most modern applications require not only line-rate forwarding of events but also support for bandwidth-efficient communication. While in SDN, the controller can use the global view of the actual physical network to ensure that an event packet does not traverse the same link multiple times, another very important factor influences bandwidth-efficient pub/sub communication. In content-based pub/sub, the expressiveness of a filter largely impacts the bandwidth efficiency of a system as it determines the amount of unnecessary traffic in the network, i.e., events that are forwarded to uninterested subscribers, commonly known as *false positives*.

As a result, a significant amount of work has been dedicated to the efficient matching of events against expressive filters in broker-based overlay networks [JCL+10, CRW01, JMVM09]. Implemented in software, these content filters have limitless potential w.r.t. expressiveness. However, such flexibility is not available to content filters installed in

the TCAM of software-defined networks due to certain inherent hardware limitations. TCAM is an extremely expensive and power-hungry resource. As a result, the design of a hardware switch only allows a limited number of flow entries (forwarding rules) in TCAM. Please note that, as discussed in Section 1.3.1, a flow entry represents a content filter. Moreover, only a limited number of bits in a flow entry can be made available to represent a content filter. As a result, only a limited number of content filters can be installed in TCAM where each individual content filter is further restricted by the bit length available to it within a flow entry. So, the question is, how can expressive filtering of events in the network layer be achieved despite inherent hardware limitations? As a result, the second problem area that this thesis focuses on is the need to employ additional mechanisms to install powerful and expressive filters on hardware switches while performing content-based routing on software-defined networks.

1.3.3 Handling Control Plane Overhead

While modern applications require high performance in terms of latency, throughput rates, and bandwidth efficiency, the need for a scalable solution is paramount in a pub/sub middleware. Considering the sheer number of modern application users with dynamically changing requests, it is only befitting to focus on scalability in pub/sub systems. For example, financial trading, traffic monitoring, or online gaming are known to be not only highly latency sensitive applications but also highly dynamic with respect to the interests of publishers and subscribers [JJE10, KORR12]. In order to analyse the trend of stocks and quotes, the threshold for receiving events is updated very frequently for a single subscription [JJE10]. Traffic monitoring and online gaming require location-dependent updates of run-time parameters such as the location of objects, often at higher frequency than one update per minute per subscriber [KORR12].

While various pub/sub middleware implementations that address scalability in overlay networks exist in literature [JJE10, CFMP04, LYK+11], these cannot be directly employed to an SDN solution where, in the presence of high dynamics, the logically centralized control plane must engage in very frequent network topology updates with changing interests of publishers and subscribers. Providing a scalable control plane with high responsiveness to such topology change requests in a dynamically changing environment is, therefore, crucial to the middleware and constitutes the final problem area explored in this thesis.

1.4 Contributions

This thesis combines and extends the findings and contributions of the work presented in [TKBR14], [BTK+17], [BTGR16], [BTHR16], [BTBR17], [BTK+15] towards realizing a high performance content-based pub/sub middleware on software-defined networks. More specifically, the main contributions of this thesis are as follows.

1. We provide methods to implement the basic functionalities of the content-based publish/subscribe paradigm on the network layer (cf. Section 1.3.1). Our designed system, PLEROMA, leverages the capabilities of software-defined networking to enable line-rate forwarding of published events. To this end, we provide mechanisms to represent content filters in a manner in which they can be installed on hardware switches, i.e., they can be represented by flow entries, such that filtering of published events can be pushed to the network layer. Our designed algorithms use these content filters to build a topology in the data plane, consisting of paths established between publishers and subscribers, for event dissemination. Moreover, PLEROMA offers methods to efficiently reconfigure a deployed topology in the presence of dynamically changing subscriptions and advertisements. This contribution is primarily based on the works published in [TKBR14] and [BTK+17]. The author of this thesis contributed around 45% and 70% of the scientific content in [TKBR14] and [BTK+17], respectively.
2. After designing methods to push content-based filtering and routing to the network layer, our next contribution involves increasing the expressiveness of these content filters, installed on TCAM, in the presence of hardware limitations w.r.t. number of bits available in each flow entry to represent a content filter (cf. Section 1.3.2). We explore various techniques to represent content filters expressively, given the bit length limitation, such that unnecessary network traffic can be reduced. We present techniques that i) use workload, in terms of events and subscriptions, to represent content, and ii) efficiently select attributes to reduce redundancy in content. Moreover, these techniques complement each other and can be combined together to further enhance performance w.r.t. bandwidth efficiency. This contribution is based on the work published in [BTGR16]. The author of this thesis contributed around 70% of the paper's scientific content.
3. To further enhance expressiveness of content-based filtering despite the discussed bit length limitation (cf. Section 1.3.2), we strike a balance between purely application-layer-based and purely network-layer-based publish/subscribe implementations by realizing a hybrid content-based middleware that enables filtering of events in both the application and the network layers. Moreover, we provide different selection algorithms with varying degrees of complexity to determine the events to be filtered at each layer such that unnecessary network traffic can be minimized while, also, considering latency/delay requirements of the middleware. Our hybrid middleware offers full flexibility to configure it according to the performance requirements of the system. This contribution is based on the work published in [BTHR16]. The author of this thesis contributed around 70% of the paper's scientific content.
4. Our next contribution addresses yet another hardware limitation—limited number of flow table entries available to pub/sub traffic in TCAM (cf. Section 1.3.2). We design a filter aggregation algorithm that merges content filters on individual

switches to respect TCAM constraints while, also, attempting to reduce the adverse effect of aggregation on the expressiveness of filters. Our designed algorithm ensures minimal increase in unnecessary network traffic due to necessary aggregation. It uses the knowledge of advertisements, subscriptions, and a global view of the network state to perform bandwidth-efficient aggregation decisions on necessary switches. We provide different flavors of this algorithm with varying degrees of accuracy and overhead. This contribution is based on the work published in [BTBR17]. The author of this thesis contributed around 70% of the paper's scientific content.

5. A key challenge in a software-defined network is to ensure high responsiveness of the control plane to dynamically changing communication interactions (cf. Section 1.3.3). So, we propose a methodology for both vertical and horizontal scaling of a distributed control plane that is capable of improving responsiveness by enabling concurrent network updates in the presence of high dynamics while ensuring consistent changes to the data plane of PLEROMA. In contrast to existing scaling approaches that aim for a general-purpose distributed control plane, our approach uses knowledge of the application semantics that is already available in the design of the data plane of the pub/sub middleware, e.g. subscriptions and advertisements. By proposing a methodology for an application-aware control distribution, we show, in the context of PLEROMA, that application-awareness is significantly beneficial in avoiding the synchronization bottlenecks for ensuring consistency in the presence of concurrent network updates. As a result, responsiveness of the control plane is greatly improved. This contribution is based on the work published in [BTK+15]. The author of this thesis contributed around 70% of the paper's scientific content.

With regards to parts of the implementation and certain evaluations of the PLEROMA middleware, a number of student theses [Bal17, Heg16, Mis13, ES14, Gru14, Sri17] have also supported this work.

1.5 Structure of the Thesis

The rest of the thesis is structured as follows. Chapter 2 provides basic mechanisms required to realize a content-based pub/sub middleware on software-defined networks. Chapters 3 and 4 focus on improving bandwidth efficiency of the designed middleware. More specifically, Chapter 3 provides a series of techniques, which impact the mapping of content to flows or forwarding rules in switches, in order to improve the expressiveness of content filters despite hardware limitations. Chapter 4 presents a hybrid pub/sub solution, a means to further reduce bandwidth utilization, that allows hardware as well as software filtering. In Chapter 5, methods to deal with limited number of flow entries in TCAM in a bandwidth-efficient manner are provided. Chapter 6

presents methods to handle the control plane overhead in the presence of dynamics to ensure a scalable pub/sub solution. Finally, in Chapter 7, we conclude with a summary of our contribution and an outlook on future work.

Chapter 2

In-network Content-based Filtering

As discussed in Chapter 1, over the past few decades, content-based publish/subscribe has been primarily implemented as an overlay network of software brokers. Even though such systems provide the possibility of expressive filtering in software, they cannot match up to the performance (e.g., end-to-end latency of events) of communication protocols implemented on the network layer. As a result, to exploit network layer performance benefits, the recent advent of new networking technologies, such as SDN, have raised some research efforts towards realizing a publish/subscribe middleware that can support event filtering and routing within the network. For example, Zhang et al. [ZJ13] and Koldehofe et al. [KDTR12] propose the prospect of using SDN in future designs of publish/subscribe middleware. However, a design and implementation of content-based pub/sub on software-defined networks was yet to be explored in literature.

In this chapter, we exploit the capabilities of SDN to offer functionalities of the content-based pub/sub paradigm in the network layer. As mentioned earlier, standards like OpenFlow specify the interface to directly install and modify flows on switches and install dedicated communication paths between the hosts connected to the network. More specifically, the communication paths are established by installing content filters on the TCAM of switches. Therefore, in a software-defined network, it is possible to install a network topology, consisting of content filters, which yields line-rate performance in forwarding events between publishers and subscribers. However, the topology needs, also, to be dynamically updated with ongoing subscriptions and advertisements. In software-defined networks, this is the task of the controller which is in charge of installing/removing/modifying flows in switches and can access the switches via a dedicated control network. The design of the control algorithm is therefore crucial to the performance of a publish/subscribe middleware in the presence of dynamically changing subscriptions and advertisements. So, in this chapter, we provide the design, implementation, and a detailed performance evaluation of an SDN-based publish/subscribe

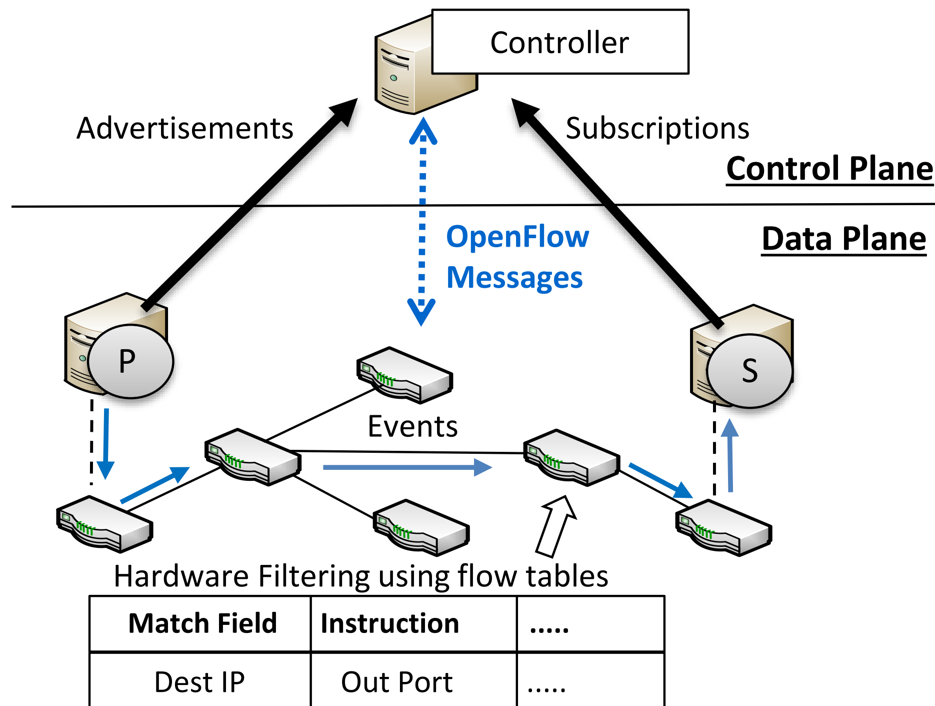


Figure 2.1: PLEROMA Middleware

system, called PLEROMA, and methods for its efficient reconfiguration.

2.1 The PLEROMA Middleware

A content-based publish/subscribe middleware using SDN consists of mainly two types of participants—publishers and subscribers—, which are connected to switches in a software-defined network. Publishers specify the information they intend to publish by sending advertisements to the control plane. Likewise, subscribers specify information they are interested in receiving by sending subscriptions to the control plane. The controller collects all these control requests ((un)advertisement/(un)subscription) based on which it installs paths on the data plane between each publisher and all its interested subscribers. In doing so, it configures the network's data plane by proactively installing suitable flow table entries—representing content filters—on SDN-configurable switches by utilizing the widely accepted OpenFlow standard. Once these filters are installed, published events can directly be filtered against the flow table entries of hardware switches in the data plane. Figure 2.1 illustrates the architecture of the PLEROMA middleware, which establishes line-rate content-based routing. We specifically use IP-Multicast addresses in flow table entries to represent filters in PLEROMA. In this thesis, we use the term *flow* to represent a flow table entry on an SDN-configurable switch. A flow further defines an outgoing port of a switch (cf. Figure 2.1) to which an

event packet with a matching header field (packet-header-based filtering) is forwarded. Note, our content representation mechanisms are generic and other fields, e.g., MAC addresses or VLAN tags [Ope13], can also be used for the same purpose.

The above description of the PLEROMA middleware directly leads us to the two main challenges to be addressed in order to provide content-based pub/sub functionalities on the data plane. These are (i) mapping of content filters to flow entries in hardware switches, and (ii) design of an efficient dissemination structure for packet forwarding and its efficient reconfiguration. These challenges are addressed in the following two sections of this chapter where we, first, provide a mechanism to map content such that advertisements, subscriptions and events can be represented as match fields in flows of switches or header fields of event packets (cf. Section 2.2). This is followed by an algorithm (topology configuration) that details the processing of both advertisements and subscription requests at the control plane such that necessary paths are deployed between publishers and relevant subscribers by installing the aforementioned mapped content filters, represented by switch flows, along these paths (cf. Section 2.3).

2.2 Content Representation

To ensure high expressiveness and establish paths with low-bandwidth usage between publishers and subscribers, we follow the content-based subscription model, i.e., an event is composed of a set of attribute value pairs. To realize the aforementioned packet-header-based filtering of events at the data plane, we need an efficient mapping between content attributes and flow identifiers (i.e., one or more header fields that uniquely identify flow entries in the flow tables of switches). There are two steps to this mapping process.

2.2.1 Spatial Indexing

The first step yields a binary representation of content following the principle of spatial indexing [KDTR12]. The event space, denoted by Ω, i.e., the set of all possible events that can be disseminated by the publishers, can be represented by a multi-dimensional space of which each dimension refers to the values of a specific attribute. An event is simply represented as a point and a subscription or advertisement as a subspace in Ω. Building on the principle of spatial indexing, we can divide the event space into regular subspaces that serve as enclosing approximations for events, advertisements, and subscriptions. In fact, since events are points in Ω, they are represented by subspaces of finest possible granularity. Any subspace can be identified by a binary string named dz-expression (in short dz). In particular, dz-expressions fulfill the following characteristics:

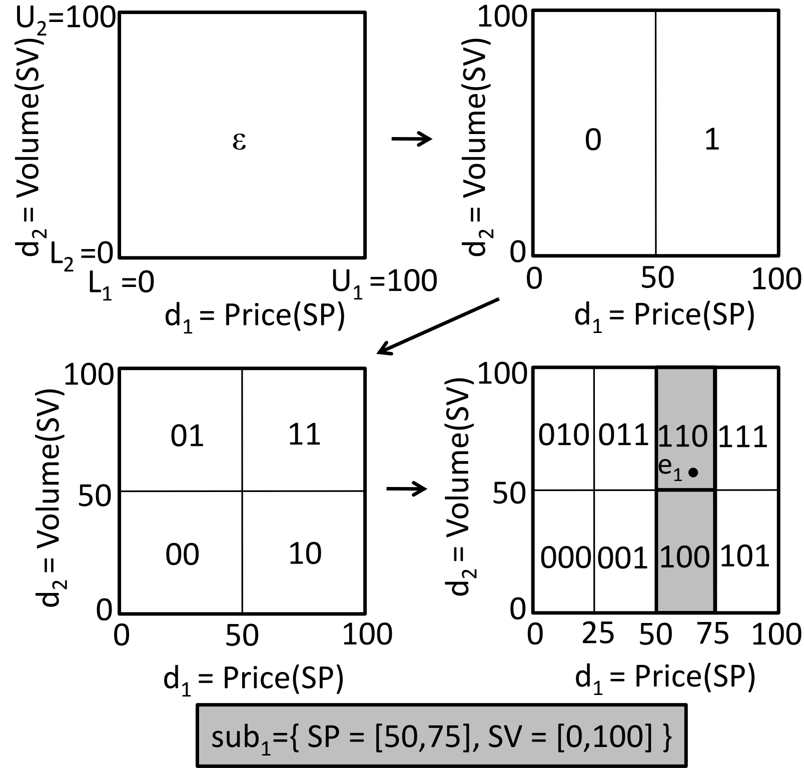


Figure 2.2: Spatial Indexing

- The shorter the dz, the larger is the corresponding subspace in Ω. Again, since events are points in Ω, they are represented by dzs of maximum length.
- A subspace represented by dz_i is *covered* by the subspace represented by dz_j iff dz_j is a prefix of dz_i. In this case, we write $dz_j \succ dz_i$.
- Two subspaces dz_i and dz_j are overlapping if either $dz_i \succ dz_j$ or $dz_j \succ dz_i$ holds and the overlap $dz_i \cap dz_j$ is identified by the longest of the two dzs.
- For overlapping non identical subspaces dz_i and dz_j, the non overlapping part, say $dz_i - dz_j$, may need to be identified by multiple subspaces. For instance, the non overlapped part of $dz_i = 0$ w.r.t. $dz_j = 000$ contains subspaces 001, 010, and 011.

We illustrate spatial indexing with an example in Figure 2.2 where we consider a stock quote dissemination system implemented by the pub/sub paradigm. In this example, we consider two attributes (or dimensions) stock price (SP) and stock volume (SV) of a stock quote dissemination system. An advertisement/subscription can be composed of several dzs, denoted as DZ. For instance, to approximate the subscription sub_1 in Figure 2.2, the event space is repeatedly divided and finally two dzs are required, i.e., $DZ = \{110, 100\}$. The containment and overlap relationships between a pair of DZ

can be defined with respect to a set of dz-expressions represented by them. For the sake of simplicity, here, we consider only two dimensions. However, multiple dimensions can be indexed which can even include string attributes such as company name in the stock quote example. The string attributes can be linearized by hashing and indexed in a similar manner [MJ14].

2.2.2 Mapping a dz

The second step involves the mapping of the generated binary strings (dzs) to flow identifiers. Using the above relations, an event e disseminated by a publisher P will contain in a chosen packet header field a dz that represents its attribute values. In order to deliver e to a subscriber S with a subscription sub which expresses an interest in e, the controller must have installed on each switch along the path (between P and S) a flow whose chosen match field matches the corresponding header field of this event. With respect to spatial indexing, an event will match a subscription filter if it lies within the subspace representing the subscription in Ω, i.e., the dz representing sub covers ($\succ$) the dz representing e. So, for a match to occur between e and sub, we utilize the characteristics of dzs such that the match field in flows representing the filters for sub covers the corresponding header field of event packet representing e. To this end, we use a range of IPv6 multicast addresses, reserved for pub/sub traffic, as the flow identifiers to which dzs are mapped. So, a subscription/advertisement is first converted to one or more dzs which are then represented by one or more corresponding IPv6 multicast addresses. These IPv6 multicast addresses are then used by the flow entries in the flow tables of switches for event matching and forwarding. The covering relation between subspaces is accommodated in IP addresses with the help of Classless Interdomain Routing (CIDR) style masking supported by hardware switches where the 'don't care' symbol (*) is used to represent masking operations. An event is also represented as an IPv6 multicast address and forms part of the header of the event packet. This enables header-based matching and subsequent forwarding of the event packet as dictated by a flow on account of a match. So, continuing the stock quote example from Figure 2.2, the dz representing the subspace $\{110\}$ is converted to the IP address `ff0e:c000:*` (`ff0e:c000::/19`). Now, if the event e_1=$\{$SP = 65, SV = 55$\}$ in the figure, which lies within (matches) sub_1, is represented by the dz $\{110010\}$, then it is converted to an IP address `ff0e:c800::` and header-based matching of this event packet takes place with the installed flows for sub_1.

2.3 Topology Reconfiguration

An efficient approach to topology reconfiguration is central to pub/sub using SDN. To this end, we need to maintain a dissemination structure which considers as constraints

latency efficiency, bandwidth usage, and cost efficiency to update the network topology. Clearly, the lowest latency is achieved if a controller establishes a shortest path for each publisher/subscriber pair. However, this severely limits the reuse in forwarding an event on common paths, i.e., the possibility to share common subpath(s) and, therefore, bandwidth between a publisher and subscribers with overlapping subscriptions. Moreover, each new subscription or advertisement would trigger updates of the network topology to add paths between all relevant publishers and subscribers and, therefore, impose a very high reconfiguration cost.

A common alternative—often taken by traditional broker-based systems [JCL+10]—is to embed the paths between publishers and subscribers by means of filters in a single spanning tree. The spanning tree reflects low latency paths between any pair of publisher and subscriber. Since all paths between publishers and subscribers are embedded in the same tree, the number of times an event needs to be forwarded is significantly reduced. The reconfiguration cost is also limited to the edges in the spanning tree and is significantly reduced wherever subscriptions and advertisements overlap.

As a result, the PLEROMA middleware maintains a spanning tree (comprising switches), denoted by T, at the controller, to account for an acyclic dissemination structure on which paths are embedded between publishers and subscribers by installing appropriate flows (filters) on switches along these paths. A path is nothing but a sequence of switches (denoted as R) on which flows are deployed to ensure connectivity between the publisher and the subscriber.

Installing paths between publishers and subscribers by the controller involves reading the existing flows of each switch (along the path), taking decisions on flow changes, and writing these changes to the switch. The network state is represented by network configuration that consists of (i) all switches constituting the network, (ii) all links connecting the switches in a spanning tree to account for an acyclic dissemination structure, and (iii) all pub/sub flows deployed on each switch. In general, the network configuration is maintained both at the data plane and the control plane of a software-defined network. In fact, we denote the network configuration at the data plane as *DP-config* which is maintained implicitly as a result of pub/sub flows deployed on the TCAM of hardware switches. On the other hand, the control plane network configuration is denoted as *CP-config* and is maintained by the controller and serves as a reflection of DP-config. The controller needs to maintain the network state CP-config so that it does not need to query the switches in the data plane and read their states for processing every control request. In fact, since the controller assumes CP-config to be identical to DP-config, it uses CP-config to read existing flows and decide on flow changes. On taking a decision, the controller sends the new flow changes to the hardware switch, resulting in a change in DP-config. After changing DP-config, the controller also performs these flow changes in the CP-config to ensure that it remains consistent with DP-config. The details of the aforementioned protocol of keeping CP-

config consistent with DP-config is provided in Chapter 6. In fact, in the remaining part of this thesis, we do not differentiate between CP-config and DP-config and consider them to be the same entity such that modifying one implies modifying the other.

Please note that, the dissemination structure of CP-config maintained at the control plane (and DP-config maintained implicitly at the data plane) represents the aforementioned spanning tree. As a result, a spanning tree maintained at the control plane, a CP-config, and a DP-config are synonymous in the rest of this thesis.

As mentioned earlier, installing a path between a publisher and a subscriber involves reading the existing flows of each switch along the path in question of the spanning tree, taking decisions on flow changes on each of these switches, and writing these changes to the affected switches in the spanning tree. In order to understand this decision-making process that determines flow changes on a switch, it is important to understand how the controller processes each type of control request, which is the subject of discussion in the remaining part of this section.

2.3.1 Maintenance of flow tables

The flow tables in the switch network are modified (e.g, by adding or removing flow entries) by the controller as a result of (un)advertisement and (un)subscription requests. In the following, we will first focus on advertisement, subscription requests and later briefly describe the handling of unsubscription, unadvertisement requests by the controller.

2.3.1.1 Advertisements and Subscriptions

On arrival of an advertisement, denoted by $DZ(P)$, from a publisher P, the controller notes each dz_i in $DZ(P)$ and adds P to the spanning tree (i.e., T). The controller then checks for already existing subscribers in T whose subscriptions overlap with $DZ(P)$. If there is no overlap, then no further actions are taken. However, if an overlap exists, then the controller establishes paths between the publisher P and all subscribers with overlapping subscriptions in T. A path consists of a sequence of physical switches (denoted as R) on which flows need to be established along with the out ports (denoted as oP) through which a matching event should be forwarded so that connectivity is achieved between the publisher P and the subscriber S, i.e., $\langle P, S, T\rangle = \{(R_i, oP_i), \ldots, (R_j, oP_j)\}$. Each path between a publisher P and a subscriber S only forwards the events matching the subspaces overlapped between $DZ(S)$ and $DZ(P)$ (cf. Algorithm 1, lines 1-6). In this way false positives (events forwarded to a subscriber that is not interested in receiving them) are avoided.

Subscription requests are handled similarly as described formally in lines 7-12 of Algorithm 1. On arrival of a subscription, as a first step, the controller calculates the path

Algorithm 1 Publish/subscribe maintenance at a single controller

```
1:  upon event Receive(ADV, P, DZ(P)) do
2:  for all dz_i ∈ DZ(P) do
3:     subSet = {S ∈ 𝕊 ∧ ∃dz_j ∈ DZ(S) : dz_i ≻ dz_j ∨ dz_j ≻ dz_i} // Subscribers with
       overlapping DZ(S)
4:     for all S ∈ subSet do
5:        overlapWithSub = dz_i ∩ DZ(S)
6:        flowAddition(overlapWithSub, ⟨P, S, T⟩, T)
7:  upon event Receive(SUB, S, DZ(S)) do
8:  for all dz_i ∈ DZ(S) do
9:     pubSet = {P ∈ ℙ ∧ ∃dz_j ∈ DZ(P) : dz_i ≻ dz_j ∨ dz_j ≻ dz_i} // Publishers with
       overlapping DZ(P)
10:    for all P ∈ pubSet do
11:       overlapWithPub = dz_i ∩ DZ(P)
12:       flowAddition(overlapWithPub, ⟨P, S, T⟩, T)
13: procedure flowAddition( dz, ⟨P, S, T⟩, T) do
14:  destIP = (binary(ff0e:b400)&(dz ≪ 112 − |dz|)) \ 16 + |dz|
15:  for all r_i ∈ ⟨P, S, T⟩ do
16:     Flow fl_n = MF ∪ IS ∪ PO
17:     fl_n.MF = destIP
18:     fl_n.PO = default value
19:     fl_n.IS.oP = {r_i.oP_i}
20:     curFlow = getCurrentFlowsFromSwitch(r_i.R_i)
21:     if curFlow ≠ ∅ ∧ ¬(∃fl_c ∈ curFlow : fl_c ≻ fl_n) then // Cases 3 - 4: None of the
        curFlow fully covers fl_n
22:        for all fl_c ∈ curFlow : fl_n ≻ fl_c do // Case 3
23:           deleteFlowFromSwitch(fl_c, r_i.R_i)
24:        for all fl_c ∈ curFlow : fl_c ≈≻ fl_n do // Case 4
25:           fl_n.IS.oP = fl_n.IS.oP ∪ fl_c.IS.oP
26:           increasePriority(fl_n.PO)
27:        for all fl_c ∈ curFlow : fl_n ≈≻ fl_c do // Case 5
28:           fl_c.IS.oP = fl_c.IS.oP ∪ fl_n.IS.oP
29:           increasePriority(fl_c.PO)
30:           modifyFlowOnSwitch(fl_c, r_i.R_i)
31:     addFlowOnSwitch(fl_n, r_i.R_i)
```

between the subscriber S and each relevant publisher P on the tree T. Once the path is calculated, the controller establishes the path by inserting (or modifying) flows on the switches along the path between the publisher P and the subscriber S. The flows ensure that only the events matching the overlapped subspaces (i.e., $DZ(S) \cap DZ(P)$) are forwarded on the path. The process of establishing paths along the switch network is discussed in detail later in this section.

2.3.1.2 Flow installation

The installation of flows on the switches requires to specify the *match field (MF)*, *instruction set (IS)*, and *priority order (PO)* of a flow [Ope13]. The match field defines the header information against which packets are matched. Recall that PLEROMA uses,

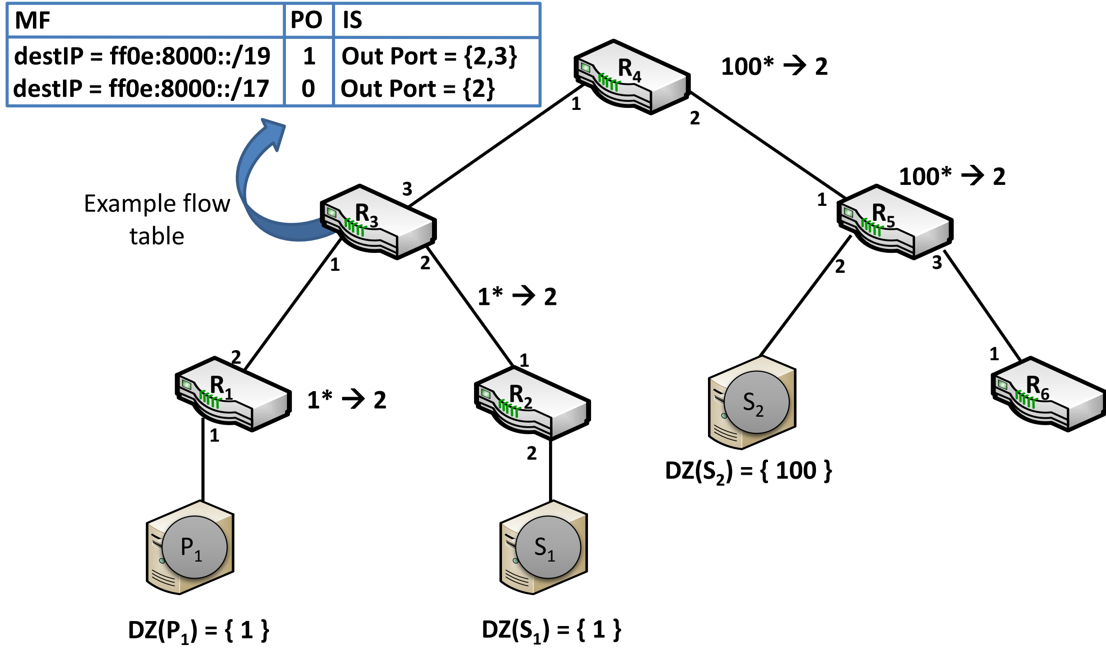


Figure 2.3: Forwarding in the switch network. Match fields of flows in R_1, R_2, R_4-R_6 are shown as dzs. Flows follow the notation MF → IS : PO

for interoperability with other services, IP-multicast ranges to embed dz-expressions. For instance, subspaces with $dz = 101101$ and $dz = 101$ are converted to IPv6 multicast addresses `ff0e:b400:*` and `ff0e:a000:*`, respectively. Therefore, an event $dz = 101101$ can be matched against a flow with $dz = 101$ by a hardware switch during forwarding, i.e., `ff0e:a000::/19` $\succ$ `ff0e:b400::`.

Furthermore, in the instruction set the outgoing ports are specified, ensuring that a matching packet (i.e., an event) can be forwarded to multiple destinations in the spanning tree. Also, the priority order needs to be defined to decide on the order in which flows will be applied to a packet. A higher priority ensures that if a packet has multiple matches in the flow table, it would be matched against and follow the IS of the flow with highest priority. For example, in Figure 2.3, an incoming event ($dz = 1001$) on switch R_3 matches multiple flows with $dz = 1$ and $dz = 100$. However, the switch only follows the instructions of the first match. Therefore, to ensure proper forwarding, the flow installation gives higher priority to the flows with longer dz. In Figure 2.3, priority order on R_3 ensures that all packets matching flow with $dz = 100$ are forwarded to both switches (R_2 and R_4). However, packets matching the flow having $dz = 1$ but not with the flow having $dz = 100$ are only forwarded to R_2.

To describe the maintenance of flows in the presence of dynamic (un)subscriptions, we first define the containment relation between flows w.r.t. a single switch. A flow fl_1 covers (or contains) another flow fl_2, denoted by $fl_1 \succ fl_2$, iff the following two condi-

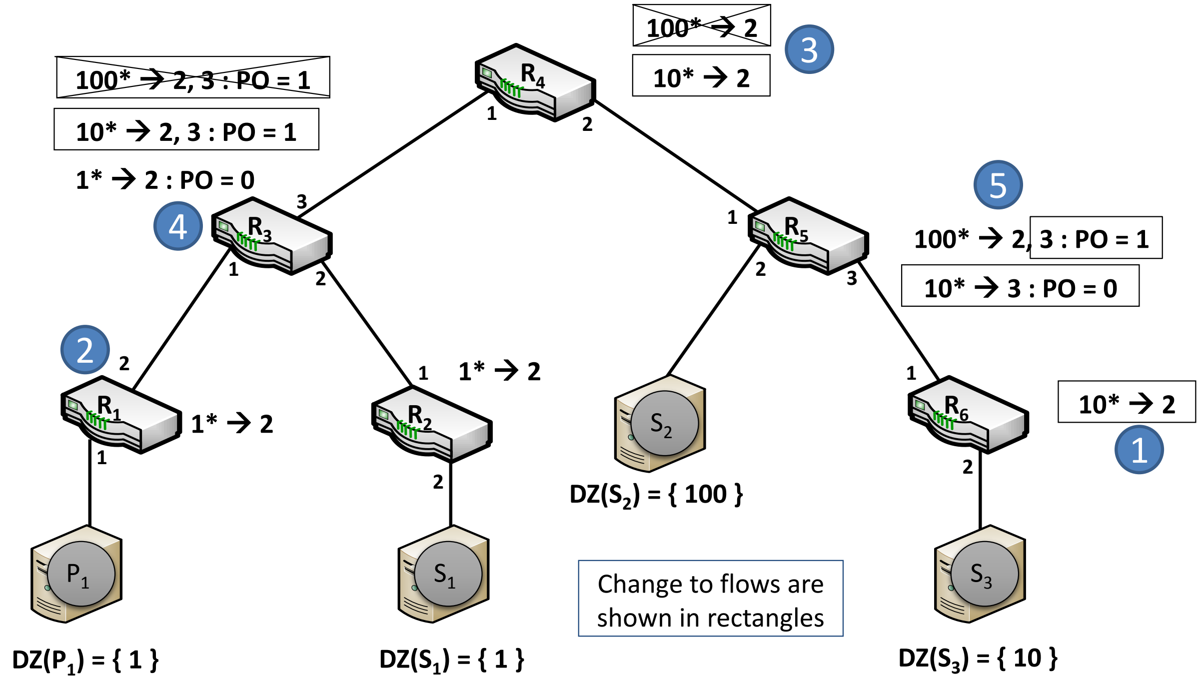


Figure 2.4: Flow maintenance on the arrival of S_3.

tions hold: (i) the dz associated with the IP address in the match field of fl_2 is covered by the dz of fl_1, and (ii) the out ports to which a packet matching fl_2 is forwarded are a subset of those specified in the IS of fl_1. Likewise, a partial containment relation ($\gtrapprox$) can be defined between flows of a switch (or flows to be installed on a switch). A flow fl_1 partially covers (or contains) another flow fl_2, denoted by $fl_1 \gtrapprox fl_2$, if dz associated with the match field of fl_1 covers dz of fl_2, but not all the out ports used for forwarding packets matching fl_2 are listed in the IS of fl_1.

The procedure *flowAddition* is used by the controller to set up flows on the switches along the path $\langle P, S, T\rangle$ between the publisher P and the subscriber S (cf. Algorithm 1, lines 13 - 31). The dz used for creating the match field of the new flows (to be added in the switch network) is determined from the overlap between $DZ(S)$ and $DZ(P)$, as mentioned earlier.

In more detail, the controller iteratively checks the existing entries in the flow tables of each switch R_i along the path $\langle P, S, T\rangle$ and determines whether to add a new flow fl_n or to modify (or delete) existing flows. The following cases drive the process of flow addition and modification at a particular switch R_i. Continuing the example from Figure 2.3, the cases are explained w.r.t. the changes to the flow tables of the switches on the arrival of a new subscriber S_3 with subscription $DZ(S_3) = \{10\}$ as depicted in Figure 2.4. (1) If the flows are not currently installed on a switch, then the new flow fl_n is simply added to the flow table of that switch, e.g., a new flow with $dz = 10$ is

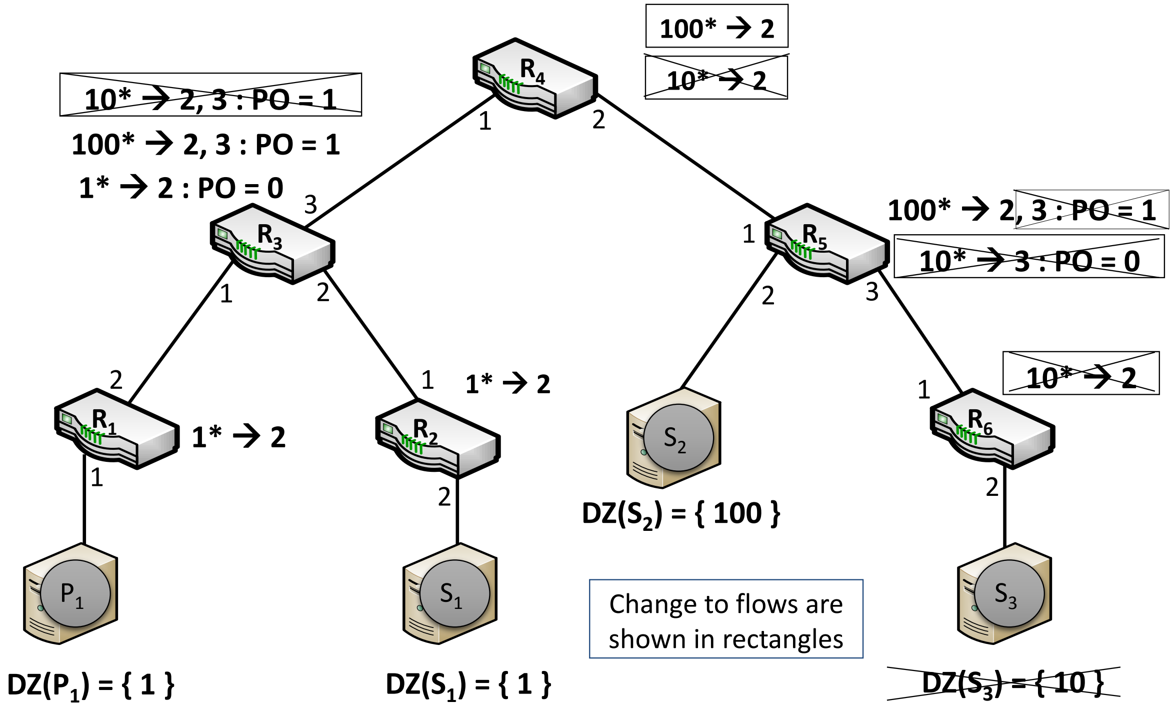


Figure 2.5: Flow maintenance on the departure of S_3

added to R_6 in Figure 2.4. (2) If an existing flow fl_c already covers the new flow fl_n to be installed on the switch (i.e., $fl_c \succ fl_n$), then no action is performed, e.g., no new flow is added to the switch R_1 in Figure 2.4 when S_3 subscribes. The flow $\{10*\}$ that needed to be installed on R_1 to direct required traffic towards S_3 is covered by the already existing flow $\{1*\}$ which directs traffic that includes required traffic for S_3 along the same direction. So, an additional flow in this case will be redundant. (3) If an existing flow fl_c is covered by the new flow fl_n, then the new flow fl_n is added and fl_c is deleted from the flow table as it is no longer needed, e.g., in Figure 2.4 existing flows associated with $dz = 100$ are replaced by new flows with $dz = 10$ on R_3 and R_4. This follows from the argument of case (2). So, the existing flow which is covered by the new flow should be replaced to avoid redundancy. (4) If the new flow fl_n is partially covered by an existing flow fl_c (i.e., $fl_c \gtrsim fl_n$), then fl_n should be added with high priority and should include the out ports in the IS of fl_c, as depicted by R_3 in Figure 2.4. This ensures that traffic specific to the flow $\{10*\}$ (subscription of S_3) strictly matches it and gets forwarded towards both S_3 and S_1. The remaining traffic that is specific only to S_1 and that does not match the new flow will now be forwarded by the existing flow $\{1*\}$ only to S_1. (5) Finally, if the existing flow fl_c is partially covered by the new flow fl_n, then besides adding fl_n to the flow table, the existing flow fl_c should be updated to include out ports used by fl_n and to hold higher priority than fl_n, e.g., in Figure 2.4 an additional out port (i.e., $oP = 3$), and a higher priority

order is assigned to an existing flow $\{100*\}$ on R_5. This follows similar logic as case (4).

2.3.1.3 Unsubscriptions and Unadvertisements

We, also, briefly discuss the handling of unsubscriptions and unadvertisements by the controller. Handling of an unsubscription or unadvertisement is the exact reverse process of handling a subscription or advertisement. On the arrival of an unsubscription, the subscriber S, associated with the corresponding subscription, is removed from T. This is accomplished by removing previously established paths between S and all publishers with overlapping advertisements. To remove a path on T, the flows are either deleted or downgraded depending upon other subscribers reachable (w.r.t. their relevant publishers) via a particular switch. For example, on arrival of an unsubscription from S_3 in Figure 2.5, the path between P_1 and S_3 comprising of switches R_1, R_3, R_4, R_5 and R_6 needs to be removed. However, the existing flows on these switches determining this path cannot simply be removed as each of these flows may share paths to other subscribers based on the covering relations between flows as seen earlier in this section. For example, the flow with $dz = 10$ is deleted from the flow table of R_6 as no other subscriber is reachable w.r.t. P_1 via R_6. However, the flows installed on switches R_3, R_4, and R_5 have to be downgraded from $dz = 10$ to $dz = 100$ (in their match fields) because the path from P_1 to subscriber S_2 with $DZ(S_2) = \{100\}$ passes through these switches. Downgrading not only ensures that no further events are forwarded to S_3 but also ensures that no other subscriber paths get affected due to these updates. So, in this example, S_2 continues to receive relevant events as downgrading of flows does not affect its path from P_1. Likewise, an unadvertisement from a publisher P is handled by removing the previously established paths in the switch network between the publisher P and all subscribers with overlapping subscriptions on T with which the publisher P is associated.

2.4 Performance Evaluations

This section is dedicated to an analysis of the design and implementation of the proposed PLEROMA middleware. A series of experiments are conducted to understand the effects of the design on the performance of the system w.r.t. end-to-end latency for event dissemination, bandwidth efficiency in terms of false positives w.r.t. length of dz, and control overhead.

2.4.1 Experimental Setup

We have conducted our evaluations under three environments—1) an SDN-testbed comprising a physically distributed network of software switches (*SDN-t-sswitch*) and

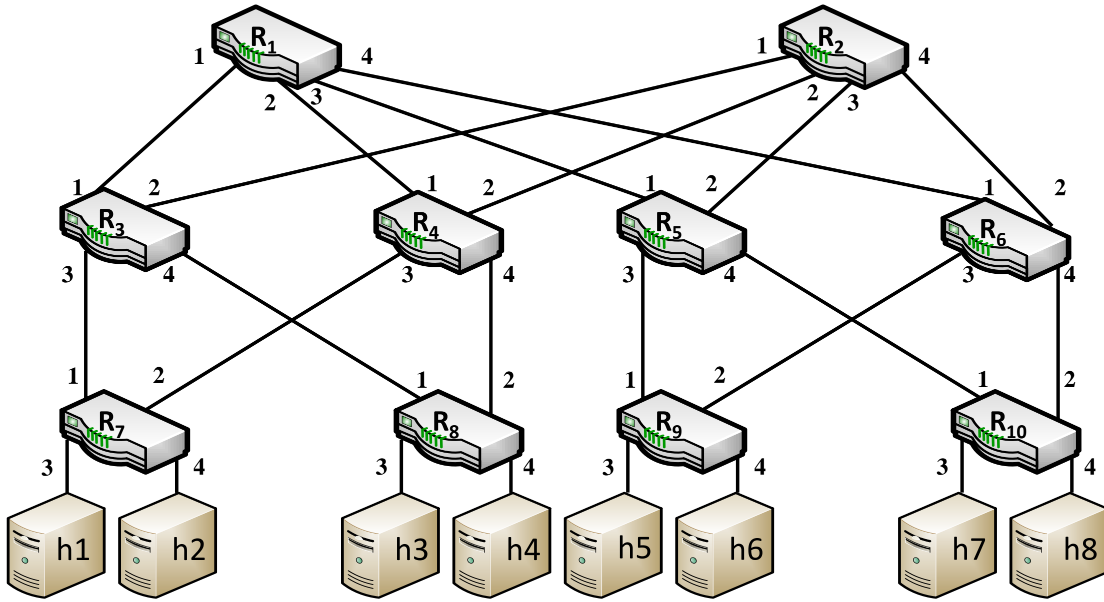


Figure 2.6: Testbed Topology

commodity PC hardware, 2) an SDN-testbed (*SDN-t-hswitch*) comprising a hardware Whitebox Openflow-enabled switch from Edge-Core and commodity PC hardware, and 3) an emulated network running on a single machine using Mininet (*SDN-m*). Majority of the experiments in this thesis have been conducted on these aforementioned test environments.

In more details, *SDN-t-sswitch* consists of commodity PC hardware and virtualization technologies as used in data centers. *SDN-t-sswitch* is created as a hierarchical fat-tree topology as depicted in Figure 2.6. The testbed consists of a cluster of hosts (running on commodity rack PCs) constituting 10 switches and 8 end systems. Some of these hosts act as OpenFlow switches with four physical ports by executing a production-grade software switch (Open vSwitch [Ope]) attached to the 4-port NIC. The other hosts act as 8 end systems (end hosts) by executing virtual machines on two physical machines. The end hosts implement the functionality to publish and subscribe events.

Besides *SDN-t-sswitch*, we have also conducted experiments on *SDN-t-hswitch* where, again, we created a hierarchical fat-tree topology consisting of 10 switches and 8 end-hosts as depicted in Figure 2.6. The 10 switches are created by partitioning the hardware Whitebox switch from Edge-Core running the network operating system PicOS (version 2.6) [Pic, Edg]. The 8 end-hosts are hosted on commodity rack PCs and perform the role of publishers and subscribers. We also implemented an application-layer based middleware to compare the performance of hardware filtering in PLEROMA with software filtering. The SDN controller and application layer reside on a 3.10 GHz machine with 40 cores.

Please note that all end-hosts are synchronized using the IEEE 1588 Precision Time Protocol (PTP). We used a separate network infrastructure for PTP traffic using a second NIC on each host dedicated to PTP synchronization to counter the possibility of inaccuracies in clock-synchronization.

Besides the aforementioned testbeds, we have also conducted experiments on a prominent tool for emulating software-defined networks, namely, Mininet [LHM10] (*SDN-m*). Mininet is an extremely flexible tool that allows to conduct experiments with different types of topology and application traffic.

For the evaluations presented in this chapter, in order to generate workload, i.e., events and subscriptions, we use both synthetic as well as real world data. With regards to synthetic data, the workload was generated using parameters similar to those used in well established publish/subscribe literature [CMTV07, MJ14, ZKV13]. So, we used a content-based schema containing up to 10 attributes [MJ14], where the domain of each attribute varies in the range $[0, 1023]$. Most real world applications, e.g., stock quote dissemination systems, perform content-based routing with not more than 10 attributes and similar domain ranges. Experiments are performed on two predominantly used models for the distributions of subscriptions and events [MJ14, CMTV07]. The uniform model generates subscriptions and events independent of each other and uniformly distributed in Ω. The interest popularity model chooses up to 8 hotspot regions around which subscriptions and events are generated using the widely used zipfian distribution.

2.4.2 End-to-End Latency

These experiments study the latency characteristics of the aforementioned *SDN-t-sswitch* and *SDN-t-hswitch* (with fat-tree topology). We analyse the end-to-end latency to deliver an event from a publisher to all interested subscribers w.r.t. the number of subscriptions in the system. For the experiment, up to 16,000 subscriptions are generated using the above mentioned distributions and divided among different end hosts.

Figure 2.7(a) compares the performance of PLEROMA implemented on *SDN-t-sswitch* with a purely application layer-based middleware (*APP-M*). We implemented the purely application layer-based middleware as a parallelized matching pub/sub service. We divided the event-space into 16 partitions and assigned them to 16 matchers running on 16 cores to enable one-hop forwarding of events similar to Bluedove [LYK+11]. Note that the highly parallelized filtering technique implemented at *APP-M* presents one of the best case scenarios for application layer filtering, resulting in relatively better latency performances when compared to state-of-the-art solutions using overlays. However, even then, Figure 2.7(a) shows that PLEROMA, with an average end-to-end latency in the order of microseconds, clearly, outperforms the application layer-based middleware *APP-M* even when virtual switches are used in the data plane. Moreover, the number of subscriptions does not significantly impact end-to-end latency in *SDN-t-sswitch*. As the number of subscriptions increase, so does the number of flows on the

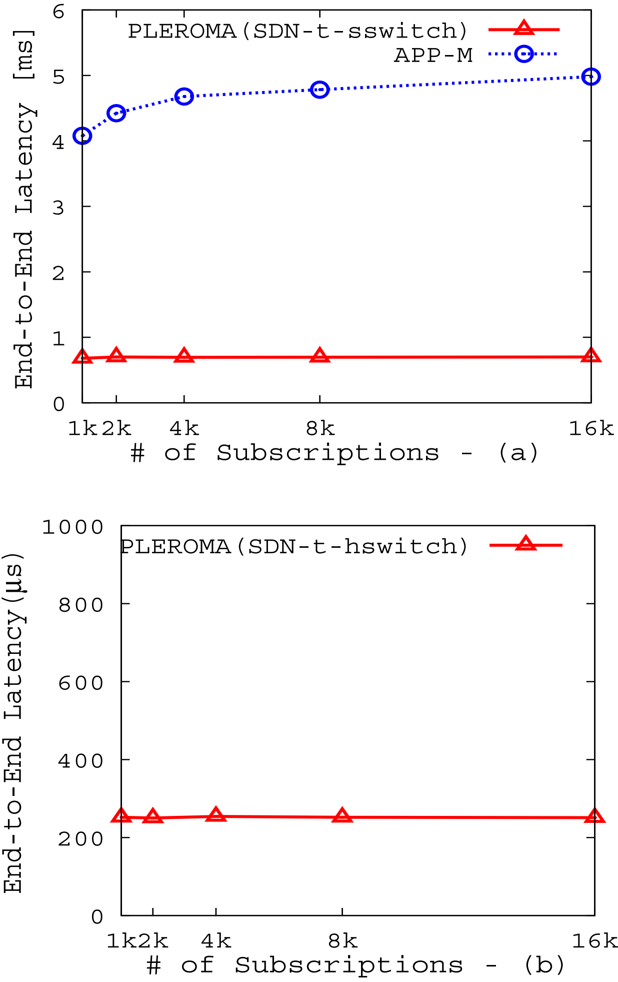


Figure 2.7: End-to-End Latency

switches of the network. So, clearly, the number of flows on switches does not impact the filtering time of the events which remains constant. On the contrary, as filtering is performed in the application layer in *APP-M*, more the number of subscriptions, more is the average end-to-end latency.

The use of virtual switches in *SDN-t-sswitch* gives conservative performance bounds. As a result, we, also, perform experiments on *SDN-t-hswitch* consisting of real hardware switches. Figure 2.7(b) shows that the use of real hardware switches further reduces the average end-to-end latency of events as compared to virtual switches. Also, as expected, in *SDN-t-hswitch* too, the latency remains constant with increasing number of subscriptions. So, the above evaluation results, clearly, show the improvement in performance w.r.t. end-to-end latency of events when content-based routing is implemented on the network layer.

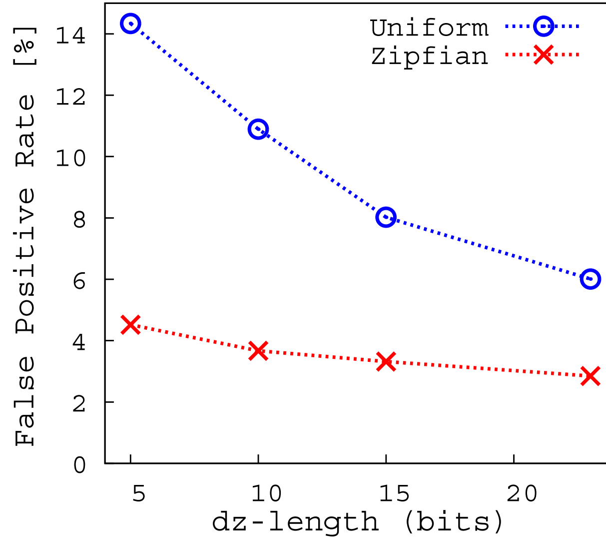


Figure 2.8: False Positive Rate

2.4.3 False Positive Rate

We define the false positive rate (FPR) as the percentage of total number of events forwarded to the subscribers that are unnecessary (i.e., false positives). Clearly, false positives are undesirable and the aim of any publish/subscribe system is to keep them to a minimum. We observe that the longer the *dz*, the fewer are the false positives. This follows from the fact that as the length of the *dz* increases, the granularity of the subspaces (assigned to advertisements, subscriptions and events) also increases and, hence, the false positives forwarded to a subscriber decrease. Figure 2.8 shows the variation of false positive rate with the length of *dz* for different number of subscriptions for both uniform as well as zipfian distribution. As seen in the figure, with increase in the length of the *dz* the false positives decrease for both distributions. As we only have a limited number of bits, say L_{dz}, for the representation of *dz* in an IP multicast address, subscriptions and events which differ in *dz* only after the L_{dz} cannot be differentiated. Thus, an event *e* might fit into the filtering criteria of a subspace—which does not actually contain (or cover) the event *e*—due to *dz* truncation and is counted as a false positive.

2.4.4 Control Overhead

The controller needs to process control requests from all publishers and subscribers in the system in order to ensure necessary forwarding of events in the data plane. In this context, we evaluate the impact of the rate at which control requests arrive at the controller on the average time it takes for a control request to be processed by the

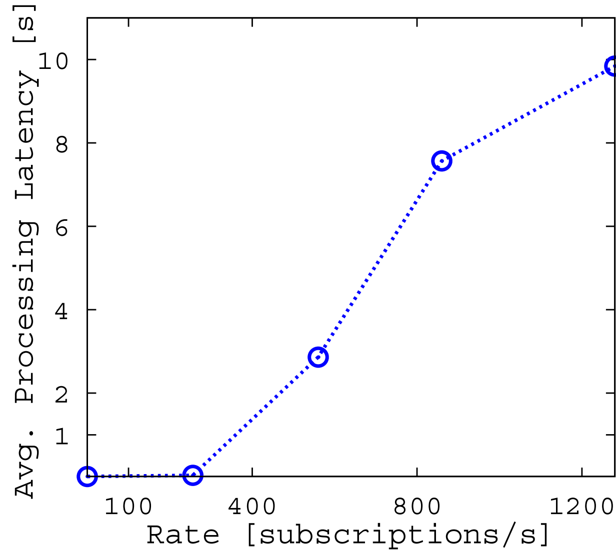


Figure 2.9: Average Processing Latency

controller (i.e., processing latency). More specifically, we define processing latency as the time elapsed from the issuance of the request by a publisher/subscriber to the time when this request has been completely processed by the controller. Please note that in order to ensure consistent processing of control requests (i.e., to ensure consistency of flow tables in the switch network), a controller processes one control request at a time, i.e., it performs sequential processing of requests. As a result, control requests from publishers and subscribers are enqueued to a waiting queue at the controller until their turn to be processed by the controller arrives. Our evaluation results depicted in Figure 2.9 show that, at relatively lower subscription rates, the processing latency is very low (in the order of milliseconds). However, as depicted in this graph, higher the rate at which subscriptions arrive at the controller, higher is the average processing latency of a subscription. This is because the waiting time of subscriptions goes up with increased subscription rates as a single controller processes requests sequentially with constant throughput.

2.4.5 Discussion

Our performance evaluations clearly show the impact of a network layer implementation of the pub/sub middleware on end-to-end latency of events in the system. We show that when compared to even a highly efficient pub/sub middleware (APP-M) implemented in the application layer, the end-to-end latency in PLEROMA is significantly lower. Moreover, with increasing number of subscriptions in the system, the end-to-end latency in APP-M further increases, whereas the number of subscriptions has no impact on end-to-end latency in PLEROMA. Our evaluations, also, show the

impact of the number of bits available for content representation, i.e., dz length, on the false positive rate in the system. Of course, longer the dz, lower are the false positives in the system. However, as the number of available bits is restricted by the selected match field (e.g. IPv6) length, additional mechanisms may need to be employed to increase expressiveness of content filters installed in TCAM. Our final set of evaluations show that on a single controller, processing control requests sequentially, an increase in the subscription rate in the system results in an increase in the average processing time of these subscriptions due to an overloaded controller. This, clearly, prompts the need for the design of a more efficient control plane with provision of concurrent yet consistent processing of control requests.

2.5 Related Work

Various approaches to the many aspects of content-based pub/sub have been presented in literature [CRW01, JCL+10, Müh02, CS04, JJE10, CFMP04, TKR13, BBQ+07]. Most of the traditional systems target the scalability aspect of pub/sub by attempting to reduce unnecessary dissemination of events in the system [CRW01, JCL+10, Müh02, TKK+11]. For example, one of the pioneers in this field, the Scalable Internet Event Notification Architecture (SIENA) [CRW01] pub/sub system, uses subscription summaries to filter out events from disseminating to the parts of the broker network that do not host interested subscribers. Similarly, forwarding of new subscriptions is only restricted to the brokers which previously do not receive subsuming (or covering) subscription summaries. In the recent past, clustering of subscribers has also been explored to realize a scalable pub/sub solution [CS04, PC05, WQA+04, BFP10]. For example, Kyra [CS04] partitions the publish/subscribe broker network into smaller routing networks. Subscriptions are assigned to relevant brokers such that event matching and subscription maintenance overhead is balanced between the networks.

While all of the aforementioned systems have their respective advantages in building a scalable pub/sub solution, a common drawback of these existing systems is their dependence on the application layer mechanisms to optimize pub/sub operations. For instance, event routing on a broker network that is organized oblivious to the underlying physical network (in short underlay), may result in higher bandwidth utilization (irrespective of the use of subscription summarization and event filtering mechanisms) and higher end-to-end latency, since multiple logical links in the broker network may share the same physical links [TKR13].

Only a few systems explicitly take into account the properties of the underlying network and its topology to organize publish/subscribe broker network [JPMH07, MSRS09, TKR13, ECG09]. In fact, Tariq et al. [TKR13], besides implementing a routing layer, build a topology discovery layer as its basis that attempts to reduce end-to-end latency, reduce packet duplicates, and minimize stress on underlying physical links. Nevertheless, inferring underlay topology or properties comes at a significant cost. Also, despite

the additional cost, it is still hard to accurately infer advanced underlay properties such as the current link utilization based on observations on end systems (such as brokers).

In the past, IP multicast has been proposed to distribute events between the clusters (or groups) of subscribers and publishers. Clearly, IP multicast overcomes many drawbacks of application layer by routing events on the network layer [RLW+02a, TKKR12]. However, IP multicast is very expensive in the presence of frequently changing subscriptions and event traffic, mainly because clusters have to be recalculated to ensure minimal false positives.

With the growing interest in technologies such as NetFPGA and SDN, some research efforts are being dedicated towards realizing pub/sub middleware that can support event filtering and routing within the network. LIPSIN [JZER+09] presents a novel multicast forwarding fabric using NetFPGAs on the network layer. More specifically, LIPSIN uses Bloom filters in data packets to encode links of the delivery tree for each event, resulting in the efficient multicasting of events on the network layer. However, the expressiveness of LIPSIN is limited to topic-based pub/sub. Zhang et al. [ZJ13] address impact of SDN on the future design of pub/sub middleware. Also, Koldehofe et al. [KDTR12, KDT13] present reference architecture of a content-based publish/subscribe using OpenFlow specifications. Nevertheless, to the best of our knowledge, we are the first to design, implement and thoroughly evaluate the performance of a content-based pub/sub middleware on software-defined networks.

2.6 Conclusion

In this chapter, we have proposed the PLEROMA middleware leveraging line-rate performance for content-based publish/subscribe in software-defined networks. In particular, we have proposed methods that preserve the basic functionalities of the pub/sub paradigm in the presence of dynamic subscriptions and publications. Our evaluations show that PLEROMA imposes very low latency in mediating events between publishers and subscribers (cf. Figure 2.7). However, the presented system does not address the problems related to hardware limitations in the data plane in terms of limited number of available bits for each filter representation and limited number of flow table entries available to pub/sub traffic in TCAM. In fact, the evaluation results presented in this chapter also indicate that fewer the number of bits available for content filter representation, more are the false positives in the system (cf. Figure 2.8). As, in this context, only a limited number of bits is available for content filter representation, additional methods must be employed to increase expressiveness of filters despite this limitation. This is addressed in Chapters 3 and 4 along with addressing the problem of limitation on number of flow entries in Chapter 5. Also, the problems of an overloaded controller, as depicted in Figure 2.9, must be tackled and new mechanisms need to be introduced at the control plane in order to handle high dynamic workload. These

mechanisms addressing the problems at the control plane of PLEROMA are addressed in Chapter 6.

Chapter 3

Expressive Mapping of Content Filters

As seen in Chapter 2, content-based pub/sub using SDN suffers from certain inherent limitations that result in bandwidth wastage. It should be noted that the effectiveness of content-based routing relies heavily on the expressiveness of content filters which are responsible for filtering out unnecessary traffic to ensure bandwidth-efficient communication. In an SDN-based pub/sub, these content filters are represented by the match fields of flows in the Ternary Content Addressable Memory (TCAM) of switches. This implies that content filters are limited by the bits available for filter representation at the selected match field (e.g., IPv6 address, VLAN tag). For instance, the choice of the destination IPv6 address to represent content filters allows a maximum of 128 bits which in reality would further reduce as the entire range of IP addresses may be shared among multiple applications. Moreover, IPv6 is not widely deployed and the use of IPv4 addresses instead can further impede the expressiveness of filters. Jokela et al., in LIPSIN [JZER+09], also target filtering on hardware (NetFPGA) in the context of topic-based pub/sub by encoding forwarding paths in packet headers. However, for a considerably small topology, even the use of a staggering 248 bits in the packet header does not suffice to prevent unnecessary traffic in the system (∼10%).

The above limitations may significantly impact bandwidth usage—something that is truly critical in a network. As a result, this chapter focuses on exploring techniques that address concerns with bandwidth efficiency in the context of limited number of bits available for the representation of each content filter in the flow entries of TCAM. First, we propose two techniques—selective indexing and adaptive spatial indexing—that consider workload in the system in terms of events and subscriptions to expressively map content to match fields of flows on hardware switches. Then, we present algorithms with varying complexities to efficiently identify and neglect redundant attributes or dimensions in the event space such that more bits are available to express more meaningful attributes in content filters. Moreover, these techniques complement each other and may be combined for enhanced effectiveness. Our evaluations show

Figure 3.1: Limitations of Content Representation

that a significant amount of irrelevant traffic (up to 97%) can be avoided by employing each of these techniques independently or in combination while benefiting from the advantages of SDN in terms of reduced end-to-end latency, high throughput, etc. The techniques proposed in this chapter result in significant updates to the network, i.e., the data plane, which must be performed in a consistent manner to ensure the desired behavior of the system. So, the contributions of this chapter also include the design of a light-weight approach that ensures data plane consistency in the presence of dynamic network updates when each of the proposed techniques is employed.

3.1 Limitations of Content Representation

In Chapter 2, we see that expressiveness or granularity at which spatial indexing can be performed is limited by the number of bits that can be appended to the destination

IP address. To further understand the true nature of this problem faced by content filters in PLEROMA, let us look at an example depicted in Figure 3.1. Let us assume that subscriber S has a subscription sub_1 : {T = [50, 100] $\wedge$ P = [50, 100]}. Spatial indexing yields the dz {11} to represent it as illustrated in the 2-bit representation in the figure. This dz is then converted into an IPv6 address (ff0e:c000:*) and installed as a destination IP in the match field of flows on the switches, enabling hardware filtering of events along the path between publisher P and subscriber S. Now, let us assume that instead of 2 bits only 1 bit can be accommodated in the IP address reserved for pub/sub traffic. In such a scenario, subscription sub_1 will be represented by the dz {1} as depicted in the 1-bit representation in Figure 3.1. This implies that all events matching the entire subspace of {1} in the figure will be received by subscriber S. So, the path between P and S will be subjected to a lot of unnecessary traffic or *false positives*. Note, the length of dzs, required to accurately represent content, increases with the increase in the number of dimensions in the system as spatial indexing is employed along every dimension.

As a result, the remaining part of this chapter is dedicated to the design of various techniques that would improve expressiveness of content filters installed on hardware switches, despite their limitations, and render content-based pub/sub realized on software-defined networks bandwidth-efficient. The presented techniques are workload dependent and are implemented by the controller. The controller already has a knowledge of all the subscriptions in the system and has to additionally collect statistics of events periodically and modify flows on switches accordingly.

Note that, although we focus on spatial indexing, other indexing techniques (e.g., Bloom filters, hashes) will encounter the same problems and the proposed techniques in this chapter are applicable in general to all indexing mechanisms.

3.2 Workload-based Indexing

The effectiveness of the previous attempts to encode content into binary form has primarily depended on the size of the event space. Be it the use of spatial indexing, or hashes, the only parameters that play a role in the mapping process are the number of available bits and size of Ω. However, in this chapter, we design two mapping techniques—selective indexing and adaptive spatial indexing—that not only consider the previous two parameters but also look into the workload of the system (i.e., events and subscriptions) to encode content to binary strings.

3.2.1 Selective Indexing

In-network filtering may result in significant number of false positives depending on the size of Ω, i.e., number of dimensions and range of values along each dimension.

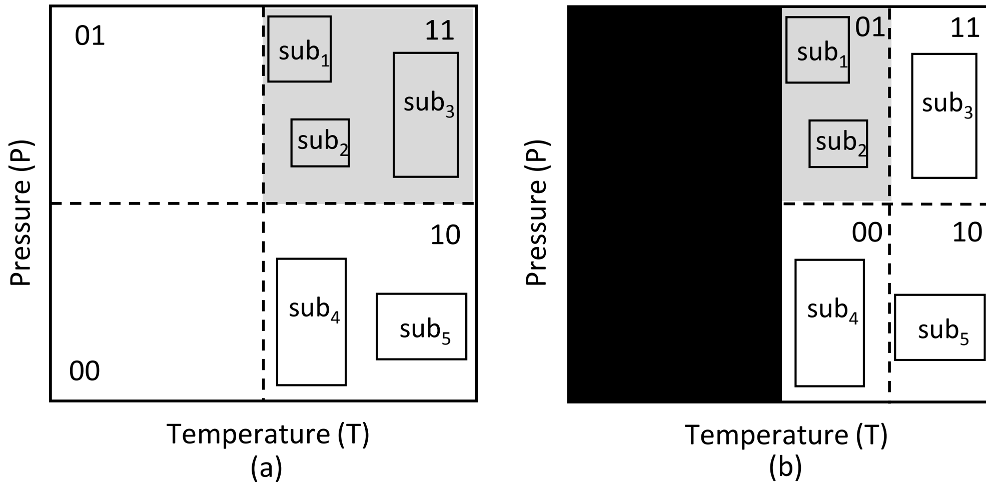


Figure 3.2: Avoiding Empty Subspaces

This is mainly due to the fact that with a fixed number of bits available for a dz (e.g., 23 bits for IPv4 multicast addresses), larger the size of Ω, less fine granular is the indexing. However, it should be noted that regular spatial indexing partitions the entire space into subspaces, even those subspaces that are of no interest to any subscriber. What if the entire event space Ω does not get indexed? What if all empty subspaces w.r.t. subscription distribution in Ω are left out and the bit strings earlier assigned to these empty spaces used for more fine granular indexing of the populated subspaces? Here, we do not specifically consider the event distribution as, in any case, only those events that lie within the subscriptions are important from filtering point of view and those lying in other subspaces can be ignored. To understand the effectiveness of such selective indexing of Ω, we look at an example from Figure 3.2. We, specifically, focus on subscription sub_1 in a 2-dimensional event space comprising the dimensions temperature (T) and pressure (P). For the sake of simplicity, let us assume that only 2 bits are available to represent sub_1 through spatial indexing. Figure 3.2(a) shows that when the entire event space is indexed, then sub_1 is represented by $\{11\}$ and it receives all events lying within this subspace (highlighted in gray). Now, since there are no subscriptions in subspaces $\{00\}$ and $\{01\}$, we completely neglect these empty spaces and use the available strings for finer indexing in the populated subspaces as illustrated in Figure 3.2(b). So, in Figure 3.2(b), sub_1 is represented as $\{01\}$, and receives only the events lying within this subspace which is much smaller than the subspace representing sub_1 in Figure 3.2(a). Due to more fine granular indexing in the latter case, the false positives received by sub_1 will also be lower compared to the former case. So, to this end, we introduce the selective indexing approach where the main idea is to identify meaningful subspaces w.r.t. subscriptions in Ω and only index those subspaces instead of indexing the entire event space.

The first step in the selective indexing approach is to select subspaces in Ω populated

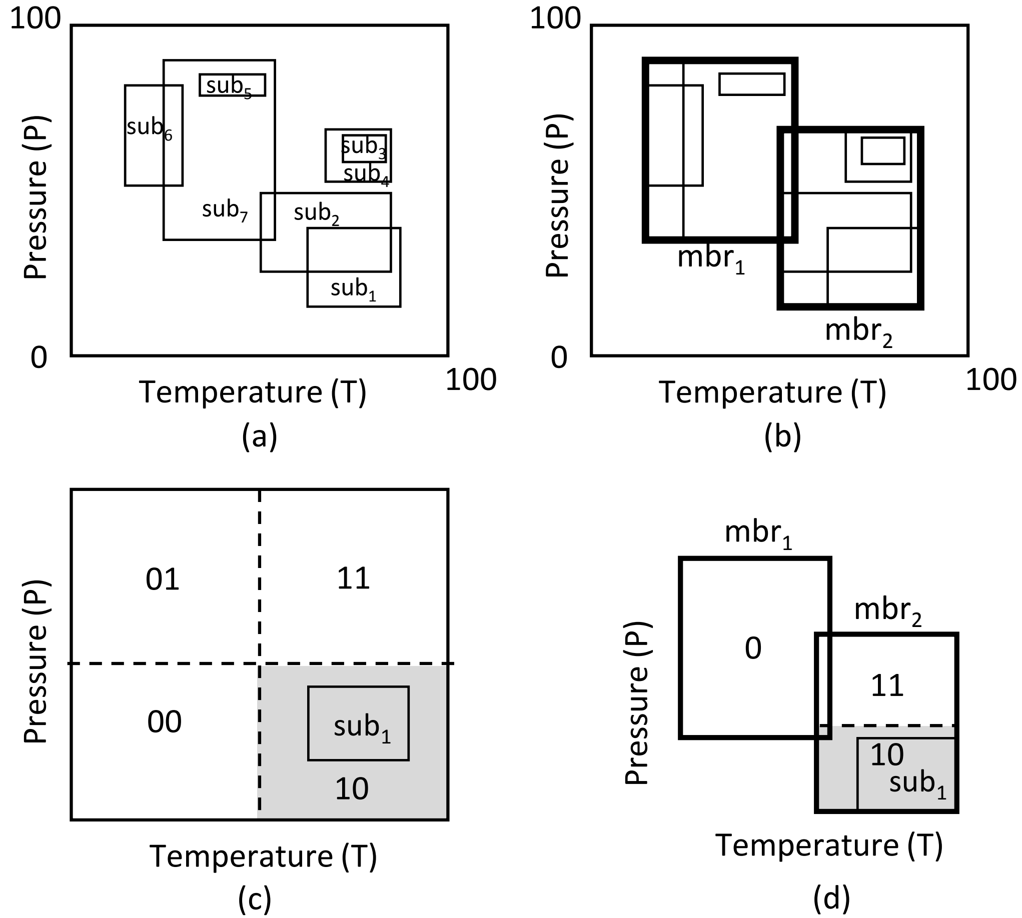


Figure 3.3: Selective Indexing

with subscriptions while identifying the empty spaces to be neglected. To identify meaningful subspaces, we benefit from the widely used mechanism of similarity-based subscription clustering [RLW+02b, BFG07]. Once subscriptions are clustered into groups, we generate polyspace rectangles which serve as the closest enclosing approximation of each of these clusters. These polyspace rectangles are known as minimum bounding rectangles or MBRs. The set of generated MBRs encloses all subscriptions in the system such that every subscription can be represented by a binary filter (or set of filters) and attempts to leave out as much empty space as possible. To understand the concept of an MBR, we provide an example from Figure 3.3. Here, the subscriptions are distributed in the event space as illustrated in Figure 3.3(a). Figure 3.3(b) shows two MBRs covering all subscriptions in the system clustered together in two groups on the basis of similarity. Please note that even though two MBRs may partially overlap as in 3.3(b), a subscription strictly belongs to a single MBR. Let us suppose that the controller chooses to have 2 MBRs for the system. So, for the purposes of our example, we proceed with the next phase of this approach with the two MBRs, mbr_1 and mbr_2,

obtained from the first phase.

Having identified the MBRs, the next phase is the actual mapping of subscriptions to dzs. We again employ spatial indexing for the binary conversion of content but, of course, now, with a difference. Spatial indexing is not employed on the entire range of values along each dimension to arrive at the dz of a subscription. Instead, spatial indexing is performed only on the range of values along each dimension of the MBR (i.e., subspace in Ω) which contains the subscription in question. This means that two subscriptions belonging to two different MBRs may end up with the exact same dz as they occupy the same relative position in their respective MBRs. However, this would be incorrect as the two subscriptions occupy different positions relative to the actual event space. This problem is mitigated by assigning unique IDs to MBRs. First, each MBR is assigned an MBR ID which is in binary form and which depends on the total number of MBRs in the system. So, if $\mathbb{M}$ is the set of MBRs in the system, then the total bits required to uniquely identify each MBR is $log_2|\mathbb{M}|$. Next, the dz representing a subscription generated by the recursive decomposition of the MBR is appended to the MBR ID that the subscription belongs to. The unique ID prefix makes a dz different from that of another MBR.

The selective indexing approach allows for more fine granular spatial indexing as it can avoid assigning bits to the subspaces in Ω that are not part of any subscription in the system, thus allowing the use of more bits to represent more meaningful subspaces. We illustrate our point in Figure 3.3(c) and Figure 3.3(d). Let us focus on the subscription sub_1 that needs to be converted to a binary string. Let us assume that again only 2 bits are available for representing content filters. Now, since there are two MBRs, one bit is required to uniquely represent them. However, this bit represents a smaller subspace as compared to what it would represent in regular spatial indexing in Ω as the empty spaces have been removed. The next step is to perform spatial indexing within mbr_2 till the closest approximation of the subscription is reached with the available number of bits. In this case, the subscription can afford just one more bit that will be appended to the MBR ID 1 for mbr_2. Therefore, for sub_1, the generated dz is $\{10\}$ as depicted in Figure 3.3(d). However, when spatial indexing is performed on entire Ω as depicted in Figure 3.3(c), false positives are more as the same dz of $\{10\}$ represents a much larger subspace in this context.

Of course, for header-based matching of packets to work, events will also need to be mapped to the selected packet header field using the selective indexing approach. For this purpose, publishers need to have information about the MBRs and their respective bounding values. As a result, the controller sends this information to each publisher whenever there is a change in MBR values. The mapping of events to the selected header field works similar to the mapping of subscriptions to match fields. However, it should be noted that MBRs may overlap. For example, in Figure 3.3(b), mbr_1 and mbr_2 overlap. In such a scenario, an event that lies in the overlapping subspace must be indexed w.r.t. both MBRs as it can match subscriptions from both. This ensures the

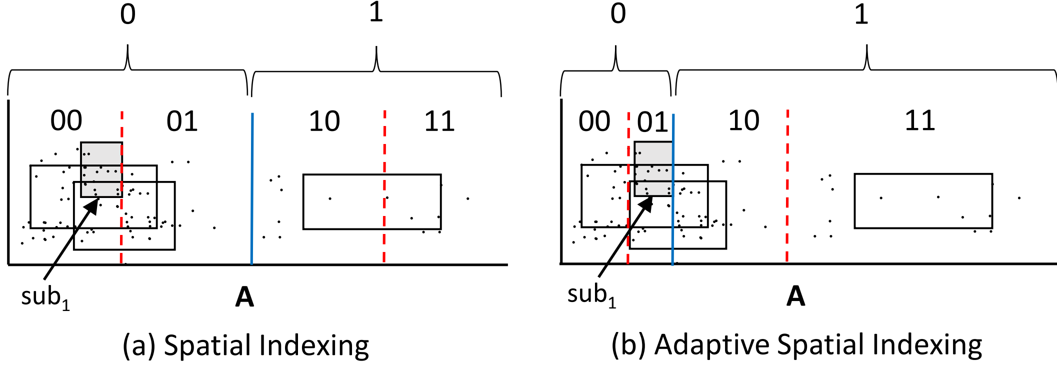


Figure 3.4: Adaptive Spatial Indexing

avoidance of false negatives, i.e., events that were not received by subscribers interested in receiving them. Also, all events that do not lie within any MBR are simply ignored by the publisher and do not get indexed.

3.2.2 Adaptive Spatial Indexing

The selective indexing approach uses regular spatial indexing to finally convert filters and events to *dz*s. As discussed in Chapter 2, spatial indexing divides the event space repeatedly to achieve subspaces of maximum possible granularity where each decomposition divides the current subspace equally into two halves. The question is, can the employed spatial indexing technique itself be modified to obtain more expressive filters and, therefore, less false positives in the system? In this section, we design an adaptive spatial indexing (ASI) approach to answer the same.

The spatial indexing technique, essentially, performs disjoint event space partitioning. We employ a similar technique but with a difference. For each recursive decomposition, instead of dividing a subspace into two equal halves in terms of range of values along dimensions, the basic idea is to divide it into two subspaces with balanced workload w.r.t. events and subscriptions. This allows indexing to have finer granularity in subspaces with higher workload in Ω. Here, it is extremely important to first define the term workload. We define workload of a subspace ss_i as $W_{ss_i} = \sum_{e_k \in E^t} |SB^{ss_i}_{e_k}|$, where $SB^{ss_i}_{e_k}$ represents the set of subscriptions within ss_i matched by an event e_k. So, when a subspace is further divided during spatial indexing along a dimension, the workload of it along that dimension is calculated and the division is made such that the resulting two subspaces have equal workload, i.e., they are not necessarily equal in terms of range of values along dimensions.

We explain the above indexing strategy with the help of an example from Figure 3.4 which depicts a 2-dimensional event space with events and subscriptions. For the sake of simplicity, we only explain indexing along one dimension, i.e., dimension A.

Let us, again, assume that only 2 bits are available for indexing. Now, while performing indexing to represent sub_1, in regular spatial indexing, the dimension range is divided equally into two subspaces {0} and {1} as depicted by the blue solid line in Figure 3.4(a). However, in our adaptive spatial indexing technique, the division is made such that the workloads in the resultant subspaces are equal. Let the blue solid line in Figure 3.4(b) illustrate this workload-based division. This allows for more fine-grained partitioning in the subspace denoted by {0} where matching traffic is heavy as compared to {1}. Further divisions in both cases, as depicted by the red dotted lines in Figure 3.4(a) and Figure 3.4(b), clearly indicate that sub_1 is represented by a much smaller subspace {01} in adaptive spatial indexing as compared to {00} in regular indexing. As a result, sub_1 suffers from fewer false positives when represented by adaptive spatial indexing.

All dimensions are divided in the exact same manner to arrive at the final dz for a subscription or an event in a multi-dimensional system. By allowing more bits to be assigned to more meaningful parts of Ω, false positives can be reduced in adaptive spatial indexing.

The efficiency of the workload-based indexing approaches w.r.t. reducing false positives may still be limited when the number of attributes (dimensions) in the system is large. As a result, the next section is dedicated to mechanisms that influence the number of dimensions to be encoded into content filters while performing in-network filtering.

3.3 Dimension Selection

As discussed before, more the number of dimensions in a system, longer are the dzs. However, what if there was no need to index every dimension? What if the available bits could be used to perform fine granular spatial indexing only on a subset of dimensions that prove to be more promising w.r.t. bandwidth efficiency? As a result, in this chapter, we use the above notion to propose and thoroughly evaluate a set of algorithms that select dimensions that are beneficial for reducing false positives and discuss their applicability, complexity, and performance w.r.t. realistic workload distributions.

3.3.1 Event Variance

The distribution of events in Ω plays a major role in determining the importance of each dimension for filtering in the system. To this end, the spread of events along a dimension is an important metric to determine the importance of that dimension. More spread would require more fine granular indexing to avoid false positives, rendering the dimension worthy of being considered for selection. More specifically, we use variance of events to measure this spread. If E^t denotes the set of all events in Ω, then event variance along a dimension d is measured as $(\sum (x_i^d - \overline{x^d})^2)/|E^t|$, where x_i^d represents

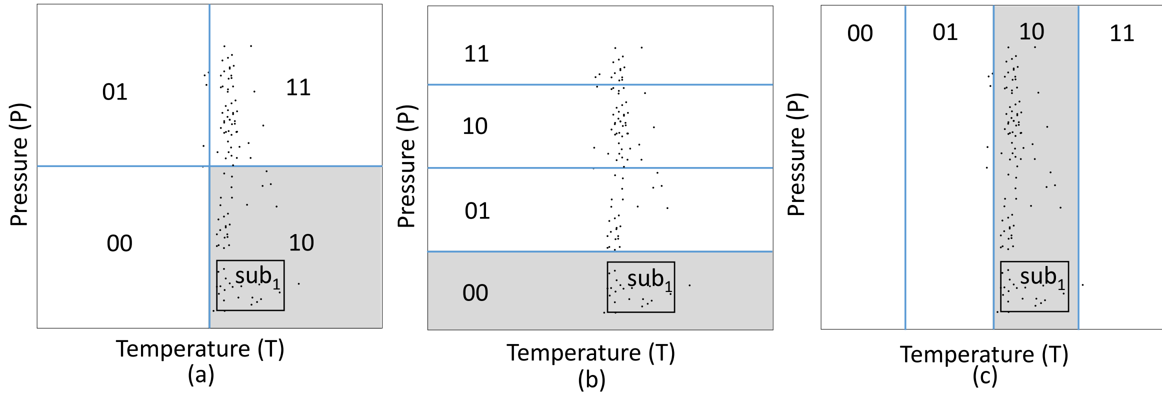


Figure 3.5: Effects of event distribution

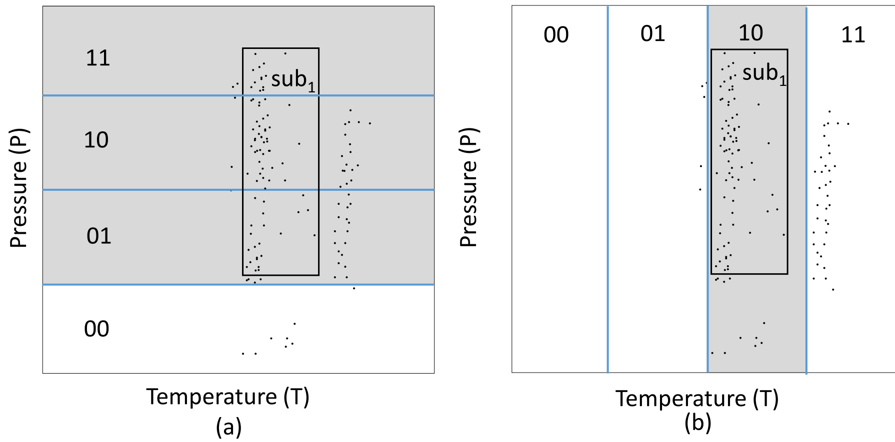


Figure 3.6: Event-based Selection

the value of the ith event along dimension d. We illustrate this with a very simple example in Figure 3.5 w.r.t. a single subscription sub_1, where the variance of events along dimension P is far greater than that along dimension T. Let us assume that only 2 bits are available for spatial indexing. Figure 3.5(a) shows spatial indexing along both dimensions according to which sub_1 is represented by the subspace $\{10\}$ which means that sub_1 receives all events lying in this subspace. Now, if only dimension P, with a high variance value for events, is selected for indexing, then sub_1 gets represented by the subspace $\{00\}$ and receives all events lying within it as shown in Figure 3.5(b). In Figure 3.5(a) sub_1 suffers from far more false positives as compared to the false positives received when only P is selected for indexing. This is because, the latter can take advantage of the fact that dimension P has a significantly high variance value for events as compared to dimension T and, thus, has the liberty of more fine granular indexing along P. As a result, most events that are irrelevant for sub_1 can be partitioned out into other subspaces. Since event variance is low along dimension T, ignoring it does not cost sub_1 much. However, if the dimension with low variance value

for events, i.e., dimension T, is selected for indexing, Figure 3.5(c) clearly shows that sub_1 would be subjected to more false positives as compared to not only indexing along dimension P but also indexing along both dimensions. This example clearly indicates the importance of event distribution within Ω in dimension selection.

So, the very first dimension selection algorithm that we present is Event Variance-based Selection (EVS). EVS calculates the variance of events along each dimension. Let $\mathbb{D}$ be the set of ω dimensions in Ω and E^t be the set of ψ events that are being considered for the algorithm in the current time window t. Let $\mathbb{SD}$ be a subset of n dimensions of $\mathbb{D}$, i.e., $\mathbb{SD} \subseteq \mathbb{D}$ and $|\mathbb{SD}| = n$. We assign, to each dimension $d \in \mathbb{D}$, a selectivity factor denoted as ϱ_d, which determines the importance of the dimension in terms of reduction of false positives if chosen for spatial indexing. Higher the value of ϱ_d, higher is the importance (selectivity) of d w.r.t. the ability to reduce false positives. For EVS, the selectivity factor ϱ_d of a dimension d is given by the variance of events along that dimension. EVS selects dimensions for $\mathbb{SD}$ by selecting n dimensions in $\mathbb{D}$ with the highest variance/selectivity factor values. Spatial indexing commences now on $\mathbb{SD}$.

The main advantage of this approach lies in its low computation overhead with a complexity of $O(\omega * \psi)$. However, the consideration of only event distribution may not be enough in every scenario. For example, in Figure 3.6(a), since event variance along dimension P is high, the subscription sub_1, when indexed along P, is represented by the subspaces $\{01\}$, $\{10\}$, and $\{11\}$ and will receive all events lying within these subspaces. However, if indexed along dimension T, with lower event variance, sub_1 is represented by the subspace $\{10\}$ and receives events lying within it as depicted in Figure 3.6(b). Here, false positives are fewer in the latter case. This clearly indicates that both events as well as subscriptions play a major role in the selection process.

3.3.2 Subscription Matching

It would be interesting to investigate the role played by subscriptions in the process of dimension selection. In fact, in doing so, we identified the importance of subscription overlaps. Dimensions where subscriptions have a lot of overlaps are less important for filtering because if an event matches a subscription along this dimension, then it matches majority of the subscriptions along this dimension, thus reducing its importance w.r.t. the ability to reduce false positives. For example, Figure 3.7(a) shows a scenario where there is a significant overlap of subscriptions along dimension P (the gray lines indicate overlaps). According to the figure, selection of dimension T would reduce more false positives than if P is selected. If indexing is performed along T, events are matched by interested subscriptions as most events are matched by disjoint subscriptions. On the contrary, if indexing is performed along P, then false positives will be high as most events match multiple overlapping subscriptions on this dimension but not along T. Note that an event is matched by a subscription if and only if it is matched on all dimensions. However, again, the selection decision cannot be taken

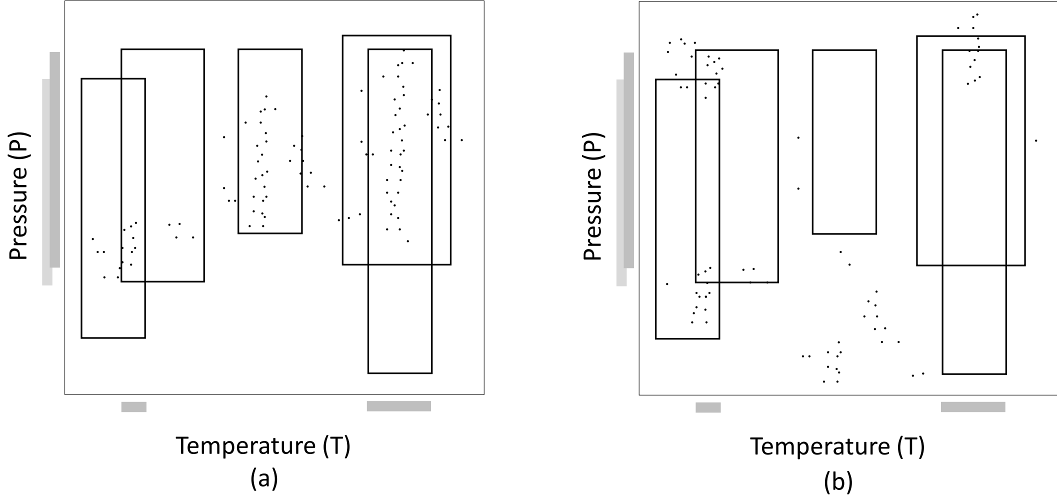


Figure 3.7: Subscription-based Selection

Algorithm 2 Event Match Count-based Selection

1: $\mathbb{D} \rightarrow$ Set of original dimensions
2: $\mathcal{SB} \rightarrow$ Set of all subscriptions
3: $E^t \rightarrow$ Set of all events
4: $\mathbb{SD} = \emptyset$ // Set of selected dimensions
5: $\varrho \rightarrow$ Set of ω selectivity factors for ω dimensions, where $\omega = |\mathbb{D}|$
6: **for all** $d \in \mathbb{D}$ **do**
7: $\quad matches = 0$
8: $\quad$ **for all** $e \in E^t$ **do**
9: $\quad\quad matches += |SB_e^d|$ // No. of subscriptions that e matches along d
10: $\quad \varrho_d = 1 - (matches/(|E^t| * |\mathcal{SB}|))$
11: $\mathbb{SD} \leftarrow$ Select dimensions corresponding to n highest values in ϱ

based on subscription overlaps alone. The reason why the selectivity of T is higher is because of not only fewer overlaps but also the distribution of events. For example, in Figure 3.7(b), we have the same subscription overlaps as before, however, due to the distribution of events, in this case, the selectivity of T is not too high.

Therefore, it is necessary to consider the combination of both subscriptions and events to determine selectivity of dimensions. As a result, we introduce another algorithm known as Event Match Count-based Selection (EMCS) which has higher computational complexity than Event-based Selection but considers both events and subscriptions to take the selection decision, rendering it more generic w.r.t. the distribution of events and subscriptions in Ω.

In the following, we provide the detailed steps of EMCS. The main idea of EMCS is to deem dimensions where event traffic matches most subscriptions as less important for dimension selection. Considering $\mathcal{SB}$ to be the total set of s subscriptions in the system, this algorithm determines the set of subscriptions that each event $e \in E^t$

matches, i.e., SB_e^d, along each dimension d and calculates the number of matches in each case, i.e., $|SB_e^d|$. Now, for each $d \in \mathbb{D}$, the selectivity factor is calculated as $\varrho_d = 1 - (\sum_{e \in E^t} |SB_e^d|)/(|E^t| * |\mathcal{SB}|)$ where the sum of all the matches of all events matching subscriptions is calculated and represented as a fraction of the maximum value possible for matches, i.e., $|E^t| * |\mathcal{SB}|$. Having calculated ϱ_d for each d, a value between 0.0 and 1.0, n dimensions with highest values of selectivity factor are added to $\mathbb{SD}$. The steps of EMCS are more formally presented in Algorithm 2. This algorithm is more generic than the previous one but has a higher time complexity of $O(\omega * \psi * s)$.

3.3.3 Correlation

Most application domains handle a large amount of data with numerous attributes. Quite often, such data has redundancy among its attributes. Redundancy in data may occur in a system due to underlying relations (i.e., correlations) between the attributes (i.e., dimensions) of the system such that the change in values in one dimension is positively or inversely correlated to the change in values in another dimension. Quite often, subscriptions and the events matching them have dimensions that are correlated or inversely correlated rendering the selection of these dimensions redundant because if an event matches a subscription in one dimension, it would also do so in the others. Such correlation between attributes exists across most applications. For instance, in IoT, most sensors detect and measure changes in various physical phenomena (i.e., dimensions) where correlations exist. For example, the sensor data set provided by the Intel Research Berkeley Lab [sen] that has a 54 node sensor network measuring values for temperature, humidity, and light shows positive correlation among all the 3 attributes [DXG+11]. Again, in a traffic monitoring scenario, for certain time periods, there may exist an inverse correlation between car speed and density. In a completely different domain, i.e., in stock exchange, it is well established that there exists a correlation between volume and stock prices [sto]. Redundancy in data can be utilized to avoid less meaningful dimensions without loss of much information while selecting dimensions. However, because of the sheer amount, data is often fuzzy making it difficult to identify such redundancy.

As a result, our next algorithm, Correlation-based Selection (CS) tries to take advantage of any redundancy in data, in the form of correlation, that may exist between dimensions while also considering the previous two factors, i.e., event variances and subscriptions across dimensions. In the previous two algorithms, the selectivity factor ϱ was independently calculated for each dimension d. However, in order to consider correlation as well, we construct a covariance matrix, $\mathbb{CM}$, which captures relations between dimensions as well as within them w.r.t. selectivity. This algorithm, formally described in Algorithm 3, consists of primarily two steps—(i) calculating the covariance matrix and (ii) performing principal component analysis (PCA) on the calculated matrix.

Algorithm 3 Correlation-based Selection

1: $\mathbb{D} \rightarrow$ Set of original dimensions
2: $\mathcal{SB} \rightarrow$ Set of all subscriptions
3: $E^t \rightarrow$ Set of all events
4: $\mathbb{SD} = \emptyset$ // Set of selected dimensions
5: $\mathbb{CM} \leftarrow$ Initialize $\omega * \omega$ covariance matrix, where $\omega = |\mathbb{D}|$
6: **for** i=0 to $\omega - 1$ **do**
7: **for** j=0 to $\omega - 1$ **do**
8: $sf_{i,j}$ = 0.0 // Similarity factor between dimension i and dimension j
9: **for all** $e_k \in E^t$ **do**
10: $sf_{e_k}^{i,j} = (|SB_{e_k}^{d_i} \cap SB_{e_k}^{d_j}|)/|\mathcal{SB}|$
11: $sf_{i,j} += sf_{e_k}^{i,j}$
12: $c_{i,j}$ = 1.0 - ($sf_{i,j}$ / $|E^t|$) // covariance value at $(i,j)^{th}$ index of $\mathbb{CM}$
13: $Q \leftarrow$ Calculate eigenvectors of $\mathbb{CM}$
14: $\Lambda \leftarrow$ Calculate eigenvalues of $\mathbb{CM}$
15: *princComp* $\leftarrow$ Select eigenvector $\in Q$ corresponding to highest eigenvalue $\in \Lambda$
16: $\mathbb{SD} \leftarrow$ Select n dimensions $\in \mathbb{D}$ with highest coefficients in *princComp*

3.3.3.1 Calculating Covariance Matrix

The basis of this approach is the calculation of the covariance matrix $\mathbb{CM}$. A covariance matrix is a symmetric matrix where each entry holds a covariance representing the relation between two random variables. In our Correlation-based Selection algorithm, we consider the random variables to be dimensions and calculate the covariance values based on the relation that must be captured between each dimension pair. As before, we consider that ω is the total number of dimensions in the system. The purpose of our covariance matrix is to identify correlations such that the dissimilarity between each dimension pair can be captured. As a result, $\mathbb{CM}$ is an (ω*ω) matrix where an element at position (i,j) represents variance or dissimilarity of the ith and the jth dimensions. $\mathbb{CM}$ captures two types of information—(i) the covariance between dimensions w.r.t. selectivity and (ii) the amount of variance within each dimension. The diagonal of $\mathbb{CM}$ captures the latter. For dimension selection, both of these information are crucial as the former highlights correlated dimensions and the latter highlights selectivity of each independent dimension.

Quite naturally, it is crucial to identify the metric representing the covariances, i.e., $c_{i,j} \in \mathbb{CM}$, depending on the type of relation between dimensions that needs to be captured. In the context of this algorithm, we define covariances between dimension pairs w.r.t. events consumed by subscriptions along each dimension. Please note that in the context of our designed middleware, if for a dimension pair, say d_i and d_j, an event matching a subscription along d_i also matches the same subscription along d_j, then such a match increases the correlation or, as we call it, similarity between d_i and d_j. As a result, we consider one dimension pair at a time and identify the similarity between them by calculating the number of times events match subscriptions along both

dimensions of the dimension pair in question. Of course, more the number of matches, higher is the correlation or similarity between the dimension pair. The inverse effect of this similarity provides the variance or dissimilarity between the dimension pair, and this is what we capture in the covariance matrix. So, an element at position (i, j) in the covariance matrix represents the dissimilarity (w.r.t. the described matches) between d_i and d_j.

We provide the steps to calculate the covariance matrix more formally as follows (cf. Algorithm 3, lines 5-12). While calculating the covariance $c_{i,j}$ between a pair of dimensions d_i and d_j, first, for each event $e_k \in E^t$, we calculate a factor called the similarity factor which calculates the set of subscriptions that the event e_k matches along both dimensions of the dimension pair. So, the similarity factor of a dimension pair d_i and d_j for an event $e_k \in E^t$ is calculated as $sf_{e_k}^{i,j} = (|SB_{e_k}^{d_i} \cap SB_{e_k}^{d_j}|)/|\mathcal{SB}|$ (cf. Algorithm 3, line 10). As before, here, $SB_{e_k}^{d_i}$ represents the set of subscriptions matched by event e_k along dimension d_i. As a result, an intersection of set $SB_{e_k}^{d_i}$ and set $SB_{e_k}^{d_j}$ provides the set of only those subscriptions that e_k matches along both d_i and d_j. The number of subscriptions in this resultant subscription set contributes to the similarity factor between the two dimensions for this event. To calculate the aggregated similarity factor ($sf_{i,j}$) between the dimension pair d_i and d_j, the similarity factors of all events are calculated as mentioned above and aggregated (cf. Algorithm 3, lines 8-11). Then, the inverse effect of this summed up (or aggregated) value is considered to measure the dissimilarity between the two dimensions in order to calculate the covariance between them. So, finally, $c_{i,j}$ is calculated as 1.0 - $\sum_{e_k \in E^t} sf_{e_k}^{i,j}/|E^t|$ (cf. Algorithm 3, line 12). This value indicates the covariance between a dimension pair w.r.t. the number of times events match subscriptions along both dimensions of a dimension pair. Along the diagonal of $\mathbb{CM}$, the variance of the match of events with subscriptions within each dimension gets captured.

3.3.3.2 Performing Principal Component Analysis

Once $\mathbb{CM}$ is calculated, the technique of principal component analysis (PCA) is applied [Jol86]. The PCA technique has found its application in pattern recognition, feature selection problems, etc., which demand mapping of data from the original dimensional space to a lower dimensional space while preserving maximum useful information [LCZT07]. Considering our objective is similar, we apply the technique of PCA on $\mathbb{CM}$ to identify the dimensions along which the variance in the event traffic matched by subscriptions is maximized.

Without going into much mathematical details, we describe the main steps required to select dimensions using PCA (cf. Algorithm 3, line 13-16). First, $\mathbb{CM}$ is subjected to spectral analysis through the process of eigendecomposition, i.e., $\mathbb{CM} = Q\Lambda Q^T$, where $\Lambda = \{\lambda_1, \ldots, \lambda_\omega\}$ is a diagonal matrix of eigenvalues and $Q = \{q_1, \ldots, q_\omega\}$ is the matrix

whose columns are orthogonal eigenvectors of $\mathbb{CM}$. So, eigendecomposition projects the original dimensions (in Ω) onto an orthogonal basis of vectors called eigenvectors. This transformation makes the highest variance by any projection of the dimensions to lie on the very first axis (i.e., first principal component). In fact, as proposed by [MG04], an eigenvector q with largest eigenvalue represents the dimension (in the orthogonal basis) along which variance is maximized (i.e., first principal component), and thus this eigenvector q is used to rank the original dimensions. In more detail, a higher absolute value of ith coefficient of q indicates that the dimension d_i is more important to be used for filtering. Thus, the dimensions (in the original space) that correspond to the first n coefficients with higher magnitude are selected for filtering. CS efficiently chooses dimensions based on the idea of reducing redundancy in data while maximizing variance of events matched by subscriptions. The time complexity of the calculation of the covariance matrix itself is $O(\omega^2 * \psi * s)$, rendering the algorithm more complex than the previous two.

3.3.4 Evaluation-based Techniques

The previous algorithms, though effective in their own ways, do not give an indication of an ideal value of n. So, in this subsection, we introduce two algorithms which not only significantly reduce false positives in the system, but also provide the most suitable value for n. Since the controller has knowledge of both $\mathcal{SB}$ and E^t, we can implement evaluation-based techniques to simulate false positives in the system for various combinations of dimensions and choose the most beneficial one, thus obtaining even a suitable value for n. The performances of these techniques are more optimal as compared to the previous three algorithms but have relatively higher computational complexities.

Ideally, in order to obtain an optimal set $\mathbb{SD}$, a brute force technique must be employed which calculates the false positives for all combinations of dimensions and finally selects the one producing least false positives. In order to do so, a complete simulation of the entire filtering process must be performed at the logically centralized controller, given a fixed value of the number of available bits for filter representation. With the information of the actual subscription and event values, their corresponding mappings to binary strings, the false positive rate can be determined for each combination of dimensions. However, running such a simulation has exponential computation overhead of $O(2^\omega * \omega * s * \psi)$.

We reduce the complexity of the brute force algorithm by using a greedy strategy which is also based on simulation but does not evaluate every combination of dimensions. Initially, the combination with all ω dimensions in $\mathbb{D}$ is considered, and the resulting false positive rate is noted. Then, all combinations with ω-1 dimensions are evaluated, i.e., each combination has ω-1 dimensions but in each combination a different dimension is removed. The combination with the lowest false positive rate is selected and in the pro-

cess one dimension gets removed. The next cycle uses this selected combination with ω-1 dimensions as input and evaluates all combinations with ω-2 dimensions to arrive at the most beneficial combination for ω-2 dimensions. The process continues till the number of dimensions being considered for the combinations is reduced to 1 by incrementally removing one dimension in every step. So, we have a total of ω combinations where the first combination consists of ω dimensions, the second consists of ω-1, and so on till the last (ω^{th}) combination contains 1 dimension. Quite often, with decreasing number of dimensions, the false positive rate decreases till the redundancies in data are removed, after which the rate increases again due to loss of important information with further reduction in dimension count. As a result, different combinations with different dimension counts can be expected to reduce different number of false positives. So, of all the aforementioned ω combinations, the one producing least false positives is chosen for $\mathbb{SD}$. By employing such a technique, we essentially also obtain the most suitable value of n. The greedy strategy has a time complexity of $O(\omega^3 * \psi * s)$.

3.4 Handling Dynamic Network Updates

All of the above discussed methods rely heavily on past event traffic and subscription distributions. However, the event distribution and the current subscriptions in the system may change over time, degrading the effectiveness of the proposed techniques. So, the controller must periodically collect workload information over time to monitor the recent distribution, execute proposed techniques, and deploy necessary changes in the network. For example, in the case of dimension selection algorithms, the event traffic distribution may change over time and the dimensions that were selected previously by the dimension selection algorithm may need to be replaced in the next period. This implies that the indexing of content will be done for a different set of dimensions now resulting in completely different dzs. As a result, a new set of flows would need to be deployed in the network. So, all the techniques described in this chapter require periodic updates to flows in the network (i.e., removal of existing flows and deployment of new flows that replace the existing ones) according to the current indexing decisions.

However, with the need for network updates comes the problem of ensuring consistency in the data plane. Please recall from Chapter 2 that the network state is represented by network configuration that consists of all flows on all switches in the network. Let us denote this network state or network configuration as NS. So, when the transition from one network state to another is being performed, the event packets in transition in the network may be incorrectly dropped or forwarded. Let us consider an example of a system where indexing is performed on 4 dimensions, A, B, C, and D, resulting in a network state NS_o. Let us assume that dimension selection is employed to improve the bandwidth efficiency of the system, and now, spatial indexing is performed on only 3 dimensions, B, C, and D. In such a scenario, the existing flows need to be removed and new flows according to the new indexing (resulting in a network state

NS_n) must be deployed as the old dzs, representing all 4 dimensions, are semantically different from the new dzs, representing only 3 dimensions. So, while the transition from NS_o to NS_n is being performed, event packets in transition and targeted to follow NS_o, may no longer find a path through NS_o or/and be incorrectly forwarded by NS_n which is semantically different from the event packet in question. The same applies to event packets targeted at NS_n. The difference in semantics can be further explained through an example depicted in Figure 3.5 where, when indexed along both dimensions, temperature and pressure, sub_1 has the dz {10} (cf. Figure 3.5(a)) but on indexing only along the dimension pressure, it has a dz {00} (cf. Figure 3.5(b)). Clearly, in the context of the new index, the old one has completely different semantics and an event published during transition, say with a dz {10001} lying within sub_1 and indexed according to the old dimension set will no longer find a match in the newly deployed flow representing sub_1 which now matches {00*}. As a result, this event may be dropped due to the absence of any flow matching it or may be incorrectly forwarded by a flow matching the event but representing a different subscription according to the new index. So, additional mechanisms must be employed to ensure that packets are not lost or incorrectly forwarded in the data plane during transitions.

3.4.1 Data Plane Consistency in PLEROMA

A lot of work has already been dedicated to ensuring data plane consistency in SDN [RFR+12, JLG+14]. Most works attempt to provide a general solution to data plane consistency for any application and are, therefore, computation intensive and/or resource intensive. However, in our case, we can design a middleware-specific solution, i.e., a light-weight approach, that targets only those data-plane consistency issues that affect the functional requirements of our specific system. Data plane consistency in SDN is primarily characterized by three properties—(i) blackhole-freedom, i.e., a packet that should be forwarded by a switch should not be dropped during the transition, (ii) loop-freedom, i.e., no packet should loop in the network, and (iii) packet coherence, i.e., no packet should see a mix of old and new flows belonging to the old (NS_o) and new (NS_n) network states, respectively.

Please note that the PLEROMA middleware installs paths between publishers and subscribers by first creating an acyclic spanning tree that covers all switches in the network and then embedding content filters along these paths. This ensures the existence of only a single path between two hosts of a network. Also, here, when we talk about transitions from one network state to another, we only talk about changing the content filters that are embedded along the path connecting a publisher to a subscriber, i.e., the path itself between two hosts remains the same in a transition. As a result, there is no possibility of cycles or loops in the network due to the transition because of which no additional measures need to be taken to ensure the inherent loop-freedom property of our system. So, in the context of our middleware, the main consistency properties

that we try to ensure are blackhole-freedom and packet coherence. Ensuring these two properties is essential for the system as both of these can lead to false negatives which is not tolerated in our system.

3.4.2 Light-Weight Approach

To ensure the above two properties, the main idea is to continue to have a path connecting a publisher to a subscriber when flows are being updated while also ensuring that a packet sees only one network state while being forwarded to its destination. So, we design a light-weight approach where the main idea is, also, to always have a path connecting each publisher to its relevant subscribers so that false negatives can be avoided. In fact, we use a temporary intermediate network state that can be traversed by events targeted at either the old network state or the new one while network updates are being performed. Let us take an example depicted in Figure 3.8 where there is a need to transition from NS_o (cf. Figure 3.8(a)) to NS_n (cf. Figure 3.8(e)) on switch R_1. In Figure 3.8 each flow in the flow table of R_1 is represented by its incoming port (iP), match field (represented by dz), outgoing ports (oP) which specifies the ports through which matching events are forwarded, and flow priority (PO). The priority of a flow may be important in certain cases as please recall that when an event satisfies the matching criteria of multiple flows, the flow with the highest priority is allowed to forward it.

Once the decision to make the switch to NS_n is taken, first, a resource-efficient intermediate network state, i.e., NS_I, comprising flows matching any pub/sub event is installed along all paths connecting publishers to their relevant subscribers. The purpose of this temporary intermediate state is to always have paths for any pub/sub event no matter which network state it is targeted at. More specifically, for each incoming port, say iP, on a switch, all flows on the switch, belonging to the old network state NS_o, which have iP as their incoming port are identified, and a set of outgoing ports oP is created from the union of all outgoing ports to which the identified flows forward packets on account of a match. For example, in Figure 3.8(b), for the incoming port iP=1, the flows fl_1 and fl_2 are identified, and a union of the outgoing ports to which these flows forward events is performed yielding the outgoing port set oP={2,3}. Next, a single flow that forwards all pub/sub traffic (representing the entire event space Ω) through all ports in oP is installed on the switch for this incoming port. So, in Figure 3.8(b), this is represented by fl_4 which forwards all incoming pub/sub traffic through the outgoing ports {2,3}. Please note that as this flow represents the dz {*}, covering entire Ω, it forwards any pub/sub traffic, irrespective of the semantics of the event and as long as it is part of the pub/sub traffic. This is done for every incoming port on each switch of the network during the transition. So, in the example in Figure 3.8, the same is done for the incoming port 3 as there exists a flow fl_3 in the old network state where incoming traffic arrives at port 3. The flows, covering the entire

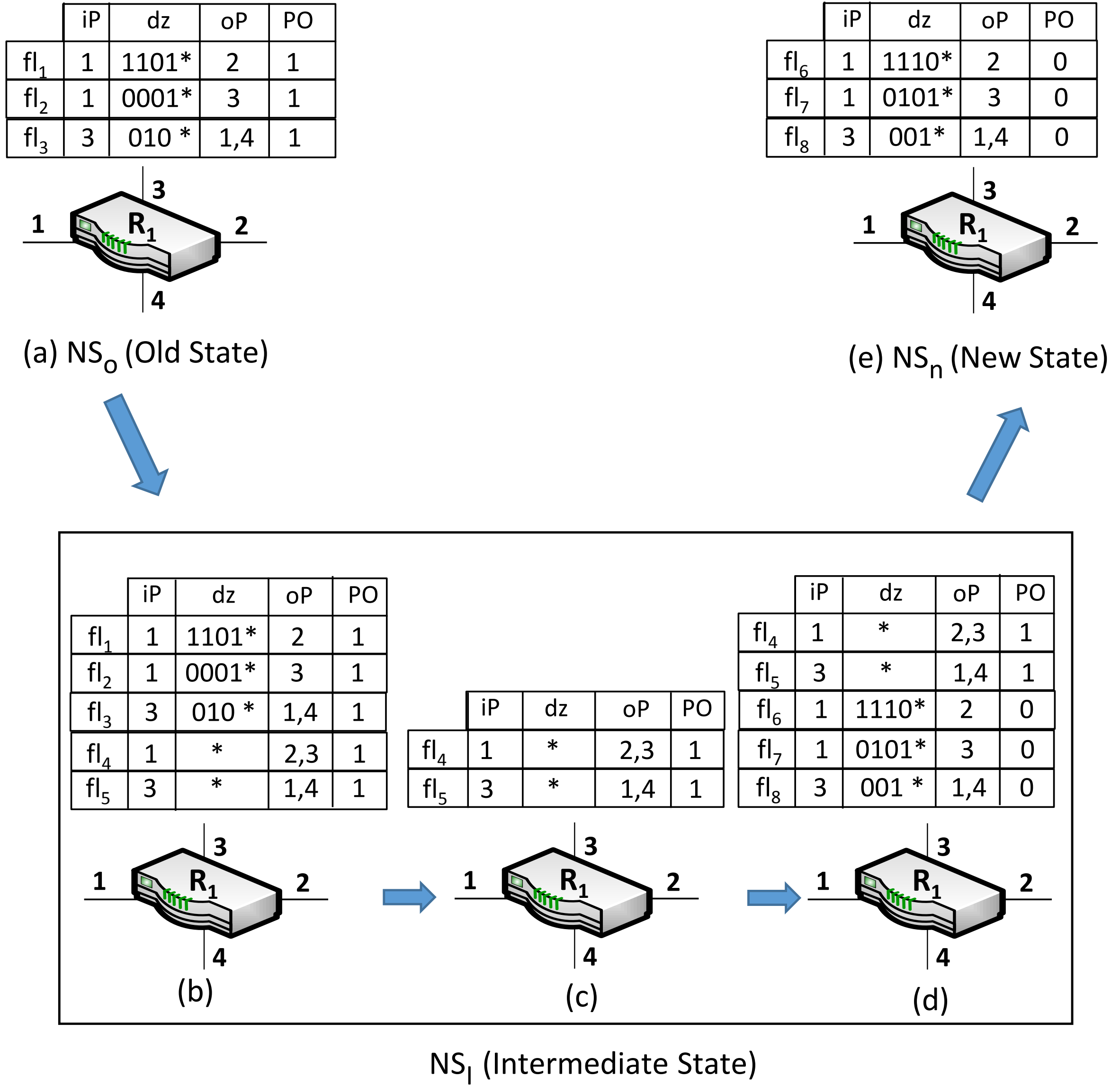


Figure 3.8: Light-Weight Approach

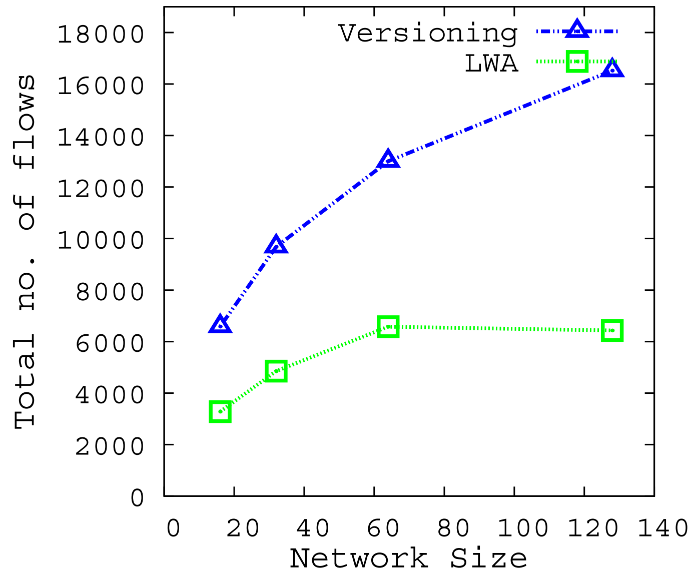


Figure 3.9: Versioning vs light-weight approach (LWA)

event space, constitute the intermediate network state NS_I. For each incoming port of a switch, replacing fine-grained filters of NS_o with a single flow covering Ω ensures the use of minimum additional flows during the transition to maintain data plane consistency. In fact, in our light-weight approach, at any given time, the maximum number of additional flows installed on a switch to avoid false negatives is the total number of incoming ports of the switch. This is because at most a temporary flow belonging to NS_I may be added for each incoming port. This is in sharp contrast to the traditional versioning method [RFR+12] which would result in almost double the number of old flows during the transition to guarantee consistency. The impact on number of flows in the pub/sub network for our light-weight approach as compared to the versioning method is depicted in Figure 3.9. Please note that TCAM is a very expensive and power-hungry resource and current vendors design SDN-compliant switches that can accommodate only a few thousand flows [CMT+11]. Clearly, there may not be any TCAM space available to install another version for the same filters as is the case with the traditional versioning method, especially when there can be a huge number of subscribers in the system where each subscription may be represented by multiple content filters. So, where TCAM is scarce, the figure clearly shows the advantage of our designed approach over the traditional versioning method.

Once NS_I is deployed in the network, the flows constituting NS_o can be removed as there is already an alternate set of flows connecting each publisher to at least its relevant subscribers through NS_I. This step is depicted in Figure 3.8(c). Once NS_o is removed, flows constituting NS_n are added to the switches of the network (cf. Figure 3.8(d)). However, the priority of the flows in NS_n is kept lower than the priority of the flows in NS_I such that any event that was targeted at NS_o matches the flows in NS_I and never

those in NS_n. We do so to satisfy the packet coherence property in the context of our approach that prohibits a packet to see a mix of old and new flows belonging to NS_o and NS_n, respectively. Please note that in our approach we provide a form of packet coherence where packet coherence is not considered to be violated if a packet sees a mix of old ($\in NS_o$) and intermediate ($\in NS_I$) flows or a mix of intermediate ($\in NS_I$) and new ($\in NS_n$) flows. This is mainly because the intermediate state semantically applies to both NS_o and NS_n, and an event belonging to either is a valid event for NS_I. Once all flows in NS_n have been deployed, the publishers are notified to index events according to the new chosen indexing approach such that events can now be targeted towards the already deployed new network state NS_n. After a given time bound (depending on the bounds on the forwarding latency of the longest path in the network) that ensures that all events matching the old network state NS_o have been delivered to the subscribers, all flows constituting NS_I are removed (cf. Figure 3.8(e)).

In this way a transition from NS_o to NS_n is performed in the network without violating the blackhole-freedom and packet coherence properties. As a result, the pub/sub system does not suffer from any false negatives. Of course, as, for each incoming port, the fine-grained content filters are temporarily substituted by a single flow representing the entire event space, the pub/sub system experiences additional false positives temporarily during the transition.

3.5 Performance Evaluations

This section is dedicated to evaluating and analyzing the performances of each of the presented approaches. We conduct a series of experiments to measure and compare the overall false positive rate at the subscribers of an SDN-based publish/subscribe system for all the techniques. We, especially, show the impact of different types of workload on the performance of each of the approaches in order to highlight their applicability in various scenarios.

3.5.1 Experimental Setup

For our experiments, we have used *SDN-m* (cf. Chapter 2) built on the prominent tool for emulating software-defined networks, i.e., Mininet. We use Mininet to experiment with up to 128 switches and 256 end-hosts on different topologies. Our evaluations include up to 10,000 subscriptions and up to 100,000 events. In order to generate workload, i.e., events and subscriptions we use both synthetic as well as real world data. In synthetic data, a content-based schema containing up to 8 attributes (dimensions) is used where the domain of each attribute varies between the range [0,4095]. We, again, use two models for the distributions of subscriptions and events to generate synthetic data—uniform model and the interest popularity model that chooses

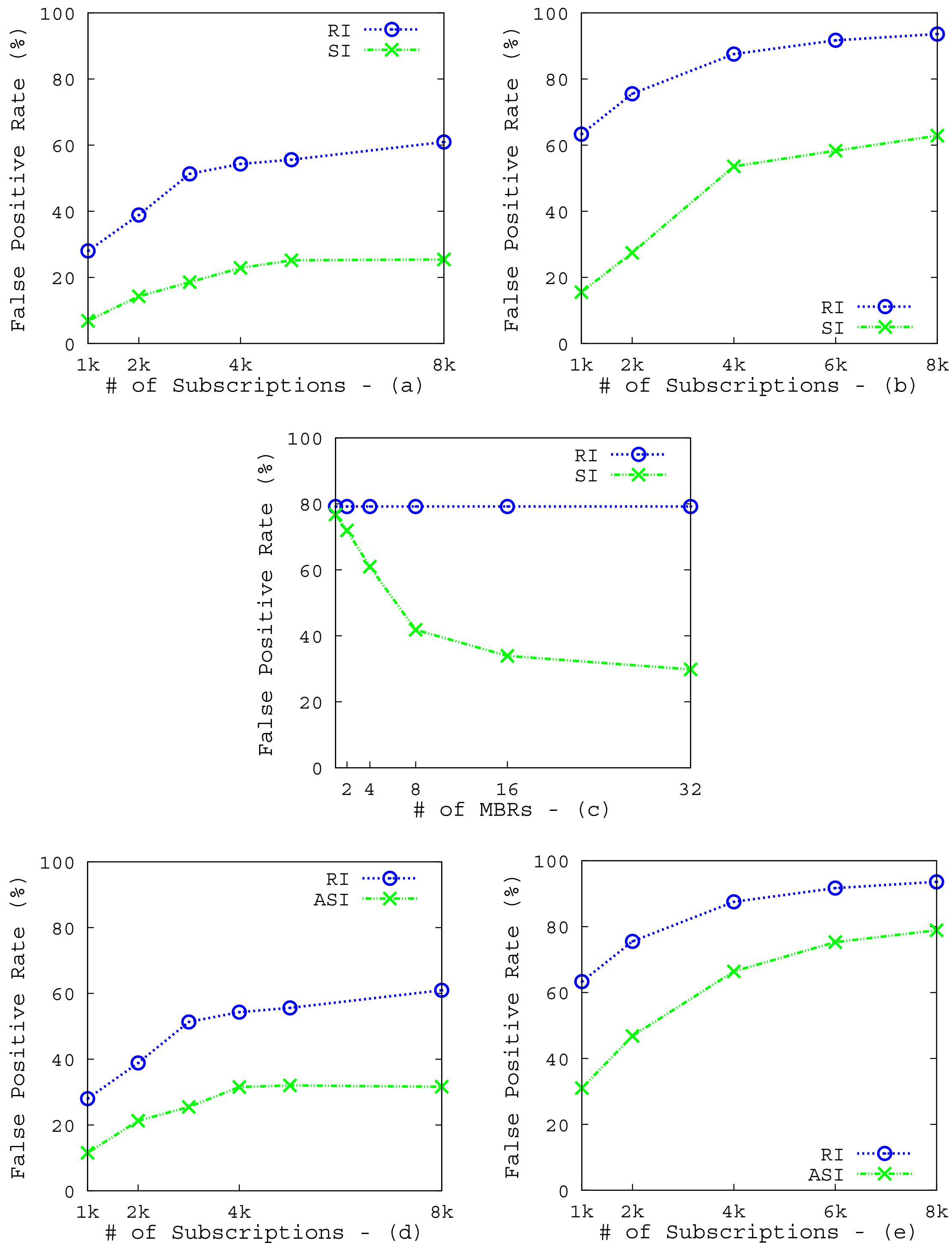


Figure 3.10: Performance Evaluations: Workload-based indexing

up to 8 hotspot regions around which it generates subscriptions and events using zipfian distribution. We use synthetic data to especially highlight certain properties of the designed algorithms by adjusting correlation between dimensions, matching traffic variances, number of correlated dimensions, etc. We also use real-world workload in the form of stock quotes procured from Yahoo! Finance containing a stock's daily closing prices [CJ11] to show the performance of our algorithms in a realistic environment. In the following evaluations, we show the effectiveness of our techniques even when the number of available bits for spatial indexing is restricted to just 23 bits as available in IPv4 multicast addresses.

3.5.2 Workload-based Indexing

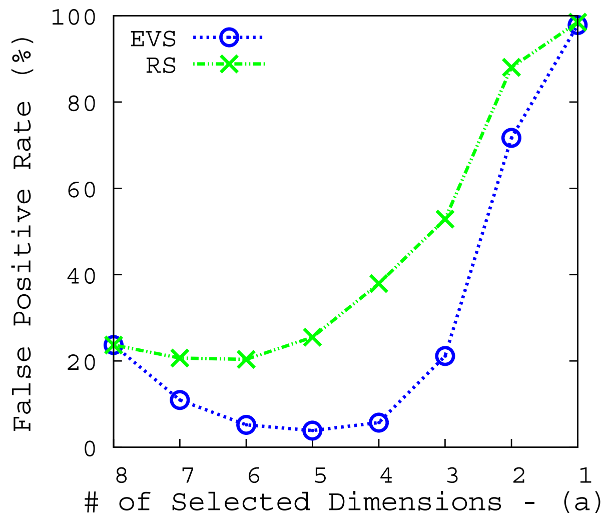
The first set of experiments evaluates the behavior of the selective indexing (SI) approach when subjected to both uniform as well as zipfian data. Figure 3.10(a) plots the false positive rate with increasing number of subscriptions for both selective indexing as well as regular indexing (RI) when uniform data is used. Figure 3.10(b) shows the same when zipfian data is used instead. These plots show that indexing within MBRs has significant benefits over regular indexing. For both uniform and zipfian data and for every subscription count, SI results in a lower false positive rate when compared to RI.

The effectiveness of selective indexing, however, depends largely on the number of MBRs used. As a result, our next set of experiments evaluates the behavior of this approach with increasing number of MBRs. We use zipfian distribution to generate events and subscriptions around 8 hotspots and then measure the false positive rate while varying the number of MBRs. Figure 3.10(c) clearly shows that the false positive rate reduces rapidly when the number of MBRs is varied between 1 to 8. After 8 MBRs, the rate reduces less rapidly because clustering of data generated around 8 hotspots into more than 8 MBRs does not have as significant additional benefit as before.

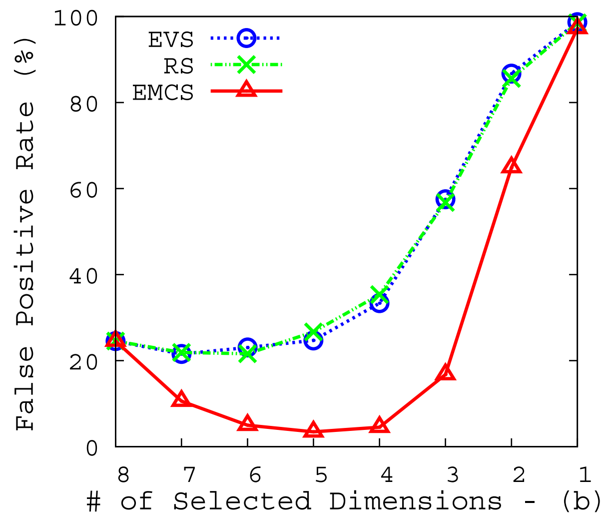
We, also, conducted a set of experiments to evaluate the impact of adaptive spatial indexing (ASI) on false positive rate of a system. We evaluated the effect of this technique when both uniform and zipfian data are used to generate events and subscriptions. Figure 3.10(d) (uniform) and Figure 3.10(e) (zipfian) clearly show that in-network filtering gains from the use of adaptive spatial indexing as opposed to regular indexing. For both uniform and zipfian data and for every subscription count, ASI results in a lower false positive rate when compared to RI and the plots behave similar to SI.

3.5.3 Dimension Selection

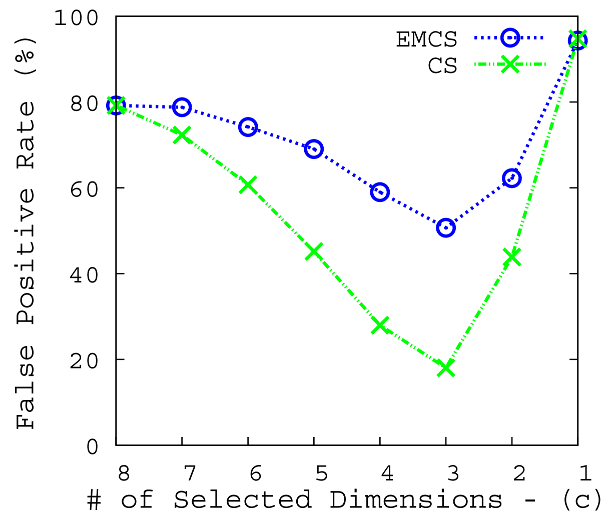
We conducted a series of experiments to evaluate the behavior of all presented dimension selection algorithms when subjected to various types of workload. In the following



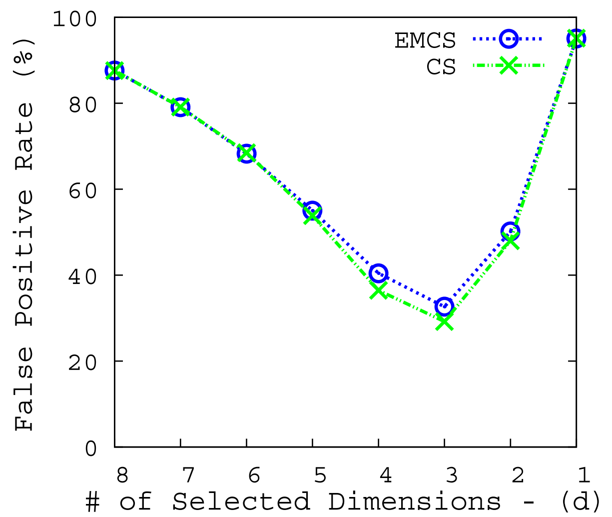
EVS
RS
False Positive Rate (%)
100
80
60
40
20
0
8 7 6 5 4 3 2 1
# of Selected Dimensions - (a)



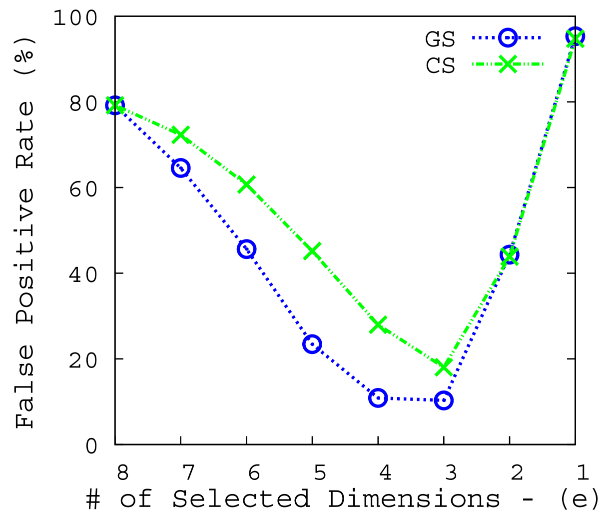
EVS
RS
EMCS
False Positive Rate (%)
100
80
60
40
20
0
8 7 6 5 4 3 2 1
# of Selected Dimensions - (b)



EMCS
CS
False Positive Rate (%)
100
80
60
40
20
0
8 7 6 5 4 3 2 1
# of Selected Dimensions - (c)



EMCS
CS
False Positive Rate (%)
100
80
60
40
20
0
8 7 6 5 4 3 2 1
# of Selected Dimensions - (d)



GS
CS
False Positive Rate (%)
100
80
60
40
20
0
8 7 6 5 4 3 2 1
# of Selected Dimensions - (e)

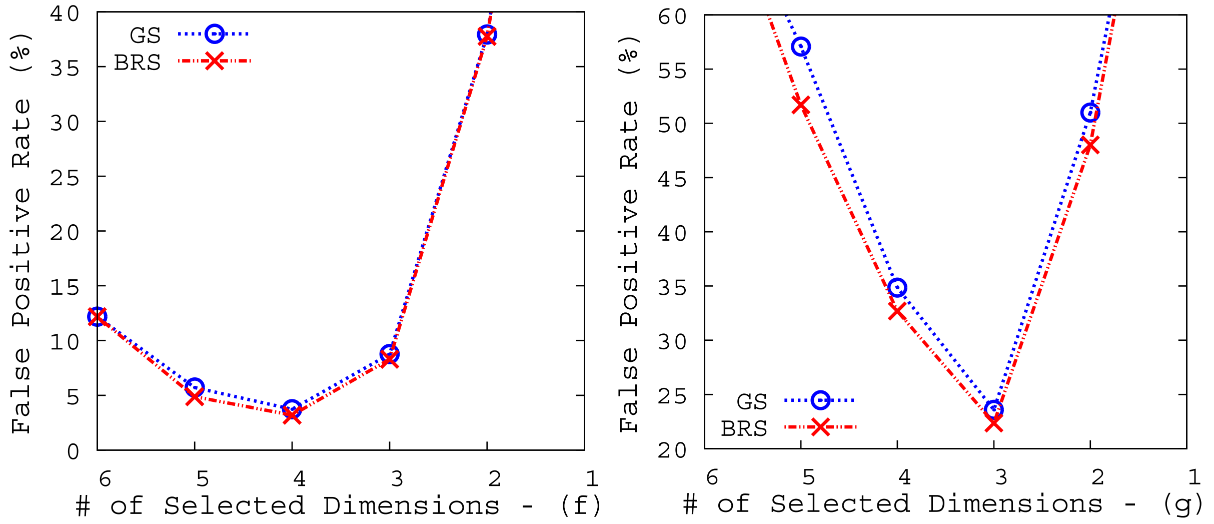


Figure 3.11: Performance Evaluations: Dimension Selection - False Positive Rate

experiments, we primarily calculate the false positive rate at the subscribers of the system when the number of selected dimensions is gradually reduced for a specific workload. We also evaluate the runtime of each approach to compare their complexities.

3.5.3.1 False Positive Rate

While generating workload (i.e., subscriptions and events), we mainly specify two factors. The first is the variance factor which can be either random or uniform. Random variance factor means that the variance of events in certain dimensions may be high whereas they may be low in others, and this is decided at random. Uniform variance factor signifies similar variance of events across all dimensions. The second factor that we define is the correlation factor. Here, a high correlation factor implies high correlation between multiple dimensions while very few dimensions are independent whereas a low correlation factor signifies low correlation between very few dimensions while most dimensions are completely independent.

The first set of experiments is dedicated to evaluating the performance of the least complex algorithm, Event Variance-based Selection (EVS). These experiments not only highlight the benefits of dimension selection on reduction of false positives but also show that even a simple approach like EVS performs better than a random dimension selection (RS) approach. Figure 3.11(a) plots false positive rate when EVS and random selection approaches are employed on multiple data sets having 8 dimensions with a random variance factor. The figure shows that, when EVS is used, reducing dimensions up to a point reduces false positives, but, after that, false positives rise again. This is because, for example, in the case of Figure 3.11(a), EVS benefits by removing 3 less

selective dimensions and assigning the additional bits to the 5 more selective dimensions. However, ignoring one or more of these 5 dimensions implies major information loss which again increases the false positive rate. EVS performs better than a random selection approach as it takes advantage of the random variance factor which allows certain dimensions to have higher selectivity than the others.

We evaluated the next set of experiments, however, with uniform variance factor instead of a random variance factor as before. We again plot the performance of EVS in such a scenario and as expected, due to uniform event variance in all dimensions, it does not succeed in reducing false positives as can be seen in Figure 3.11(b). In fact, its performance can be compared to random selection. However, in such a scenario, the Event Match Count-based Selection (EMCS) approach performs much better than EVS, providing a significant benefit in terms of reduction of the false positive rate (cf. Figure 3.11(b)). When event distribution alone cannot differentiate between selectivity of dimensions, then it is necessary to look at both events and subscriptions to determine selectivity, and this is the reason why EMCS performs much better in this case.

EMCS works very well in the previous scenario. However, in the following experiments we compare its performance to Correlation-based Selection (CS) when the correlation factor is both high and low. Figure 3.11(c) plots false positive rate when selected dimensions are gradually reduced for data with high correlation factor. The figure clearly shows that CS gains significantly over EMCS in the presence of high correlation. When the correlation factor is low, quite understandably EMCS and CS perform similarly as depicted in 3.11(d). However, please note that even with low correlation CS does not perform worse than EMCS.

The next set of experiments compares the performance of the greedy selection (GS) algorithm with CS when a high correlation factor is used while data generation. Figure 3.11(e) shows that GS outperforms CS even in the very best case for CS, i.e., high correlation. Since GS is an evaluation-based technique, it performs in most cases better than the other techniques and is very close to the performance of ideal selection, i.e., Brute-Force Selection (BRS), as can be seen in Figure 3.11(f) and Figure 3.11(g) for uniform and zipfian data, respectively. BRS, of course, produces the most optimal set of dimensions but, as can be seen from the evaluation results, the performance of GS is almost equivalent to this optimal.

3.5.3.2 Runtime Overhead

The dimension selection evaluation results show the performance of the selection algorithms in increasing order of effectiveness, i.e., EVS, EMCS, CS, GS, and BRS. However, better the performance, higher is the time complexity of the selection algorithm. This is visible in the next set of experiments that we conducted. The first set of experiments shows the impact of increasing the number of events on the time required to select a set of 4 dimensions from a set of 8 dimensions when the number

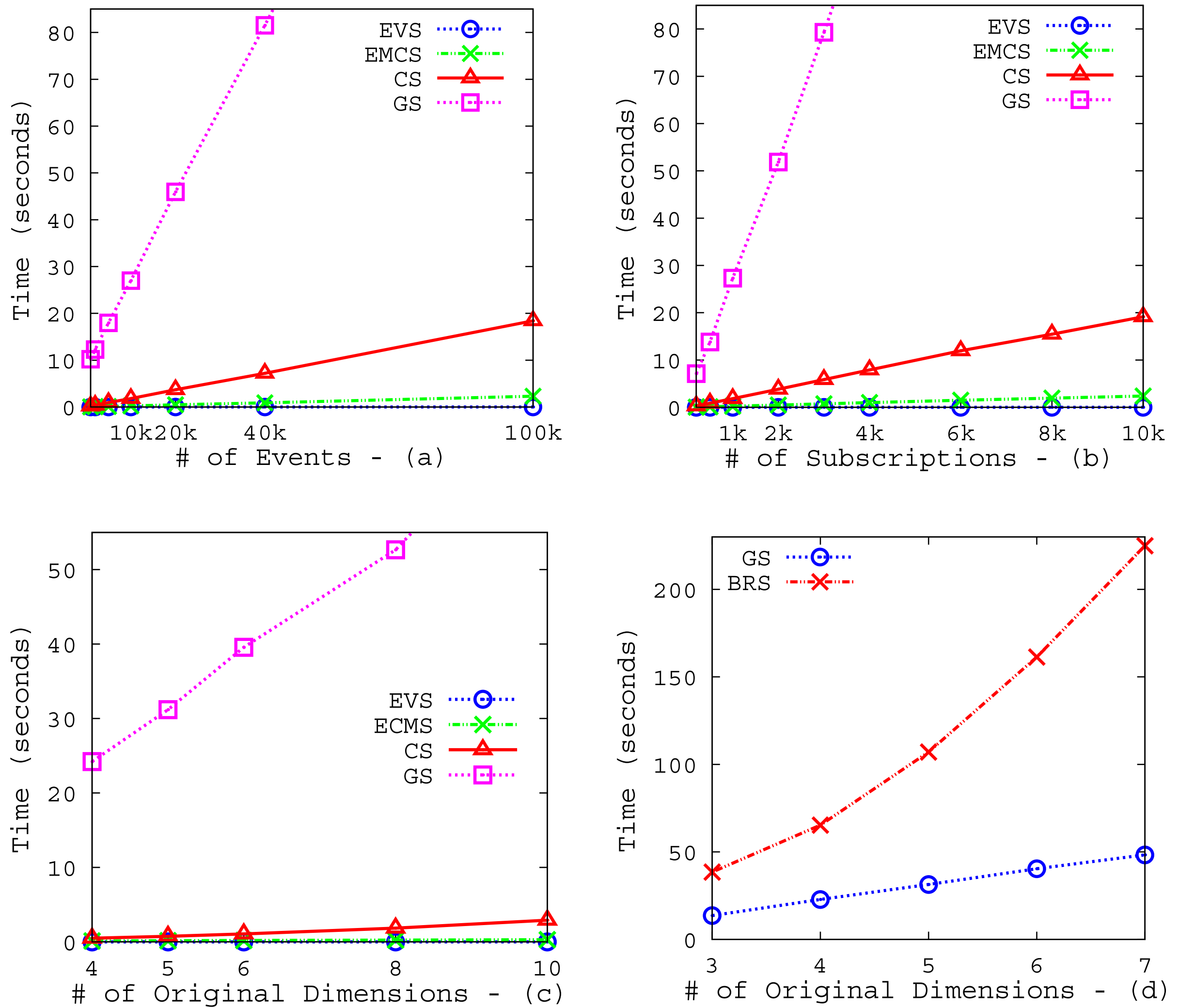


Figure 3.12: Performance Evaluations: Dimension Selection - Runtime Overhead

of subscriptions is fixed to 1000. Figure 3.12(a), clearly shows that EVS and EMCS require least computation time (in the order of milliseconds), whereas CS takes significantly more time than them with GS requiring most. Similarly, the impact of number of subscriptions on computation time, with the event count set to 100,000, can be seen in Figure 3.12(b). As expected, again EVS performs fastest, followed by EMCS, CS, and finally GS. The impact of number of original dimensions on computation time can also be seen in Figure 3.12(c) when number of subscriptions is set to 1000 and events set to 100,000. We, also, conducted evaluations to compare computation time between GS and BRS with increasing number of original dimensions. Figure 3.12(d) shows that for BRS computation time increases, as expected, rapidly with increasing number of input dimensions.

3.5.4 Combining Approaches

The next set of experiments are dedicated to highlighting the effect of combining various algorithms. We used zipfian distribution to generate data for these experiments with a random selectivity factor. Figure 3.13(a) shows the performance of CS and GS both independently and when combined with adaptive spatial indexing. As expected, the combinations perform much better than CS or GS alone. In fact, for GS+ASI, the false positive rate goes down from 80% (if regular spatial indexing is performed on 8 dimensions) to merely 3.33%.

Figure 3.13(b) shows the performance of CS and GS both independently and this time when combined with selective indexing. Here too, the combined approaches outperform the others. In fact, GS+SI reduces false positive rate in the system from 80% (if regular spatial indexing is performed on 8 dimensions) to an almost negligible 2% (∽97% reduction in false positive rate).

To ensure that our approaches are effective in realistic scenarios, we conducted experiments to show their effects on real-world stock data. As can be seen in Figure 3.13(c), our algorithms are capable of significantly reducing false positives in a real-world system. This time we combine EVS, CS, and GS with selective indexing. The plots show that even an approach combined with EVS reduces false positive rate by 48% when 2 dimensions are selected. Also, in this case, GS , CS, and EVS, when combined with SI, have very similar performances. GS successfully reduces the false positive rate by up to 53%. These evaluation results further highlight the applicability of the approaches presented in this chapter.

3.5.5 Handling Dynamics

The final set of experiments have been conducted to show the dynamic behavior of the system with the passage of time in Figure 3.14. So, we start evaluating false positive rate for a system on which CS has been recently employed and then plot its behavior with changing dynamics over time. Initially, the false positive rate is quite low due to the recent execution of CS that selected dimensions based on the recent traffic distribution. However, around the 95th second the traffic distribution changes because of which the dimensions chosen in the previous period become less effective. As a result, the false positive rate goes up significantly in the system. Around the 350th second, CS is again executed, and indexing is done based on the current selected dimensions chosen according to the current traffic distribution. Now, new flows are installed in the system following the light-weight approach. Here, we define the term false negative rate as the percentage of events dropped in the network that should have been forwarded to interested subscribers. Our evaluation results confirm that there are no false negatives in the system when LWA is employed. As expected, during this process, the false positive rate is very high as the fine-grained filters are temporarily

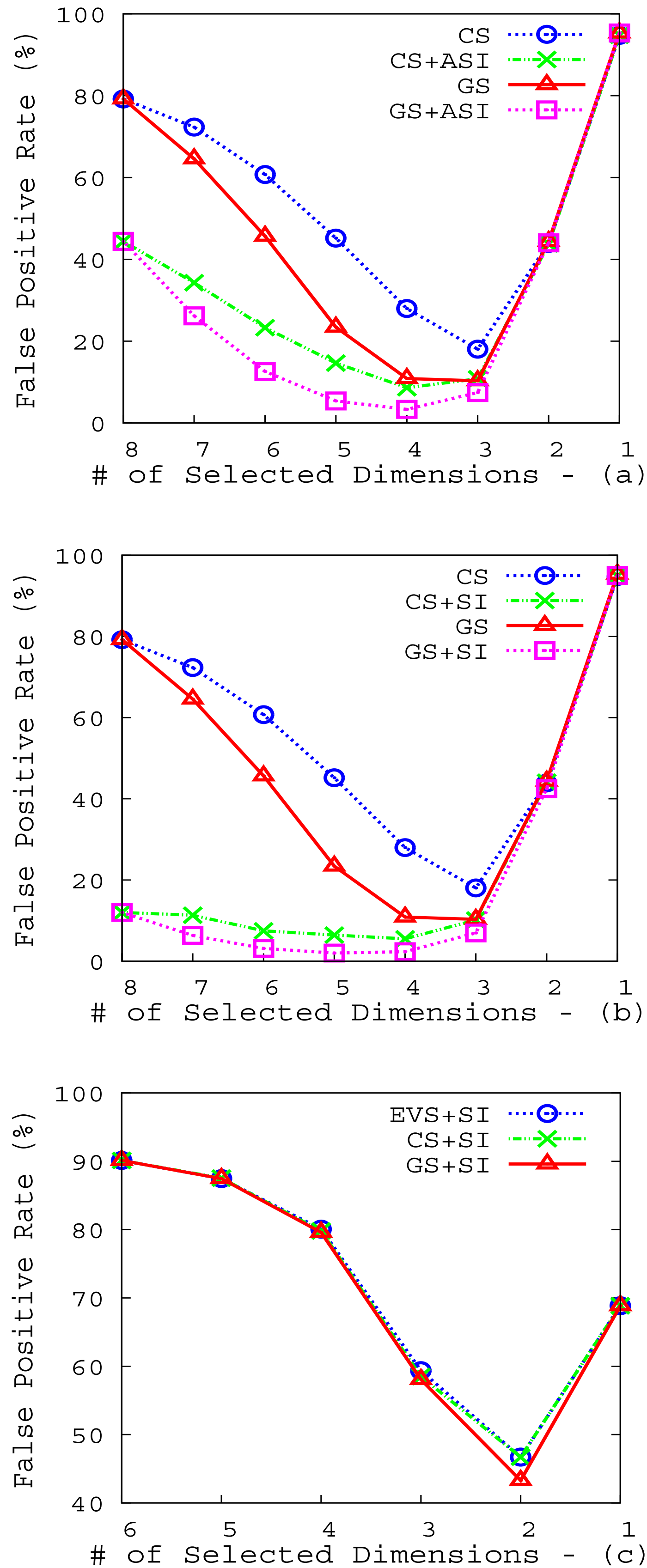


Figure 3.13: Performance Evaluations: Combined Approaches

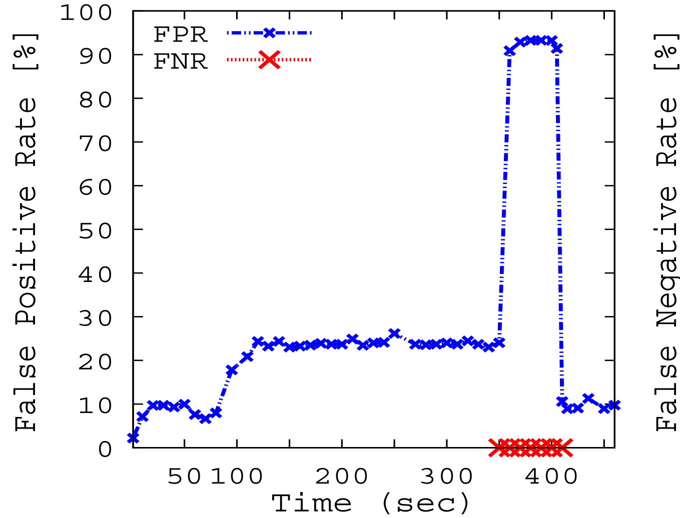


Figure 3.14: Performance Evaluations: Handling Dynamics

replaced by filters representing the entire event space. However, around the 410th second, the deployment of the new flows is complete, and the false positive rate goes down significantly as indexing is now according to the current traffic distribution.

3.5.6 Discussion

Our evaluation results show the huge potential of the proposed algorithms in improving expressiveness of content filters installed on TCAM. When evaluated with different data distributions, in every scenario, both workload-based indexing approaches outperform regular spatial indexing in the context of bandwidth efficiency. The dimension selection algorithms, also, show promising results, where the evaluation-based techniques, i.e., BRS and GS, reduce most number of false positives in the system, followed by CS, EMCS, and EVS. Of course all these algorithms perform significantly better than a random selection approach. While the performance w.r.t. bandwidth efficiency of the algorithms decreases in the aforementioned order, the runtime overhead associated with their computation increases in the reverse order. Our evaluations show that, with increasing number of events or subscriptions or original dimensions, always EVS has the least runtime overhead (in the order of milliseconds) which is closely followed by EMCS and CS. As expected, the evaluation-based techniques, GS and BRS have higher runtime overhead (in the order of seconds) with BRS being the most computationally intensive algorithm. Moreover, our results show that false positive rate in PLEROMA can be further reduced by up to ∽97% on combining the workload-based indexing approaches with dimension selection algorithms. Finally, our evaluation results confirm that, on employing the light-weight approach during transition from one network state to another, no false negatives occur in the system.

3.6 Related Work

The past decade has seen a significant amount of effort being devoted to the realization of scalable and bandwidth-efficient publish/subscribe systems [CRW01, VRKS06, RLW+02b, BCM+99, MPP15, PB02]. The primary focus of most of these systems has been efficient communication that not only ensures scalability, but also preserves expressiveness of content in order to avoid unnecessary traffic in the system. A very widely used technique employed to reduce false positives in overlay networks is subscription clustering where events are flooded within clusters [RLW+02b, BCM+99, CMTV07, PRGK09]. Riabov et al. perform clustering for content-based pub/sub systems by grouping subscribers into multicast channels and performing IP multicast thereafter [RLW+02b]. However, this approach largely depends on the similarity of subscriptions within generated clusters and may fail to ensure minimal false positives as multicasting is employed eventually within a cluster. Please note that, in this chapter, we combine the concept of subscription clustering (essentially an overlay-level mechanism) with in-network filtering on a software-defined network to avoid unnecessary traffic. Such a combination largely preserves expressiveness of a content-based subscription model.

Another technique based on workload and most often used for load balancing in overlays is content space partitioning. Content space partitioning is a much researched topic in various fields of computer science [Van91, WQA+04, CS04]. Although these techniques are not directly applicable in content-based routing using SDN, the notion can be used for workload-based indexing.

Linearizing content space (e.g. hashes, bit strings, etc.) for fast matching of events with subscriptions while balancing the load in structured P2P overlay network has been much researched in the past [GSAA04, AT06, BMVV05, MJ14]. Most of these works are based on distributed hash tables (DHT) that are load-balanced and self-organizing. Baldoni et al. in [BMVV05], realize a Chord-based [SMK+01] publish/subscribe where events and subscriptions are mapped to bit strings. Where on one hand, [BMVV05] maps subscriptions and events to multiple nodes, on the other hand, Muthusamy et al. in [MJ14] design a protocol that primarily indexes subscriptions at a single node. Of course, linearizing content space is also of extreme relevance to content-based routing in software-defined networks. However, in order to directly employ the above techniques, the SDN-compliant switches would have to support far more expressive operations. As a result, none of these linearizing techniques can be directly deployed in an SDN-based pub/sub system.

While attempting efficient content-based routing, considerable work has been dedicated to subscription summarization techniques that compact subscription information. With regards to this, various data structures and matching algorithms have been developed. For example, Jerzak et al., in [JF08], use Bloom filters [Blo70] to encode subscriptions and events. While this expedites content-based routing, it suffers from

the inherent limitations of a Bloom filter w.r.t. presence of considerable amount of false positives in the system. Again, the system MICS [JMVM09] uses Hilbert space filling curve to generate a one-dimensional representation of events and subscriptions. However, MICS too suffers from false positives in the system.

While dealing with data plane consistency in SDN, we come across a considerable amount of work in literature [RFR+12, JLG+14, MW13, KDR15]. For example, Reitblatt et al. [RFR+12] propose the method of versioning, which allows both the old as well as the new network states to be installed in the network simultaneously with different version numbers. A packet with one of the two version numbers is forwarded by the old or the new network state, depending on its version number, but is never forwarded by a mixture of both. However, this implies that each switch will require almost double the number of flows to accommodate both the old as well as the new version. Of course, this is a very resource-intensive method, and, as a result, we provide a more resource-efficient light-weight approach in this chapter.

3.7 Conclusion

In this chapter, we attempt to mitigate the limitations of an SDN-based publish/subscribe middleware w.r.t. expressiveness of content filters. We present a series of algorithms that improve the bandwidth efficiency of such a system and provide extensive evaluation results to analyze their behavior. The workload-based indexing approaches and dimension selection techniques complement each other and can build on top of each other to considerably impact unnecessary traffic in the middleware. Our evaluation results show that these strategies can significantly reduce false positive rate in the system (up to 97%) when subjected to various kinds of workload. Each of these algorithms preserve the benefits of using SDN for pub/sub by ensuring line-rate forwarding of events directly on switches while also preserving the benefits of content-based routing by focusing on bandwidth-efficient communication. Moreover, we present a light-weight approach to ensure data plane consistency in the presence of significant amount of network updates that are necessary in order to preserve the effectiveness of the algorithms introduced for expressive filtering of content.

Chapter 4

Expressive Filtering by Combining Application Layer

The algorithms presented in the previous chapter (cf. Chapter 3) significantly reduce false positives in the system. However, these solutions, also, come with significant overhead of deploying a completely different set of flows (due to change in the mapping of content to dz strings) in the data plane with each content mapping decision. Especially, with rapidly changing event distribution, the algorithms may need to be executed frequently as the previous decisions would not be as effective. As a result, in this chapter, we present a complementary method to reduce false positives in the system which involves making relatively fewer changes to the data plane. Please note that the methods presented in this chapter can, also, be employed together with the algorithms presented in the previous chapter (i.e., Chapter 3) to further enhance bandwidth efficiency of the system.

So, in this chapter too, we attempt to reduce bandwidth usage in the network but approach the target without considering any changes to the content mapping process. In fact, in this chapter, in order to increase bandwidth efficiency of PLEROMA, we attempt to involve traditional filtering methods at overlays while carefully considering their advantages and disadvantages. Where on one hand event forwarding in overlay networks provides possibilities of accurate filtering but suffers in terms of responsiveness to event delivery, on the other hand an SDN-based middleware provides line-rate performance but suffers in terms of bandwidth efficiency. So, while considering these two state-of-the-art implementations independently, we are tempted to ask the question, can we do any better? Is it possible to make these two radically different filtering approaches meet in the middle? And this is where we attempt to combine the benefits of both application layer filtering and network layer filtering in realizing a content-based pub/sub middleware that provides hybrid filtering of events. Therefore, in this chapter, we focus on designing an SDN-based pub/sub that not only aims at line-rate

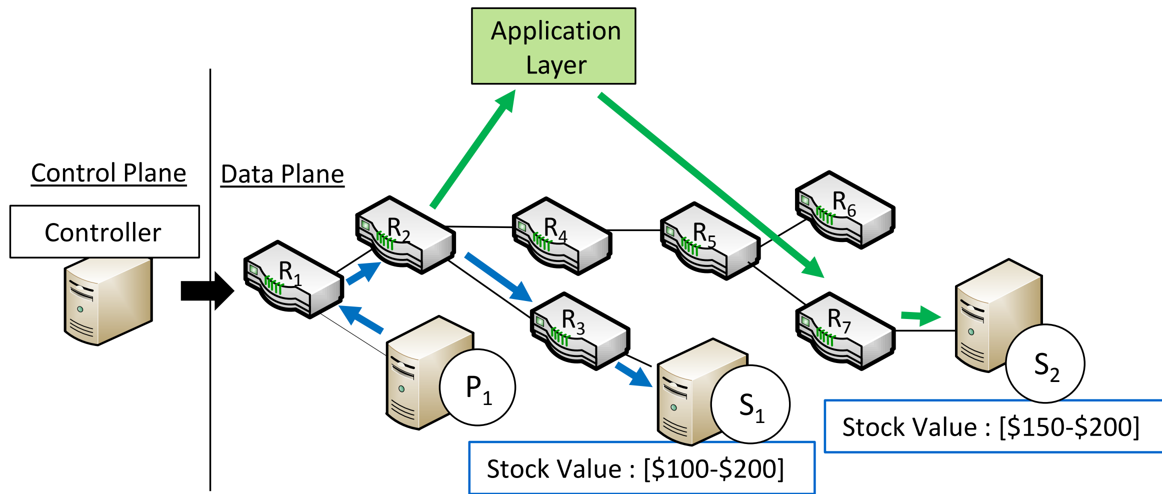


Figure 4.1: Hybrid Content-based Routing

performance but also bandwidth efficiency by providing a mechanism to filter events both in software (application layer) and on hardware (network layer). We provide selection mechanisms to determine the layer in which each event gets filtered in order to minimize unnecessary traffic in the network while also considering latency requirements of the middleware. Our hybrid approach offers complete flexibility to control the amount of filtering to be performed at each of the layers where the two extreme cases are pure software filtering and pure hardware filtering, thereby providing a complete degree of freedom to select the performance of the system in terms of latency and bandwidth efficiency. In summary, the contributions of this chapter are the design, implementation, and detailed performance evaluation of a hybrid SDN-based pub/sub middleware, the first of its kind, that provides event selection techniques to enable filtering of events both in the application layer as well as in the network layer in a latency and bandwidth-efficient manner.

4.1 System Architecture

The architecture of the designed hybrid middleware is very similar to the PLEROMA middleware except that it has an additional component in the form of the *Application Layer*. Figure 4.1 illustrates a hybrid approach to content-based filtering where events are filtered both in the network as well as in the application layer. We realize the application layer in our middleware as a pub/sub cloud service similar to BlueDove [LYK+11]. We perform multi-dimensional subscription space partitioning and distribute them among multiple servers (or matchers) that parallelize event filtering. Let us understand how events can be filtered at both layers with an example depicted in Figure 4.1. In Figure 4.1, the filters for subscriber S_1 are completely installed on the

network layer, whereas the filter corresponding to the subscription of S_2 at R_2 sends all events matching this filter to the application layer which enables accurate filtering. Only matched events are then injected back to the network at R_7 and forwarded to S_2, resulting in no false positives for this subscriber and subsequently no false positives along the path from R_2 to S_2 in the network. However, there may be false positives along the path between P_1 and S_1 depending on the expressiveness of the filters for S_1 on the switches. Where on one hand, application layer filtering has a distinct advantage over network layer filtering, in terms of reduced false positives, it loses out in matters of end-to-end latency/delay incurred for the delivery of events. Forwarding of events to S_1 occurs at line-rate, whereas that to S_2 is delayed due to filtering in software. Thus, there is a trade-off between reduction in false positives and end-to-end latency in the network as the improvement in one adversely affects the other.

4.2 Filter Selection Problem

Due to the aforementioned trade-off between end-to-end latency and bandwidth efficiency, the selection of filters that forward events to the application layer is very crucial. In fact, the main problem that we tackle in our hybrid approach to filtering is the selection of filters in the network layer that forward events to the application layer for more accurate filtering in the attempt to reduce the overall false positives in the network. However, we do so while ensuring that the average end-to-end latency of events in the system stays within the application-specified threshold. More formally, let F be the set of all filters on all switches in the network, where $f_i \in F$. Also, let rfp_i be the number of false positives reduced in the system if filter f_i is chosen to send matched events for further filtering in the application layer. Let $\mathbb{S}$ be the set of all subscribers in the system, where $S_k \in \mathbb{S}$. Again, let δ_k be the average end-to-end latency at subscriber S_k. Finally, let Δ be the average end-to-end latency threshold to be maintained in the system.

Our objective is to determine the subset $SF \in F$ that forwards events to the application layer such that the combined effect of the filters in SF results in maximum reduction of false positives in the network while staying within a given average end-to-end latency threshold, i.e.,

$$Maximize \sum_{i \in SF} rfp_i$$

$$subject\ to\ (\sum_{k=1}^{|\mathbb{S}|} \delta_k)/|\mathbb{S}| \leq \Delta$$

This is an optimization problem. Let there be a total of m filters on n switches constituting the network, where m ranges from 0 to $\sum_{j=1}^{|\mathbb{S}|} |DZ|_j$. Then, to arrive at the optimal solution, all combinations of filters, i.e., 2^m possible subsets SF have to

be calculated and considered. Also, it should be noted that the value of m can be in the order of hundreds of thousands, making the optimal solution impractical and not scalable in a realistically large network.

The above problem may look seemingly like the Knapsack Problem [KPP04] where the value (i.e., benefit) of each item in the Knapsack Problem may be compared to the false positives reduced by each filter and the weight (i.e., penalty) of each item may be compared to the increase in average end-to-end latency on selecting a filter. The goal in the Knapsack Problem is to maximize the total value of items selected for the knapsack while the total weight of the knapsack remains within a given threshold. This is similar to our problem where the goal is to maximize the total false positives reduced by selected filters while the total delay penalty incurred by them remains within a given threshold.

However, there is a major difference between the two problems that sets them apart. If an item gets selected for the knapsack, this selection has no influence on the values and weights of the remaining items to be considered for selection. This is where our optimization problem differs. In our optimization problem, if a filter gets selected, this may influence the false positives reduced by the remaining filters and the increase in average end-to-end latency on selecting each of these filters. Due to this significant difference, approaches for solving the Knapsack Problem cannot be directly employed to our problem. As a result, in this thesis, we propose two selection algorithms with varying degrees of complexity and benefits, in terms of bandwidth efficiency, to solve the filter selection problem. Till now, we have made the assumption that the false positives reduced and the increase in end-to-end latency on selection of every individual filter are already known. However, the process of determining these values is not straightforward.

So, to arrive at a scalable solution to our optimization problem, we have to tackle three subproblems – (i) detect false positives due to each filter on each link of the network such that rfp_i for each filter f_i, i.e., benefit, can be determined (Section 4.3.1), (ii) determine the increase in the average end-to-end latency of the system, i.e., penalty, on selecting each filter (Section 4.3.2), and (iii) with the knowledge of these calculated benefits and penalties for all filters in the network, design efficient filter selection algorithms (Section 4.4).

4.3 Filter Benefit and Penalty Calculation

In this section, we provide the means of calculating benefit and penalty associated with the selection of each filter. These metrics form the basis of the filter selection algorithms.

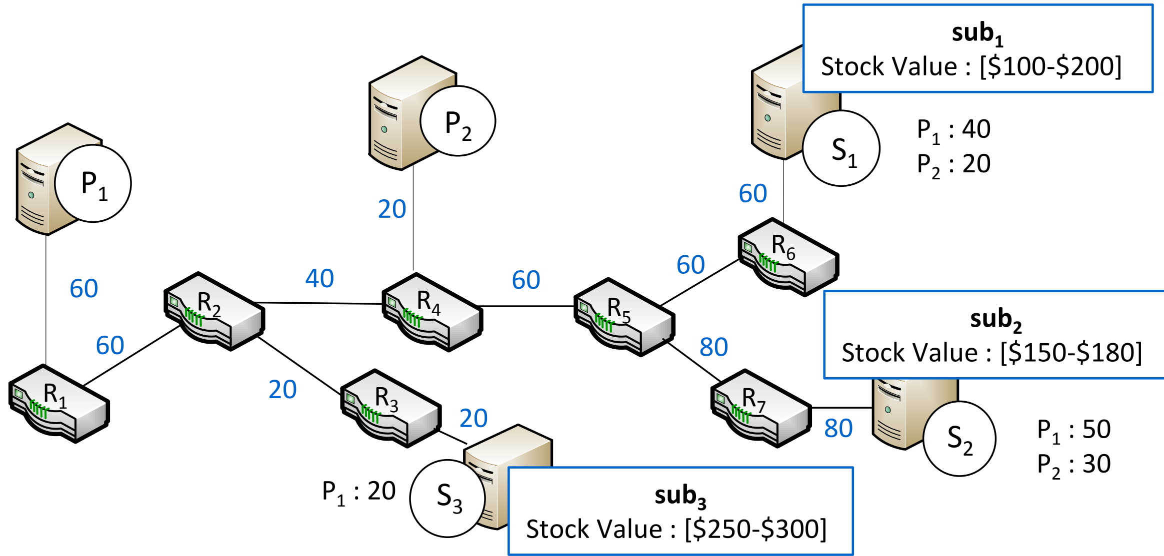


Figure 4.2: False Positive Detection

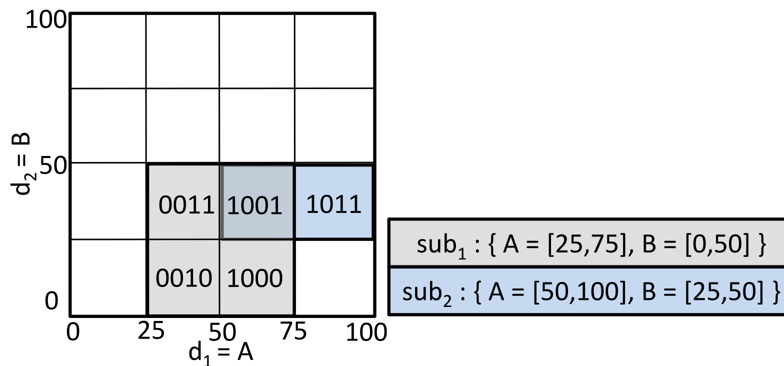


Figure 4.3: Partial Overlap

4.3.1 Benefit

To determine the false positives reduced on selection of each filter, it is imperative to first calculate false positives on each link of the network due to each filter. To do so, each subscriber needs to periodically send the false positives received by it from all its associated publishers to the controller such that the controller can determine false positives along each path between publishers and subscribers. More specifically, the controller calculates false positives on each link of the network for each filter by back-tracking on the paths from subscribers to publishers while aggregating false positives on links. However, containment relations between subscriptions, i.e., (i) disjoint, (ii) complete overlap, and (iii) partial overlap, need to be considered during aggregation. The method to handle these three cases is described as follows.

(i) In the scenario where false positives of two subscriptions that are completely disjoint are disseminated over a link, the aggregation over this link will be a sum of the false positives of both subscribers.

(ii) However, if one of the subscriptions is contained by the other or both are equal, then a simple sum will account for more than the actual false positive count over the link. In this case, false positives for the broader subscription (or one of the subscriptions in case of equality) should only be considered over the link as all other false positives are either already accounted for in the broader subscription or are events that should be forwarded along this link as they match the broader subscription.

This can be further understood with an example in Figure 4.2 where subscribers S_1 and S_2 subscribe for sub_1 and sub_2, respectively, and $sub_1 \succ sub_2$. Let us start the backtracking process from S_1 by aggregating the false positives along the paths to publishers P_1 and P_2. This is straightforward, until we reach switch R_5 as R_5 also forwards false positives to S_2 which might need to be aggregated for the link between R_4 and R_5. However, since $sub_1 \succ sub_2$, only the false positives of the broader subscription sub_1 will be considered for this link.

(iii) In the case of a partial overlap between two subscriptions, we mainly identify 3 subspaces, i.e., the overlapping subspace and the 2 disjoint subspaces. Now, if false positives for the overlapping parts and the disjoint parts for each subscription can be identified, then the aforementioned mechanisms can be employed to detect false positives on network links.

For example, Figure 4.3 depicts two subscriptions sub_1 and sub_2 with a partial overlap. Now, if the two subscribers divide the subscriptions into subspaces of finer granularity (as depicted in the figure) and locally detect false positives corresponding to each subspace, then the controller would have to only deal with complete overlaps and disjoint relations. So, while calculating false positives over a link delivering events to sub_1 and sub_2, for the subspace $\{1001\}$ (i.e., overlapping subspace), false positives for only one subscription are counted. Again, the false positives for the other disjoint subspaces can be simply aggregated. Of course, here, there is a trade-off between the accuracy of the false positive count and the granularity at which detection occurs at the subscriber. Finer the granularity, greater is the accuracy as well as the overhead of management at both the subscriber and the controller. Analyzing this trade-off has been the subject of previous research [JMVM09] and is not the focus of this thesis. Instead, we focus on the challenging issue of performing efficient hybrid content-based filtering.

With the knowledge of the number of false positives on each link of the network due to each filter, we can calculate the benefit of a filter, i.e., the false positives reduced by it in the network, by aggregating all false positives forwarded by it along its downstream paths.

4.3.2 Penalty

The delay penalty incurred by a filter on its selection primarily deals with the number of paths between publishers and subscribers along which it forwards events. On its selection, a filter, say f_i, will forward events to the application layer, increasing the end-to-end latency for these events along all paths that f_i is associated to. This means that while calculating the new average end-to-end latency for the system on selection of f_i, the specific network delays along each path that f_i affects have to be replaced with application delays. Naturally, there is an increase in the average end-to-end latency and this increase is the calculated penalty for f_i.

For the sake of simplicity and without loss of generality, to explain our selection algorithms, we represent penalty in terms of the number of affected paths between publishers and subscribers as this number directly affects the average end-to-end latency. Let us assume that the average end-to-end latency of events in the network is N_d, average end-to-end latency of events when the application layer is involved is A_d, and total number of paths between publishers and subscribers in the network is TP. Now, if x is the number of paths involving application layer filtering, then the average end-to-end latency in the network is $[(x * A_d) + ((TP - x) * N_d)]/TP$. Also, the calculation of Δ (i.e., average end-to-end latency threshold) in terms of the maximum number of paths that can be allowed to be affected by application layer filtering delay, say AP_{Th}, follows directly from the previous formulation, and can be calculated as $AP_{Th} = (TP * (\Delta - N_d))/(A_d - N_d)$. Note that we calculate penalty and the penalty threshold (i.e., average end-to-end latency threshold) here w.r.t. average delays and represent them as affected paths for the sake of better understanding of the following sections. However, in reality, while calculating the penalty, our system can consider the exact network delay incurred along each path between publishers and subscribers and can calculate penalty as the exact increase in average end-to-end latency of the system as described earlier.

4.4 Selection Algorithms

After determining the benefits and penalties for each individual filter in the network, we proceed to propose two selection algorithms—the Switch Selection Algorithm and the Cluster-based Selection Algorithm—that differ in time complexity as well as in quality w.r.t. reduction in false positives.

4.4.1 Switch Selection Algorithm

We can simplify our problem by selecting switches in place of filters that forward events to the application layer. However, even in this case, we must consider 2^n possible

Algorithm 4 Switch Selection Algorithm

1: $\mathbb{R} \leftarrow$ Set of all switches in the network
2: $AP_{Th} \leftarrow$ Penalty threshold
3: $SR = \emptyset$ // Set of selected switches
4: **while** $\mathbb{R} \neq \emptyset \mid\mid AP_{Th} \neq 0$ **do**
5: **for all** $R \in \mathbb{R}$ **do**
6: $benefit_R \leftarrow$ Aggregate benefits of flows on R // Total benefit of R
7: $penalty_R \leftarrow$ Aggregate penalties of flows on R // Total penalty of R
8: $noBenefit_SwitchSet = \{R \in \mathbb{R} : benefit_R = 0\}$
9: $AP_{Th}_exceeded_SwitchSet = \{R \in \mathbb{R} : penalty_R > AP_{Th}\}$
10: $\mathbb{R} = \mathbb{R} \setminus (AP_{Th}_exceeded_SwitchSet \cup noBenefit_SwitchSet)$
11: **if** $\mathbb{R} \neq \emptyset$ **then**
12: $selectedSwitch = \{R \in \mathbb{R} : benefit_R = maxBenefit(\mathbb{R})\}$
13: $SR = SR \cup selectedSwitch$
14: $\mathbb{R} = \mathbb{R} \setminus selectedSwitch$
15: $AP_{Th} = AP_{Th} - penalty_{selectedSwitch}$

subsets of the entire set of n switches in the network for an optimal solution w.r.t. switches. The question is, can we do something better to reduce this complexity? The main idea behind the Switch Selection Algorithm (SSA) is to iteratively select switches, such that the most beneficial switch gets selected in each iteration, till the given penalty threshold is reached and eventually obtain a subset of switches $SR \in \mathbb{R}$ that forward incoming events to the application layer.

We provide a detailed description of the steps of SSA as follows and a formal description in Algorithm 4. SSA starts by considering the set $\mathbb{R}$, consisting of all switches in the network, and calculates the benefit and penalty of each switch (cf. Algorithm 4, lines 5-7). The benefit and penalty of each switch is the aggregation of the benefits and penalties of all filters on it. Next, all switches with no benefit are removed from $\mathbb{R}$. Also, all switches whose penalty violates the average end-to-end latency (or available path) threshold are removed from further consideration for selection (cf. Algorithm 4, lines 8-10). From the remaining switches, the switch with the highest benefit within the penalty threshold is selected and added to the subset SR and removed from $\mathbb{R}$ (cf. Algorithm 4, lines 12-14).

If the penalty threshold has not been reached, then all remaining switches in $\mathbb{R}$ are again considered for the next cycle. Please recall that the selection of a switch for application layer filtering may change the number of false positives reduced and the additional delay incurred by other switches due to the filters on the selected switch. Assume, a switch R_i having a filter f_i, is added to SR. Now, for another switch R_j $\in \mathbb{R}$ having a filter f_j, the benefit and penalty additionally offered by f_j need to be recalculated. Otherwise, the same false positives already reduced by f_i will again be considered for f_j. Also, the same path already considered for application layer delay may again be counted in the delay penalty for f_j. So, after determining the benefit and penalty for f_i, all false positives for filters corresponding to f_i on subsequent switches along the downstream paths of f_i must be set to zero and the paths marked as already

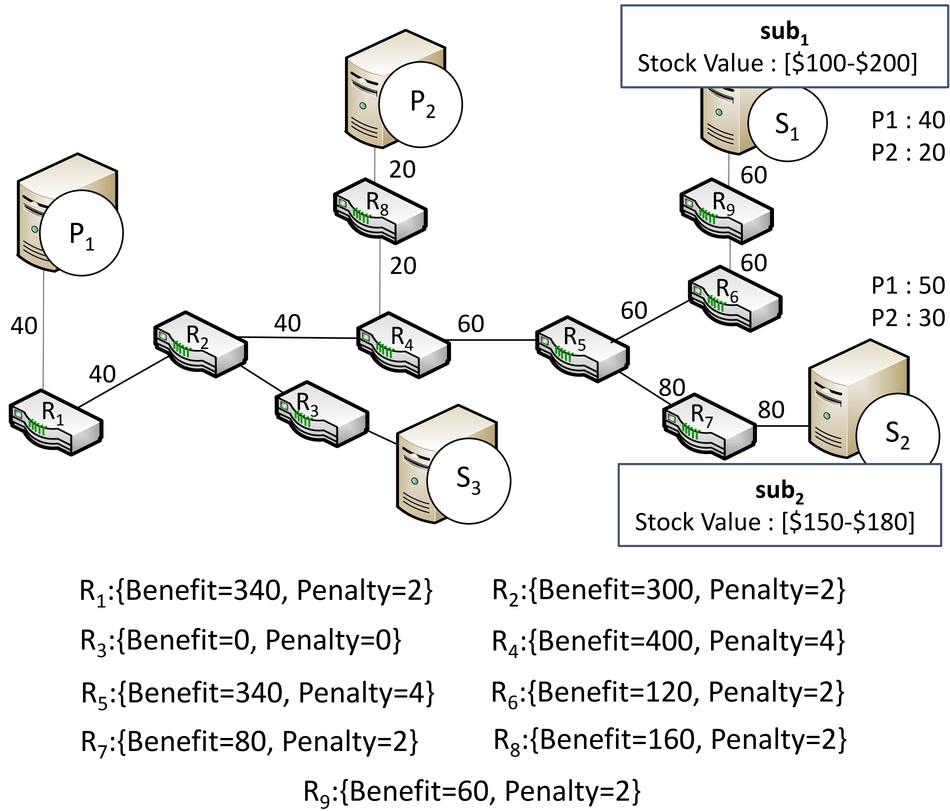


Figure 4.4: Switch Selection

considered. Now, while calculating the benefit and penalty of f_j, none of the false positives and paths already considered for f_i will be reconsidered. So, for each cycle, the benefits and penalties of all filters on the remaining switches in $\mathbb{R}$ are recalculated based on the filters on switches in SR. The cycles continue until the penalty threshold is reached, or would be potentially exceeded with any further selection, or if $\mathbb{R}$ is empty.

To further explain the aforementioned algorithm, we use an example from Figure 4.4, where the initially calculated benefits and penalties for each switch $\in \mathbb{R}$ are depicted. Let us assume that the average latency threshold Δ when mapped to the affected paths threshold AP_{Th} (cf. Section 4.2) has the value 3. Of all the switches in $\mathbb{R}$, R_4 and R_5 get removed as they violate the threshold. Also, R_3 can be removed as it has 0 benefit. According to the algorithm, R_1 gets selected as it has maximum benefit within the given threshold. As a result, at the end of this cycle, $\mathbb{R}$=$\{R_2, R_6, R_7, R_8, R_9\}$ and SR=$\{R_1\}$. Since, AP_{Th} has not been reached yet, another cycle will commence. Now, since R_1 has already been selected, it will send all events received by it to the application layer, resulting in no or fewer false positives on its downstream paths. As a result, the benefits and penalties of the remaining switches need to be recalculated. In this case, the recalculated benefits and penalties for R_2 are 0 and 0, R_6 are 40 and

Algorithm 5 Cluster-based Selection Algorithm

```
1:  ℝ ← Set of all switches in the network
2:  SB ← Set of all subscriptions
3:  AP_Th ← Penalty threshold
4:  ℂ𝕃 = clusterFilters(SB, ℝ) // Create a set of dissemination trees
5:  SRCl = ∅ // Selected set of switch-filter_clusters
6:  while AP_Th ≠ 0 do
7:     RCl = ∅ // Switch-filter_cluster set refreshed every cycle
8:     for all cl ∈ ℂ𝕃 do
9:        R_cl ← Select switch in cl with highest benefit of filters within penalty threshold
          // cf. Algorithm 4
10:       RCl = RCl ∪ R_cl
11:    if RCl ≠ ∅ then
12:       SRCl = KnapsackSolution(RCl, AP_Th)
13:       updateClusters(ℂ𝕃, SRCl) // Remove already selected switch-filter_clusters in
          ℂ𝕃 from further consideration
14:       AP_Th = AP_Th − Σ_{R_cl∈SRCl} penalty_{R_cl}
15:    else
16:       AP_Th = 0
```

1, R_7 are 30 and 1, R_8 are 160 and 2, and R_9 are 20 and 1, respectively. As per the algorithm, in this cycle, R_8 gets removed from further consideration as it violates the threshold, while R_6 gets selected and added to SR. Finally, since the penalty threshold of 3 is reached, the algorithm terminates with the final set of selected filters returned as SR=$\{R_1, R_6\}$. The switch selection algorithm has a complexity of O(n^2).

4.4.2 Cluster-based Selection Algorithm

In SSA, we selected switches instead of filters as a solution considering individual filters is impractical. However, a solution which is in the middle of these two, would be interesting to analyze. As a result, in our next algorithm called Cluster-based Selection Algorithm (CSA), we select filters rather than switches but this time we consider a group of filters based on the subscriptions they represent. We provide a detailed description of the steps of CSA as follows and a formal description in Algorithm 5. In CSA, first, we cluster all binary filters (representing subscriptions on the network layer) based on their similarity into spatially disjoint groups (cf. Algorithm 5, line 4). There are many subscription clustering techniques proposed in literature and any one of them may be selected for the clustering of filters [RLW+02b, Gut84]. Since the filter clusters (i.e., $\mathbb{CL}$) are spatially disjoint, each cluster disseminates a disjoint set of events in the network, thus giving the notion of separate event dissemination trees embedded in the network for each cluster. Therefore, in the following description, we consider each cluster to have its own dissemination tree disjoint from those of other clusters such that an event gets disseminated along only a single cluster's tree and can only affect the false positive count along the links of this tree. For example, Figure 4.5

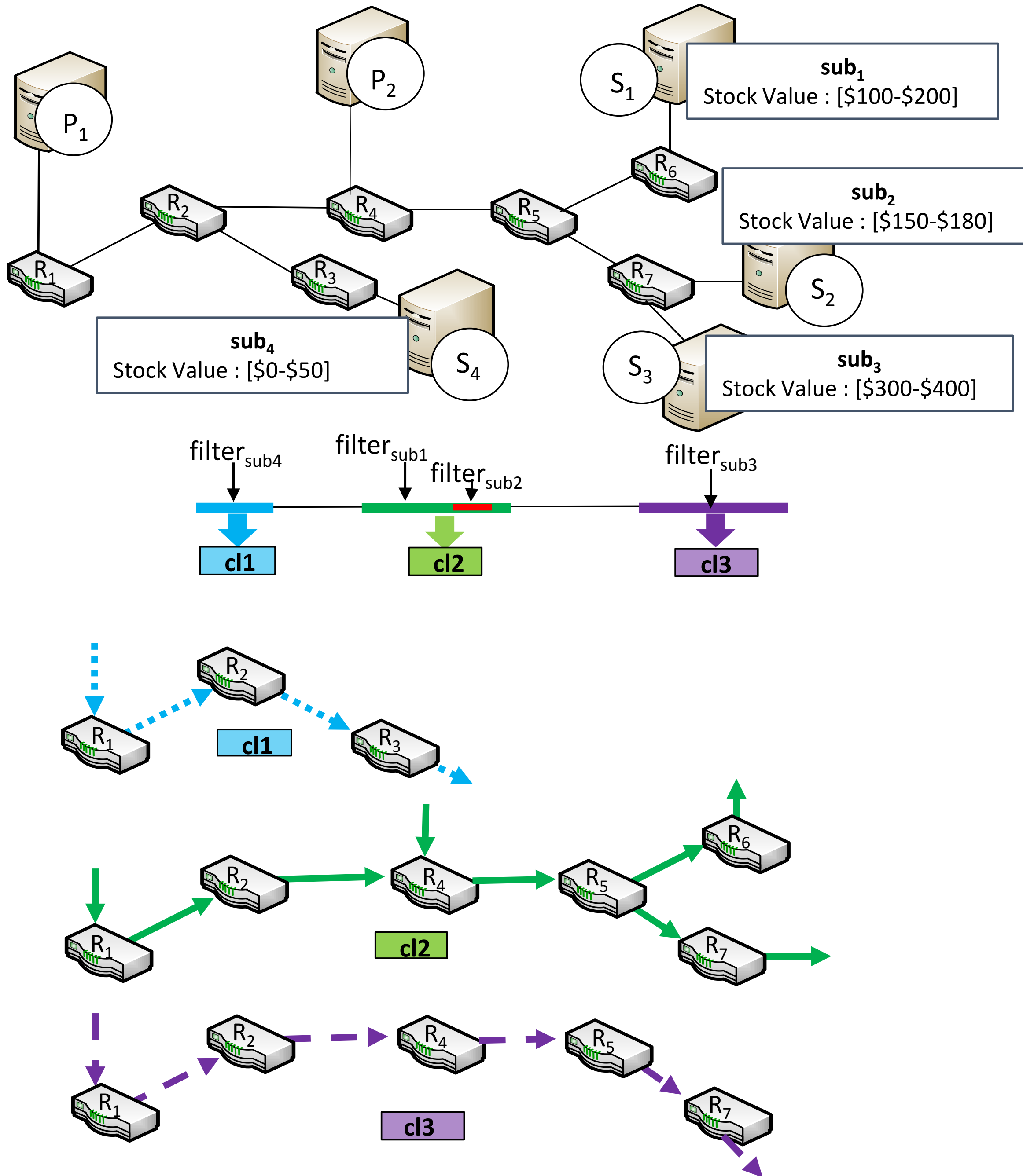


Figure 4.5: Cluster-based Selection

illustrates a scenario where 4 subscribers S_1, S_2, S_3, and S_4 subscribe for sub_1, sub_2, sub_3, and sub_4, respectively, where the containment relations between subscriptions and consequently the filters they represent are depicted. Here, we consider a very simple case with 3 clusters, cl_1, cl_2, and cl_3, that are disjoint in space as can be seen in the figure. Also, there are three dissemination trees embedded in the network for each of these clusters.

After clustering of filters, in each cluster $\in \mathbb{CL}$, we identify the switch with maximum benefit within the penalty threshold. So, if a switch R_i gets selected in the cluster cl_j, we represent this switch-filter_cluster as $R_i_cl_j$. This process of identifying the most beneficial switch within each cluster is identical to calculating the benefits and penalties of each switch in $\mathbb{R}$ and selecting the most beneficial and feasible switch as discussed in details in Section 4.4.1. In this approach, all filters of a switch do not get selected but only a filter set representing a cluster on the switch gets selected for application layer filtering. As a result, we get a total of $|\mathbb{CL}|$ switches from all the clusters and add them to a switch-filter_cluster set RCl (cf. Algorithm 5, lines 8-10). Let R_1, R_4, and R_1 be selected in cl_1, cl_2, and cl_3, respectively such that $RCl = \{R_1_cl_1, R_4_cl_2, R_1_cl_3\}$. Note that even though the same switch R_1 gets selected for two clusters, the switch-filter_cluster makes each pair unique. Now, we try to find the subset $SRCl \in RCl$ that maximizes the combined reduction of false positives in the network due to all selected switch-filter_cluster pairs $\in SRCl$ while ensuring the average end-to-end latency of the system within Δ.

If all combinations of switch-filter_clusters are considered for the solution, then the complexity is $O(2^{|\mathbb{CL}|})$. It should be noted that unlike our original optimization problem, selection of a particular switch-filter_cluster pair for forwarding events to the application layer does not affect the reduction in false positive count and the delay penalty of the other switch-filter_cluster pairs as the clusters are disjoint. So, no recalculation of benefit and penalty need to be done at the switches. This problem can now be solved by directly mapping it to the Knapsack Problem.

The aforementioned steps produce the subset $SRCl$ that maximizes reduction in false positives while staying within Δ. However, if the threshold has not yet been reached, then the entire cycle has to be repeated. In the next cycle, again the benefits and penalties of switches have to be recalculated for clusters that are part of $SRCl$. This is because the selection of a switch from a cluster will affect the benefits and penalties of switches within the same cluster. Based on the new values, again, $|\mathbb{CL}|$ switch-filter_clusters are selected from all the clusters, and the cycle progresses as before with a new set RCl (cf. Algorithm 5, lines 6-16). The cycles continue till the threshold is reached or would be potentially exceeded with any further selection. As a result, CSA has a complexity of $O(n^2 + n * 2^{|\mathbb{CL}|})$.

4.4.3 Network Updates

Once the filters for forwarding events to the application layer are selected, the controller makes the necessary changes to the network by modifying the action field of each flow representing the selected filters. As a result, all events that match these filters get forwarded to the application layer as dictated by the action field of the flows. Clearly, the event distribution and the current subscriptions in the system might change over time degrading the performance of our deployed solution to forward events to the application layer. So, in order to adapt to changes, the controller periodically collects information about the false positives (in the recent time window) from the subscribers, recalculates the most beneficial set of filters, and deploys the changes in the network.

4.5 Further Optimizations

The complexity of both the proposed algorithms depend on the number of switches in the network, i.e., n. So, these algorithms may be further optimized if we reduce the search space, i.e., reduce the number of switches on which they operate. In fact, we identify those switches that would add value to the solution and will be candidates for the desired solution while neglecting all other switches. A switch is selected as a candidate if no other switch in the network reduces more false positives than this one for the same set of paths that this switch affects. In doing so, we identify 3 types of switches as candidates for selection—a leaf switch connected directly to a publisher, a switch with two or more ingress ports, a switch connected directly to a switch with two or more egress ports. All other switches in the network may be ignored. The reason why these switches are the only ones that make a difference to the solution is because, due to their ingress ports, they are the starting points of new combinations of paths and, therefore, will always reduce the most false positives on these path combinations. This can be understood in the following example depicted in Figure 4.4.

Figure 4.4 shows false positives on each link of the network when two subscribers S_1 and S_2 subscribe and two publishers P_1 and P_2 publish events. Let us focus on switch R_1 which is directly connected to P_1. If R_1 is selected to forward events for further filtering, the number of false positives it will decrease in the network is 340. Also, selection of R_1 introduces application filtering delay along two paths, i.e., between P_1—S_1 and P_1—S_2. For the same two paths, we need to check if any other switch reduces more false positives. In fact, R_2 reduces 300 false positives while incurring a delay penalty of 2 for the same two paths. As R_1 is the starting point of the path combination consisting of these two paths, it will always reduce more false positives than R_2 for the same penalty. As a result, R_1 gets selected as a candidate while switches like R_2, which are guaranteed to have less benefit for the same penalty, can safely be ignored for further consideration as they will never be a part of the desired solution. A switch like R_4 with two or more ingress ports, however, must be considered as it

is a switch where multiple paths join. As a result, for this new combination of paths (P_1—S_1, P_1—S_2, P_2—S_1, and P_2—S_2), this switch will always have the most benefit as it is the starting point of this path combination in the switch network. So, a switch like R_5, with two or more egress ports makes no difference to the solution as its benefit is less than R_4, while it affects the same paths as R_4. Again, a switch like R_6 that is directly connected to a switch with two or more egress ports that splits paths, poses as the source of a new combination of paths, i.e., P_1—S_1, and P_2—S_1, and therefore must be considered for further processing. The example shows how the aforementioned three types of switches should only be considered for further processing without adversely affecting quality of the solution. Pruning the network has a complexity of $O(n)$ and can reduce the runtime of SSA and CSA without degrading their quality. Of course, the effectiveness of this optimization depends largely on the paths between publishers and subscribers.

4.6 Performance Evaluations

This section is dedicated to an analysis of the proposed hybrid pub/sub middleware and its comparison to purely network layer-based and purely application layer-based implementations. A series of experiments are conducted to understand the effects of the design on performance metrics such as end-to-end latency for event dissemination and bandwidth efficiency in terms of false positives disseminated in the network. We further compare the performances of the two proposed selection approaches, SSA and CSA, in terms of benefit and complexity.

4.6.1 Experimental setup

The following experiments have been evaluated under two test environments (cf. Chapter 2)—1) the SDN-testbed (*SDN-t-hswitch*) comprising a hardware Whitebox Openflow-enabled switch from Edge-Core and commodity PC hardware, and 2) emulated networks running on a single machine using Mininet (*SDN-m*). The latency-related experiments were conducted on *SDN-t-hswitch*. Besides the SDN-testbed, we used *SDN-m* to experiment with up to 337 switches and 729 end-hosts on different topologies.

We used a content-based schema that contains up to 6 attributes, where the domain of each attribute varies in the range [0,1023]. We use both real-world workload as well as synthetic workload to conduct our experiments. For synthetic data, we, again, use both uniform and zipfian distributions. Also, for real-world workload, we, again, use data in the form of stock quotes procured from Yahoo! Finance containing a stock's daily closing prices. Such real world data further highlights the performance and importance of the hybrid approach under realistic scenarios.

4.6.2 Comparing with State-of-the-Art

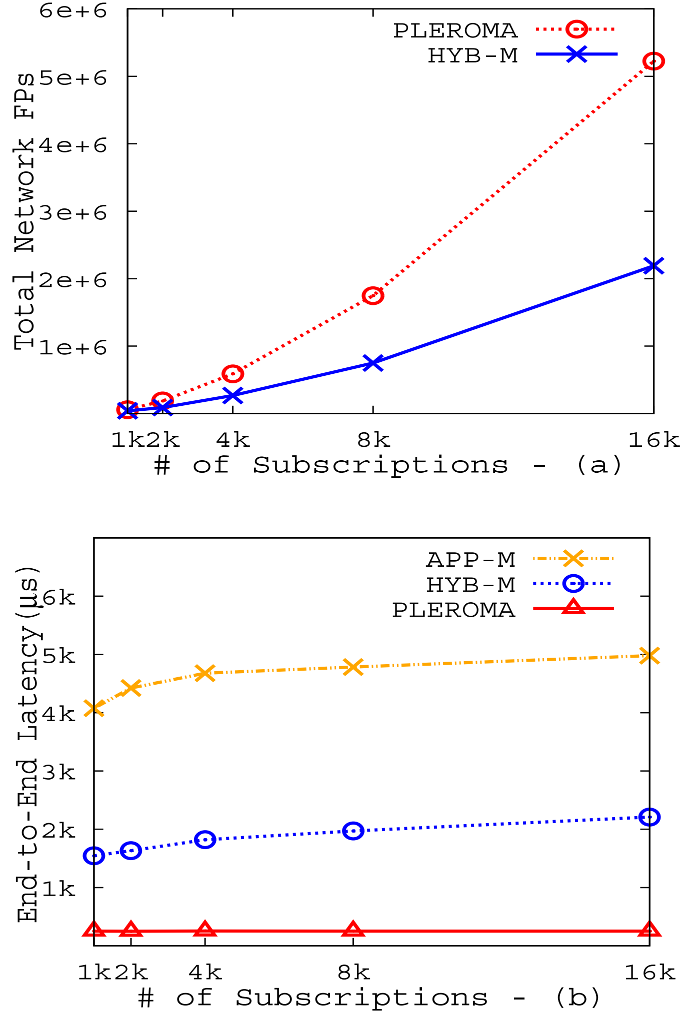


Figure 4.6: Performance Evaluations : Comparing with State-of-the-Art

The first set of experiments compares the performances of the hybrid middleware (*HYB-M*), a purely network layer-based middleware (PLEROMA), and a purely application layer-based middleware (*APP-M*). PLEROMA, of course, implements SDN-based in-network filtering. Moreover, please recall from Chapter 2 that we implemented the purely application layer-based middleware *APP-M* as a parallelized matching pub/sub service where we divided the event-space into 16 partitions and assigned them to 16 matchers running on 16 cores to enable one-hop forwarding of events as performed in Bluedove [LYK+11]. All measurements in the application layer have been performed corresponding to this configuration. Also, the application layer has been hosted on a 3.10 GHz machine with 40 cores.

Please recall that the performance of the hybrid middleware can be regulated by adjusting the value of Δ. In the following experiments we represent this threshold value in terms of a factor of the application layer filtering delay, such that a factor of 0 implies pure network filtering and a factor of 1 implies pure application layer filtering. Also, HYB-M uses SSA for switch selection such that we can compare the performance of a pessimistic hybrid approach with state-of-the-art solutions, rather than CSA which outperforms SSA w.r.t. reduction in false positives as can be seen later in this section.

Figure 4.6(a) depicts the performance of HYB-M and PLEROMA w.r.t. total false positives in the network, i.e., the sum of all false positives on all links, with increasing number of subscriptions when 10,000 events are disseminated. Since APP-M performs accurate filtering of events in software, we do not plot its performance in this graph. For a threshold factor of 0.6, the figure shows that the false positives for HYB-M are much less in every case than those for pure in-network filtering. Even though the hybrid middleware performs better as compared to PLEROMA in terms of bandwidth efficiency, it comes with a price. Figure 4.6(b) depicts the plots for average end-to-end latency with increasing subscriptions for all 3 systems. The figure shows that pure network-layer filtering has minimum latency in the order of a few microseconds. Also, the increase in number of subscriptions has no influence on latency. On the other hand, APP-M has the worst performance with latency, in the order of milliseconds, which increases with increasing number of subscriptions because, in software, more the number of subscriptions, more will be the time needed to match events. The figure shows that hybrid filtering results in latency less than that of APP-M but greater than that of PLEROMA, as a certain percentage of events are now affected by application layer filtering delay. However, both bandwidth efficiency and latency of the hybrid approach may be regulated by adjusting the threshold factor, i.e., Δ, which is clearly visible in the following set of experiments.

4.6.3 Impact of Threshold Factor

Figure 4.7(a) and Figure 4.7(b) show the effects on bandwidth efficiency and latency in a system with 8000 subscriptions when the threshold factor is increased from 0 to 1. In Figure 4.7(a), the term benefit signifies the percentage of false positives reduced by HYB-M w.r.t. the false positives occurring in PLEROMA. With the threshold factor set to 0, HYB-M has no benefit as it is comparable to pure in-network filtering. With increasing threshold factor, the benefit gradually increases. However, this also implies an increase in average end-to-end latency, as depicted in Figure 4.7(b). It should be noted that a factor of 1 is comparable to APP-M w.r.t. latency. However, even in this case the benefit will not be 100% as the false positives on the links between the publishers and the switches they are directly connected to will remain in the system.

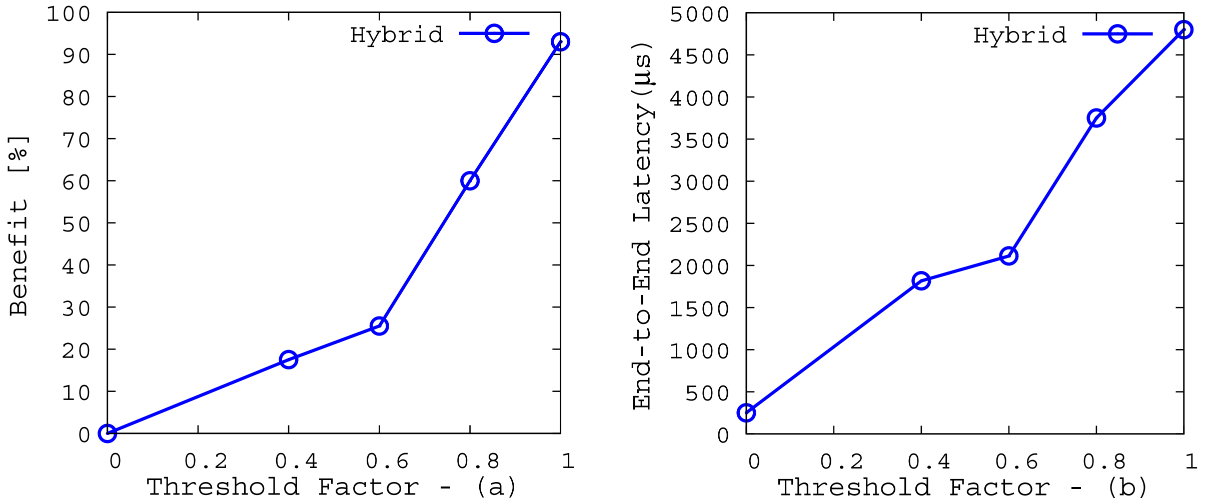


Figure 4.7: Performance Evaluations: Impact of Threshold Factor

4.6.4 SSA vs CSA

The next set of experiments evaluates and compares the performance, in terms of benefit and complexity, of the two proposed selection algorithms – SSA and CSA. Figure 4.8(a) depicts the benefit of SSA and CSA with increasing number of subscriptions. For these experiments, we used 16 clusters for CSA. The figure shows that, in each case, CSA has higher benefit than SSA. This is because CSA has higher flexibility w.r.t. selection of filters as it chooses groups of filters within a switch rather than all filters on it as is the case in SSA. We also conducted experiments to see the behavior of both methods when the threshold factor is gradually increased in the system. Figure 4.8(b) shows that with increasing threshold, the benefit increases in both approaches and CSA performs consistently better than SSA.

Please note that the performance of CSA largely depends on the number of clusters used by it and as a result our next set of experiments is conducted to analyze the effect of increasing clusters on CSA. Figure 4.8(c) clearly depicts that, with increasing number of clusters, the performance of CSA improves further as its flexibility of filter selection increases manifold. Of course, when a single cluster is used, CSA is essentially reduced to SSA. We conducted the above experiments using real-world workload which clearly highlights the bandwidth efficiency achievable by the hybrid middleware under realistic scenarios even for a threshold factor of just 0.4.

Even though CSA offers higher benefit than SSA, it loses out to SSA in terms of runtime overhead. We conducted experiments to compare the runtime of both algorithms. Figure 4.8(d) shows that with increasing number of subscriptions, the runtime of both approaches increases. This is because, higher the number of subscribers in the system, higher will be the number of paths and filters on switches to be considered, thus

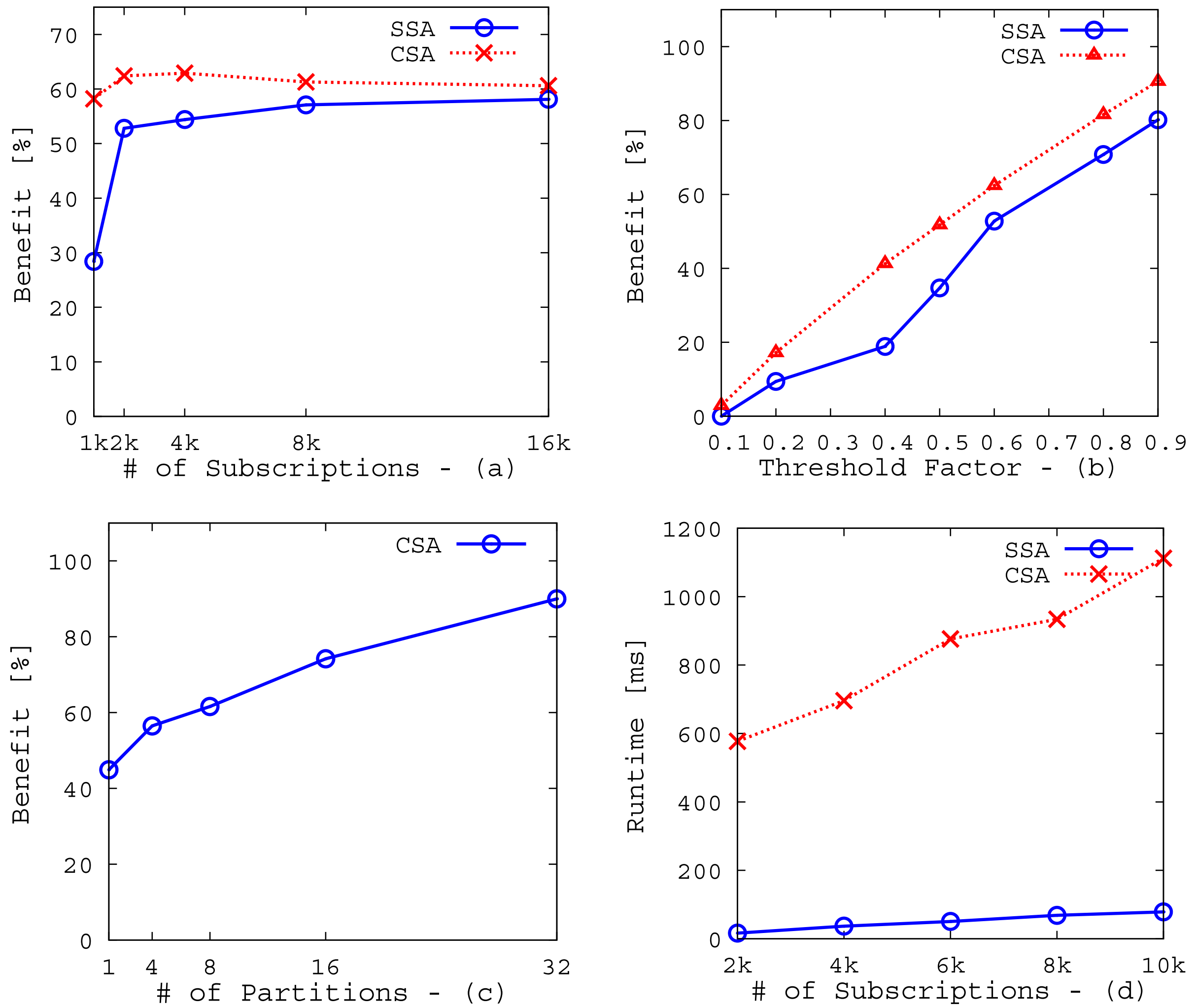


Figure 4.8: Performance Evaluations: SSA vs CSA

increasing the runtime. Also, the runtime of CSA is consistently higher than that of SSA, as in each iteration of CSA, not only does the most beneficial switch get determined in each partition, but also all combinations of switch-filter_clusters are considered to achieve a high quality solution.

4.6.5 Discussion

Our evaluation results clearly show that the designed hybrid middleware can explore the entire range of performances (w.r.t. bandwidth efficiency and end-to-end latency) between those of PLEROMA and APP-M by adjusting the threshold factor. The results provide a clear insight into the performances of all three systems, highlighting their benefits and drawbacks. Moreover, results confirm that CSA has higher benefit

than SSA. However, as expected, the runtime overhead of CSA is higher than that of SSA. So, while choosing between SSA and CSA, one needs to consider the trade-off between quality and complexity.

4.7 Related Work

The importance of a scalable and elastic pub/sub middleware providing high throughput and low end-to-end latency has always been impressed upon. In recent times, significant contributions in this respect have been made possible with the emergence of cloud computing which has driven the idea of realizing pub/sub middleware as a cloud service. Li et al. present BlueDove [LYK+11], an attribute-based pub/sub service, that targets parallelism of the event filtering process. Bluedove organizes multiple servers into a scalable overlay as candidates for one-hop forwarding of events. Based on a multi-dimensional subscription space partitioning technique for the distribution of subscriptions between servers, BlueDove exploits skewness in data distribution for performance-aware event filtering at the least loaded servers. In fact, we use similar techniques to implement our application layer.

Similarly, Barazzutti et al., also, focus on parallelizing the event filtering process by designing StreamHub [BFF+13], a scalable pub/sub service based on a tiered architecture. StreamHub comprises a set of independent operators that take advantage of multiple cores on multiple servers to perform pub/sub operations which include subscription partitioning, event filtering, and event dispatching. Scalability of StreamHub is further supported with elasticity in e-StreamHub [BHM+14], which is capable of scaling in and scaling out depending on load observations of the system to improve system throughput. Although the performance of these services is ahead of traditional broker-based overlay implementations, they are curbed by the limitations of filtering in software.

As mentioned earlier, in literature, we find efforts towards realizing a topic-based pub/sub middleware that performs event filtering and routing within the network. As discussed in the previous chapters, LIPSIN [JZER+09] proposes an efficient multicast strategy to route events on the network layer using Bloom filters. LIPSIN performs routing similar to source-routing where the set of links to be traversed by the packet is encoded in the packet header. This encoding is restricted by the available bits in the packet header, resulting in the generation of fixed length Bloom filters. The use of fixed length Bloom filters to encode links implies the presence of false positives in the network impeding bandwidth efficiency of the system. Clearly, the data plane limitations highlight the cost that network layer implementations need to bear in order to achieve line-rate performance. Thus, the need to combine filtering at both layers, to strike a balance between their performances, is quite apparent.

4.8 Conclusion

In this chapter, we propose, implement and thoroughly evaluate the performance of a hybrid content-based pub/sub middleware. To the best of our knowledge, we are the first to combine filtering of events in application and network layers in the context of content-based pub/sub. We provide algorithms with various associated complexities and benefits to determine the layer in which each event gets filtered such that the overall false positives in the system can be minimized while staying within an average end-to-end latency threshold. The evaluation results show that our hybrid middleware can be configured by an application to various settings ranging from pure network layer filtering to pure application layer filtering by adjusting the average end-to-end latency threshold in order to achieve desired performance.

Chapter 5

Addressing TCAM Limitations

The previous chapters focus on in-network content-based filtering on software-defined networks and the problems (in terms of expressiveness) faced by content filters due to hardware limitations with respect to limited number of bits available for representation of each filter in match fields of flows. However, so far, we have not considered another very crucial limitation of hardware switches that can have a serious impact on PLEROMA. This happens to be the limitation on the number of flow table entries available for content-based filtering in the TCAM of switches.

TCAM is an expensive, power-hungry resource and as a result the number of flow table entries (forwarding rules) available for content-based filtering is limited in hardware switches. Most switch vendors design Openflow-enabled switches that typically support up to a few thousand flow entries per switch [CMT+11, KARW16], and such a hardware limitation has already been the subject of much research in the past [KLRW13, CMT+11, KARW16]. The main reason behind the design of TCAM with such limited space is the inherent trade-off between table size and other factors such as power and cost. In fact, studies show that compared to conventional RAM, TCAM consumes almost 100 times more power [STT03] and has almost 100 times more cost [100]. As a result, applications should only rely on a limited number of flow entries for their design.

Interestingly, the growth of routing table is a common problem faced by most routers and switches today. For example, in August 2014, Microsoft Cloud, Ebay, Lastpass along with some others were hit by outages as a result of full BGP routing tables [512]. The TCAM in affected Cisco routers had a default limit of 512k entries for IPv4 routes which was exceeded, causing a spillover effect. So, with a growing demand in connectivity, the limited TCAM resources must be judiciously utilized. In fact, the number of available flow entries largely depend on the match fields used. For example, in an Openflow-enabled switch such as NEC PF5240, the TCAM size for IPv6 traffic is further reduced as compared to the TCAM size for IPv4 traffic. Moreover,

considering traffic from various applications being routed over the switches of a network, the number of flow entries reserved for content-based routing is merely a fraction of the entire capacity of TCAM on a switch. As a result, to abide by the limit on number of available flow entries on TCAM, expressiveness of content filters may need to be compromised to allow for limited flows representing multiple aggregated filters.

So, in this chapter, given the constraint on available flow table entries, we propose the deployment of aggregated filters (i.e., merged flows) on switches. However, aggregation (or merging) of filters may compromise preciseness of the filters w.r.t. the subscriber interest they represent, increasing unnecessary traffic in the network. This may significantly impact bandwidth efficiency in a content-based pub/sub system where much of the benefit provided by content-based routing would be rendered less effective due to the aggregation of filters. As a result, this chapter focuses on minimizing bandwidth usage by unnecessary traffic in the network despite the given constraint on available TCAM for pub/sub traffic by judiciously making filter aggregation decisions based on multiple factors. In particular, we design techniques that use the knowledge of advertisements, subscriptions, and global network state to optimize the aggregation process such that the overall amount of unnecessary traffic in the network can be kept to a minimum. We realize and thoroughly evaluate, in both emulated environments and a real SDN testbed, a filter aggregation algorithm with different flavors having varying degrees of accuracy and complexity. Our evaluation results show that, in order to respect TCAM constraints of individual switches, the designed algorithm can perform efficient aggregation decisions that result in almost negligible unnecessary traffic in the network under realistic workload. In fact, it reduces unnecessary traffic introduced in the network due to aggregation by a baseline approach by up to 99.9%.

5.1 Impact of TCAM Limitations on PLEROMA

From the discussion on spatial indexing of content and its subsequent conversion to flow entries in Chapter 2, it is quite evident that more expressive representation of subscriptions demands the installation of multiple flows on a switch. In fact, as mentioned before, even a single subscription may yield multiple dzs which results in multiple flow entries (cf. Figure 2.2). Also, typically, applications using content-based pub/sub may have up to millions of subscribers which might require deployment of millions of filters. With such high demand of TCAM resources, it is very natural to run out of TCAM space in such applications. A limited number of available flow table entries implies two paths of action—ignoring any subsequent subscription filters once TCAM capacity is reached, or aggregating flows to reduce occupied TCAM space. The former will lead to false negatives (i.e., events that are not forwarded to interested subscribers), which is of course not acceptable in the context of this pub/sub middleware, and the latter may result in false positives (i.e., events that are forwarded to uninterested subscribers)

which means unnecessary bandwidth usage. In this chapter, we employ the latter, i.e., aggregate or merge filters, while also striving for bandwidth efficiency.

Before we discuss the details of the filter aggregation problem, it is important to understand the manner in which we can merge flows and the impact of these merges. When a flow fl_i is merged with a flow fl_j, the match field of the resultant flow fl_r has a dz that covers both dz_{fl_i} and dz_{fl_j}. In the context of spatial indexing, this effectively means that the resultant dz is the longest common prefix of the two dzs. For example, if dz_{fl_i}={1101} and dz_{fl_j}={1110}, then dz_{fl_r}={11}. Also, the instruction set for fl_r will be the union of outgoing ports (i.e., oP) of fl_i and fl_j. So, if oP_{fl_i}={1,2} and oP_{fl_j}={2,3}, then oP_{fl_r}={1,2,3}. Note, according to the previously defined flow containment relations (cf. Chapter 2), $fl_r \succ fl_i$ and $fl_r \succ fl_j$ irrespective of the relation between fl_i and fl_j. Here, we also define two other operations '+' and '-' in the context of dzs. The expression $dz_1 + dz_2$ simply refers to the two subspaces representing the two dzs being addressed together. The expression dz_1 - dz_2 refers to that part of the subspace representing dz_1 that does not overlap with dz_2. In this example, after the aggregation, fl_r forwards all traffic matching {11} through ports {1,2,3} which means that all traffic lying in the subspace {11-1101} are false positives forwarded by port 1, all traffic lying in the subspace {11-1110} are false positives for port 3, and all traffic lying in the subspace {11-(1101+1101)} are false positives for port 2. So, we see how even a single merge (merely aggregating two flows) can result in forwarding of a significant amount of false positives in the network.

5.2 Filter Aggregation Problem

It is clear from the above discussion that it is very important to select the combination of flows that should be merged on a switch as the decision directly impacts false positives in the system. As a result, in this thesis, we address the filter aggregation problem. More specifically, we consider a given system where switches may need to install sets of flows that are more than the TCAM capacity available to them and attempt to aggregate flows from the given set of original flows to meet the capacity requirements of individual switches in a bandwidth-efficient manner. Note that different switches may have different TCAM capacities available to pub/sub traffic, depending on other applications using these switches, as specified by the system administrator.

5.2.1 Problem Statement

More formally, let $\mathbb{R}$ be the set of all switches in the network and F_{R_i} be the set of all flows (or filters) that should be deployed for a given set of advertisements and subscriptions on switch R_i where $fl \in F_{R_i}$. Let cap_{R_i} be the maximum TCAM capacity of switch R_i available for pub/sub traffic. For each switch R_i, we need to determine a

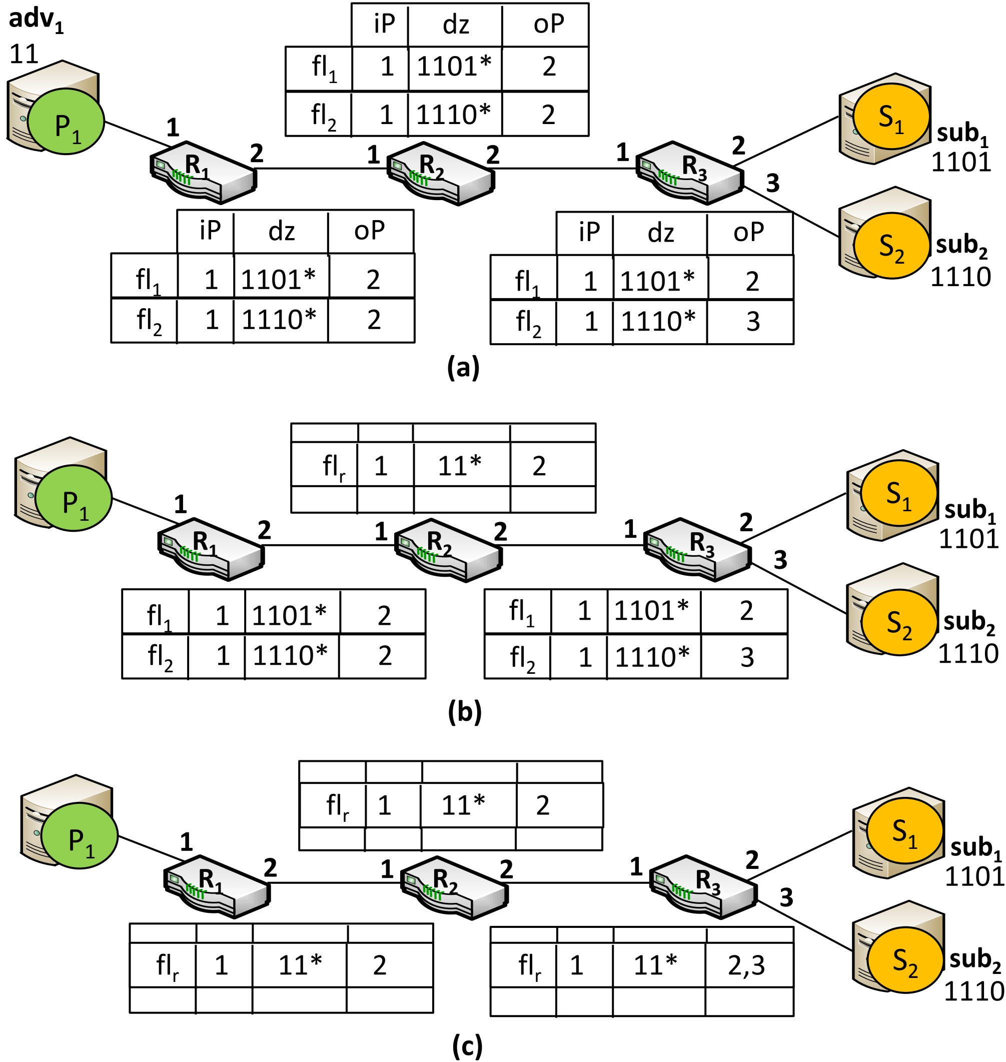


Figure 5.1: Importance of upstream switch filters

set of flows SF_{R_i} belonging to and aggregated from F_{R_i} that is within the given TCAM capacity. Let $\mathcal{C}_{R_i}$ be the aggregation cost, in terms of unnecessary traffic forwarded due to aggregation of filters, of R_i. So, our objective is to determine the set SF_{R_i} subject to $|SF_{R_i}| \leq cap_{R_i}$ for each $R_i \in \mathbb{R}$ such that $\sum_{i=1}^{|\mathbb{R}|} \mathcal{C}_{R_i}$, i.e., overall unnecessary traffic in the network due to aggregation of filters, is minimum.

5.2.2 Problem Analysis

The defined problem specifies minimizing the aggregation cost on each individual switch such that the overall false positives in the network also get minimized. The aggregation cost of a switch is nothing but the sum of the aggregation cost of all the flow merges

made on that switch to meet the TCAM capacity of that switch. As a result, while deciding on the filters to be aggregated on a switch, the aggregation cost of each possible merge should be calculated. However, just looking at the flow information local to a switch for a possible local merge is not the optimal way to determine its cost as our investigation into the defined problem shows that existing filters on other switches in the network have a significant role to play on the aggregation cost of a merge. In fact, the main challenge in the filter aggregation problem is the determination of the aggregation cost of each possible merge on a switch which depends largely on the already installed filters on switches in the upstream paths of the aggregated filter.

To understand the importance of filters on switches in the upstream paths of a filter being considered for a merge, we look at an example depicted in Figure 5.1. Figure 5.1(a) shows a system with a publisher and two subscribers and their respective advertisement and subscriptions which result in the deployment of the depicted flows on the three switches. Each flow is depicted by the flow name, incoming port (iP), dz constituting the destination IP address, and the outgoing ports (oP) in IS. Let us assume that R_2 can only accommodate a single flow. As a result, fl_1 and fl_2 on R_2 are merged, in the manner explained in Section 5.1, to compose fl_r as depicted in Figure 5.1(b). fl_1 and fl_2 had the same incoming and outgoing ports and as a result only the filter subspace in fl_r gets expanded. Now, R_2 will forward all traffic matching {11*} instead of just {1101*} and {1110*}. However, if we look upstream, we see that R_1 will already filter out any traffic that does not lie within the subspaces {1101} and {1110} in Ω. As a result, the merge at R_2 does not impact the false positives in the system as R_2 does not receive any additional false positives from its upstream path and acts only as a forwarder in Figure 5.1(b). However, the scenario is different in Figure 5.1(c) where there is a need to merge the two flows on R_1. In this case, not only does R_1 forward all traffic matching {11*} but also these false positives get forwarded by R_2 due to the aggregation of its filter. At R_3 too, owing to a merge, these false positives from previous switches do not get filtered out. In fact, as the resultant flow combines the outgoing ports of the two original flows fl_1 and fl_2 on R_3, false positives are now forwarded along both downstream links of R_3. Port 2 forwards false positives lying within the subspace {11 - 1101} and port 3 forwards those lying within {11 - 1110}. If the upstream filters at R_1 or R_2 were precise, then fl_r on R_3 would only forward false positives lying within {1110} through port 2 and {1101} through port 3 as the remaining would be filtered out upstream. This example clearly indicates that (i) even if filter expansion occurs, false positives forwarded by a merged flow depends on the filter aggregation on upstream switches. Also, (ii) even if no filter aggregation occurs on the upstream path, an aggregation involving merging of two or more flows whose outgoing ports are not subsets of each other will result in traffic, meant to be forwarded by one port, being forwarded by the remaining ones as well. This will always result in false positives along all involved outgoing ports (cf. R_3 in Figure 5.1(c)).

Clearly, due to the importance of flows in the upstream paths, a merge on a switch

based on flow information local to that switch is not the most optimal solution. As a result, while calculating the cost of a possible merge on a switch, we propose to consider the global view of the network state to avoid as much unnecessary traffic as possible due to aggregation.

5.3 Filter Aggregation Algorithm

While defining our filter aggregation problem, we specify that the input to the problem is a set of flows which need to be aggregated to meet the TCAM capacity of the switches, and this set of flows is maintained by the controller based on the current subscriptions and advertisements in the system. Let us assume that $\mathbb{ER} \in \mathbb{R}$ is the set of switches in the network where one or more flow(s) need to be aggregated, i.e., $\forall R_i \in \mathbb{ER}, |F_{R_i}| > cap_{R_i}$. So, now, we need to reduce the number of flows according to the given capacity on each switch in $\mathbb{ER}$.

Approach Overview: As discussed in the previous section, our objective is to reduce the combined aggregation cost of all switches in the network. So, it is important to reduce the aggregation cost of each individual switch. While doing so, we try to aggregate those flows that result in minimum aggregation cost for the switch while staying within the maximum available TCAM capacity. As a result, the main idea behind the filter aggregation algorithm is to calculate the aggregation cost of each possible flow merge on a switch $R \in \mathbb{ER}$, and then select a combination of those flow merges that would result in minimum combined aggregation cost for the switch below its designated capacity. As we saw in the previous section, the aggregation cost of a possible merge depends on filters installed on previous switches. In fact, the main challenge that the filter aggregation algorithm faces is the determination of an efficient cost function for a possible merge that captures various factors of the aggregation cost, including dependencies on upstream switch filters.

So, in the remaining part of this section, we introduce the details of the mechanisms used to (i) identify the possible combinations of flow merges on each switch, (ii) calculate the cost and benefit for each of these flow merges, (iii) select the set of merges resulting in minimum aggregation cost for the switch such that the resulting number of flows is within the capacity threshold for the switch.

5.3.1 Filter Aggregation on a Switch

While discussing the details of our proposed filter aggregation algorithm, we, first, explain the method of determining all possible flow merges on the switch being considered for aggregation, and then provide methods to select merges from these possibilities such that minimum aggregation cost is incurred for that switch.

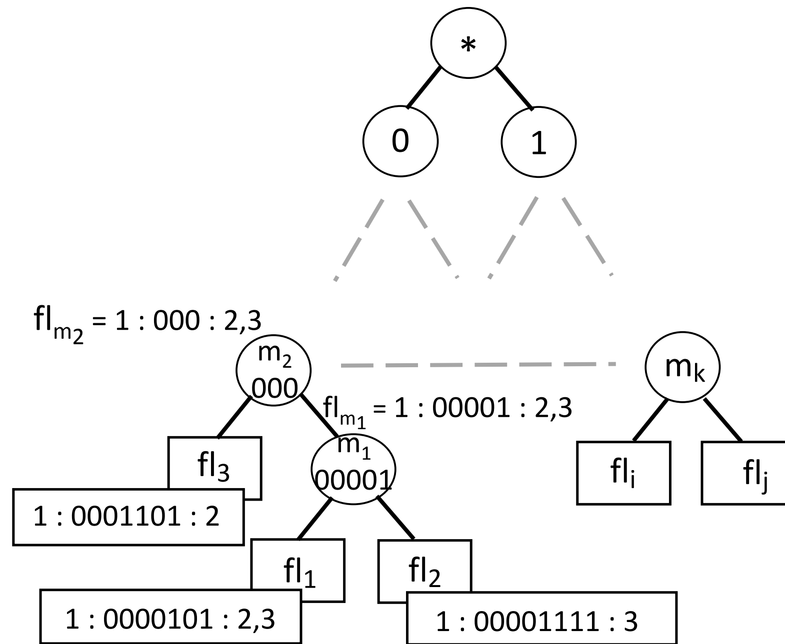


Figure 5.2: Merge Point Tree for Incoming Port 1

5.3.1.1 Determining Possible Flow Merges

When a switch exceeds its flow limits, various combinations of flows may be merged to reduce the flow count on that switch. We denote every possible merge as a *merge point*. So, the objective is to select an ideal set of merge points on a switch that has minimum combined aggregation cost. In fact, to determine all possible merge points on a switch, we create a prefix tree called the merge point tree which contains all possible merge points. However, not all flows can be merged to create a merge point. Two flows cannot be merged if one of the outgoing ports of a flow is the incoming port of the other. This will lead to cycles in the network and we call these flows with such a conflicting relation as conflicting flows. So, clearly, merge points are only possible for non-conflicting flows. For the sake of simplicity, we create a separate merge point tree for every incoming port of a switch, i.e, merge points in the tree merge flows that have the same incoming port. This ensures the absence of conflicting flows within each tree as this eliminates the possibility of merging two flows where the incoming port of one is among the outgoing ports of the other. Of course, while selecting the ideal set of merge points for a switch, all merge points across all trees are considered.

So, a merge point tree is a prefix tree where every non-leaf node is a merge point and every leaf node is a flow. In the tree, a merge point signifies the minimum filter expansion required to cover two or more unrelated filters. So, a flow is merged with another flow if this results in minimum filter expansion for this flow as compared to the filter expansion when merged with others. The first step towards creating a merge

point tree is to identify the flows $\in F$ with the longest common prefixes among all flows and perform their respective merges to create merge points. So, within the tree these identified flows with longest common prefixes form the lowest level nodes and the newly created merge points form the nodes at the immediate upper level of the tree. At the following upper levels, merging according to longest common prefix continues, this time with not only the remaining flows but also the merge points from the lower levels, until we finally arrive at the root of the merge point tree which represents a filter covering the entire event space Ω. For example, Figure 5.2 depicts a merge point tree aggregating all flows with a specific incoming port 1. This merge point tree depicts merge points and flows where flows have the format $iP : dz : oP$. In this example, let us assume that fl_1 and fl_2 have the longest common prefix, i.e., {00001}, among all flows and, therefore, reside on the lowest level of the tree. So, when they are merged, their immediate merge point is m_1 with a dz of {00001}. So, the resultant flow at m_1, i.e., fl_{m_1}, represents the filter with minimum expansion required to forward the traffic for both fl_1 and fl_2. We continue building the tree upwards, now, with not only remaining flows but also all newly created merge points from the already existing levels. So, let us assume that a flow fl_3 shares the longest common prefix with m_1. So, at the next upper level, m_1 and fl_3 are merged to create m_2. Please note that two merge points of a level may similarly be merged based on their common prefixes. The entire merge point tree is built once the root representing the entire space is reached. A merge point tree contains all flows on the switch and all possible merges signified by the merge points. Since a merge point merges all flows belonging to its child nodes, clearly, a merge point in the upper level of the tree merges more flows as compared to a merge point at a relatively lower level. For example, aggregation at m_1 reduces the flow count on the switch by 1 as fl_{m_1} aggregates 2 flows. However, aggregation at m_2 at an upper level reduces flow count by 2 as fl_{m_2} aggregates 3 flows fl_1, fl_2, and fl_3. Even though m_2 reduces more flows, there is a possibility that it forwards more false positives as the filter expansion for fl_1 and fl_2 is more at m_2 than at m_1.

5.3.1.2 Selecting Flow Merges on a Switch

We provide a detailed description of the steps of the flow selection process as follows and a formal description in Algorithm 6. Once the set of all merge points (M) across all merge point trees are determined for the switch being processed, the aggregation cost (denoted by $\mathcal{C}$) for each of them is calculated to determine the final set of merges on the switch. We explain the cost calculation at each merge point in details in the following Section 5.3.2. Having calculated the aggregation cost ($\mathcal{C}$) for every possible merge point across all merge point trees of a switch, we also determine the benefit (denoted by $\mathcal{B}$) of each merge. The benefit is, simply, the number of flows reduced on the switch due to the merge, i.e., $\mathcal{B}_m$ = number of flows covered by merge point m - 1 (cf. Algorithm 6, line 4). Next, we calculate the cost per benefit of each merge point m, i.e., $\mathcal{C}_m/\mathcal{B}_m$ (cf. Algorithm 6, line 5). For a switch, say R_i, we start selecting merge

Algorithm 6 Selection of Flow Merges on a Switch R_i

1: $M \leftarrow$ Calculate all merge points on R_i
2: **for all** $m \in M$ **do**
3: $\quad \mathcal{C}_m \leftarrow$ Calculate aggregation cost of m (cf. Algorithm 7)
4: $\quad \mathcal{B}_m = |F_m|$ - 1
5: $\quad costPerBenefit_m = \mathcal{C}_m / \mathcal{B}_m$
6: $SF_{R_i} = F_{R_i}$
7: **while** $|SF_{R_i}| > cap_{R_i}$ **do**
8: $\quad \mathcal{C}_per_\mathcal{B}_Set = \{costPerBenefit_m : m \in M\}$
9: $\quad m_{selected} = \{m \in M : costPerBenefit_m = min(\mathcal{C}_per_\mathcal{B}_Set)\}$
10: $\quad SF_{R_i} = SF_{R_i} \setminus F_{m_{selected}}$
11: $\quad SF_{R_i} = SF_{R_i} \cup fl_{m_{selected}}$
12: $\quad$ Update M

points with minimum cost per benefit for the final set of selected flows (i.e., SF_{R_i}) on R_i.

We start by initializing SF_{R_i} to the set of all original unaggregated flows (cf. Algorithm 6, line 6). Then, we select the merge point, say $m_{selected}$, with minimum cost per benefit from the set of all merge points (cf. Algorithm 6, line 9). Once $m_{selected}$ gets selected, (i) all original flows covered by $m_{selected}$ are removed from SF_{R_i}, (ii) the flow $fl_{m_{selected}}$ at $m_{selected}$ is added to SF_{R_i}, (iii) the next step depends on one of the following three scenarios—(a) if a merge point m_j is covered by the selected merge point $m_{selected}$ (i.e., $m_{selected} \succ m_j$), then m_j is removed from the set of merge points as it is now redundant, (b) if a merge point m_j covers the selected merge point $m_{selected}$ (i.e., $m_j \succ m_{selected}$), then m_j cannot be removed but its cost and benefit get reduced as some of its cost and benefit have already been considered when selecting $m_{selected}$, and (c) if $m_{selected}$ has no relation with any other merge point then no action is taken. After each selection, the next merge point with least cost per benefit is selected until $|SF_{R_i}| \leq cap_{R_i}$, i.e., the number of flows on R_i is within the TCAM capacity of the switch (cf. Algorithm 6, lines 7-12).

Once the final set of flows for every switch which exceeds TCAM capacity is determined, the flow changes are pushed onto the physical network and all hardware switches in the network are updated accordingly. This concludes the final step of the filter aggregation algorithm.

5.3.2 Aggregation Cost at a Merge Point

As the final selection of merge points on a switch depends largely on their cost per benefit value, determination of the cost (in terms of false positives forwarded by the aggregated filter) for each merge is an integral part of this algorithm. The aggregation cost of a possible merge is nothing but the amount of additional network false positives that the merge could introduce in its downstream paths.

So, in order to calculate the cost of a merge point, say m, we need to, firstly, identify the incoming traffic at the incoming port of the merged flow fl_m at m as only this traffic is relevant for forwarding by fl_m. With regards to incoming traffic, note, there may be multiple incoming paths ($iPaths$) from multiple publishers that forward traffic to the incoming port of fl_m. The key factors in determining the incoming traffic are the traffic load of each publisher intended to be forwarded along $iPaths$ and the upstream switch filters which influence the filtering of this published traffic.

Secondly, we need to identify the false positives (fp) from this incoming traffic that fl_m forwards along its downstream paths. While calculating downstream false positives forwarded by fl_m for a specific path, note that each outgoing port, $op \in oP$, of fl_m has its own set of downstream paths to subscribers. Let the number of downstream links of the downstream paths of an outgoing port be denoted by $dLinks$. Also, each outgoing port forwards its own share of false positives after the merge based on the traffic it was meant to forward as per the original flows. For example, at m_1 in Figure 5.2, outgoing port 2 forwards false positives lying within the subspace {00001-0000101} after the merge as this port originally forwarded events matched only by fl_1. On the contrary, all traffic lying within {00001-(0000101 + 00001111)} are false positives for outgoing port 3 after the merge, as this port originally forwarded events matched by both fl_1 and fl_2.

So, with regards to the amount of false positives forwarded by each outgoing port, $op \in oP$, of fl_m, the key factors are, first and foremost, the expansion in filter space due to the aggregation of the original filter spaces, the incoming traffic along each path $\in iPaths$, and the number of downstream links along the downstream paths of *op*. So, broadly speaking, the aggregation cost of a merge point is as follows :

$$\mathcal{C} = \sum_{p \in iPaths} \sum_{op \in oP} fp^{p}_{op} * dLinks_{op} \tag{5.1}$$

In fact, the two flavors of cost calculation proposed in this chapter—load-based method and pattern-based method—differ only in the manner of determining the false positive value, i.e, fp in Equation 5.1 as discussed later in this section. The load-based method estimates resultant network false positives due to the merge by using the knowledge of incoming traffic load, whereas the pattern-based method collects and inspects published event packets to accurately determine network false positives introduced due to the merge by not only using the knowledge of traffic load but also traffic pattern in Ω.

In the remaining part of this section, we introduce the details of (i) determining the incoming traffic at a merge point and (ii) determining false positives from this incoming traffic along the downstream paths of the merge point. As explained above, these two steps together determine the aggregation cost, $\mathcal{C}$, of a merge point. In the following, we provide a detailed description of the steps of aggregation cost calculation along with a formal description in Algorithm 7.

Algorithm 7 Aggregation cost calculation at a merge point m

```
1:  procedure calculateAggregationCost(m) do
2:    pubSet ← Set of all upstream publishers with advertisements overlapping with dz_{fl_m}
3:    C = 0 // Aggregation cost
4:    for all P ∈ pubSet do
5:      C+ = computeCostOnPath(P, fl_m)
6:    return C
7:  procedure computeCostOnPath(P, fl_m) do
8:    uFilters ← Get all relevant path filters between P and fl_m // Set of upstream filters
      with flow relations to fl_m along the path to P
9:    mfgDzs = {dz_fl : fl ∈ uFilters}
10:   for all fl_i ∈ uFilters do
11:     for all fl_j ∈ uFilters do
12:       if fl_i ≠ fl_j ∧ dz_{fl_i} ≻ dz_{fl_j} then
13:         mfgDzs = mfgDzs \ dz_{fl_i}
14:   mfgDzs = mfgDzs ∩ DZ(P) ∩ dz_{fl_m} // dzs of most fine-grained upstream filters
15:   costOnPath=0
16:   for all oP ∈ fl_m do
17:     F_oP = {fl ∈ F_m : oP_fl = oP} // set of flows being merged (F_m) at m that have oP
        among their out ports
18:     fpSpace = mfgDzs \ (∪_{fl∈F_oP} dz_fl ∩ mfgDzs) // False positive subspace
19:     dLinks ← No. of downstream links of oP
20:     if Load-based Method then
21:       adSpace = ∑_{dz∈DZ(P)} dz // advertised subspace
22:       p_traffic ← Traffic published in the advertised subspace
23:       costOnPath+ = (fpSpace/adSpace) * p_traffic * dLinks
24:     if Pattern-based Method then
25:       pb_fp ← Calculate false positives through oP from fpSpace and event history
26:       costOnPath+ = pb_fp * dLinks
27:   return costOnPath
```

5.3.2.1 Incoming Traffic

Let us calculate the aggregation cost ($\mathcal{C}_m$) at a merge point, say m, of a switch R_i which aggregates a set of flows denoted by F_m. As mentioned earlier, in order to calculate the aggregation cost of m, the first step is to determine the incoming traffic at the incoming port of the newly aggregated flow fl_m at m. So, first, we identify all relevant publishers publishing events that will arrive at this incoming port. If $DZ(P)$ is the set of dzs representing an advertisement of publisher P, then P is a relevant publisher even if one of the $dzs \in DZ(P)$ covers or is covered by dz_{fl_m} and there is a path from P to the incoming port of fl_m. So, by identifying all relevant publishers, we also identify all paths, i.e., $iPaths$ from these publishers to the incoming port of fl_m. Next, we proceed to determine the incoming traffic from each path $p \in iPaths$ so that we can eventually calculate the aggregation cost for each path depending on the amount of traffic each upstream path forwards to fl_m and the amount of false positives among this traffic that fl_m forwards to its downstream paths.

As mentioned earlier, the amount of incoming traffic along a path depends on the fil-

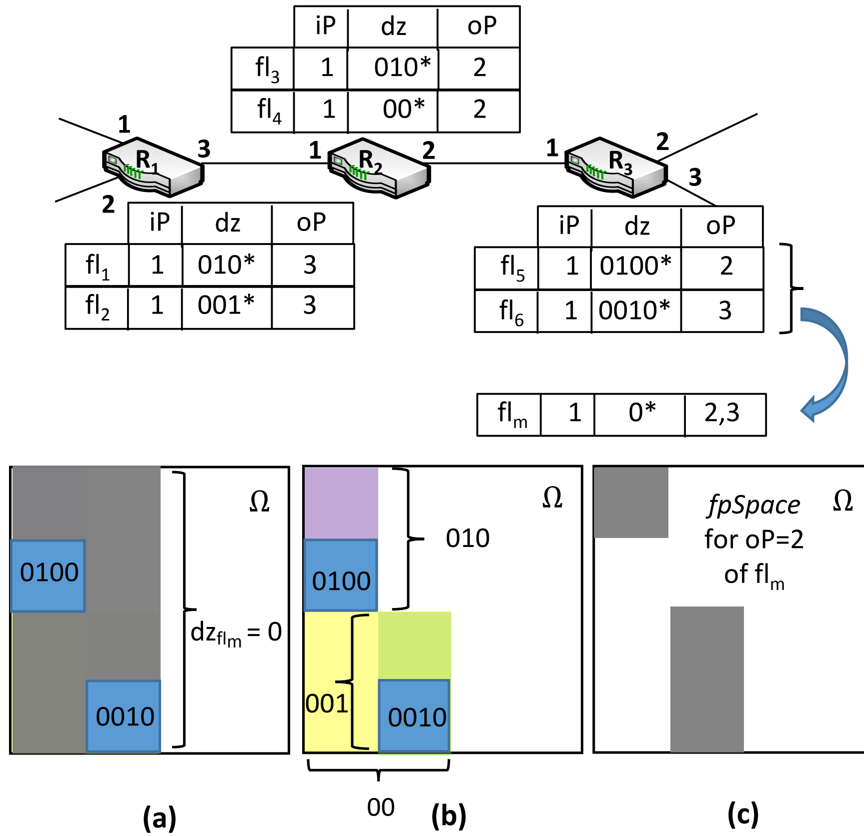


Figure 5.3: Cost Calculation

ters installed on the upstream paths of the merge point. So, for a path $p \in iPaths$, first, we determine the set of all upstream filters, i.e., $uFilters$. This is, effectively, the set of filters on all upstream switches on the current path that forward events to fl_m. As explained in Section 5.2, these upstream filters/flows are of utmost importance in determining the aggregation cost. In fact, the most fine-grained filters among this set dictate the incoming traffic for fl_m as these filters already filter out the bulk of unnecessary traffic. So, it is imperative to only identify the set of most fine-grained filters, i.e., $mfgFilters \in uFilters$, as the traffic forwarded by them is the only traffic that reaches fl_m. We denote the set of dzs representing the filter subspaces of $mfgFilters$ as $mfgDzs$ (cf. Algorithm 7, lines 8-14). Let us look at an example from Figure 5.3 where the aggregation cost of m needs to be calculated on switch R_3 and $F_m = \{fl_5, fl_6\}$. The set of relevant flows on upstream switches, i.e., $uFilters$, of a path p that connects a publisher P to fl_m, consists of fl_1, fl_2, fl_3, and fl_4 as depicted in the figure. Figure 5.3(b) illustrates the event subspace representation of each of the filter dzs of $uFilters$ and Figure 5.3(a) depicts that of fl_m. From the figure, it is quite clear that $\{dz_{fl_4} = 00\} \succ \{dz_{fl_2} = 001\}$. This means that fl_2 already filters out events lying in the subspace $\{00$ - $001\}$ depicted by the yellow subspace in Figure 5.3(a). As

a result, all events lying within the yellow subspace do not reach the incoming port of fl_m, and so, this subspace cannot be considered as a false positive subspace in the aggregation cost of fl_m. So, in this example, $mfgFilters = \{fl_1, fl_2, fl_3\}$ and the effective subspaces they represent, i.e., the most fine-grained *dz*s, $mfgDzs = \{001, 010\}$. Also, only those subspaces in $mfgDzs$ are considered to contribute to the incoming traffic that lie within the advertised subspace of the current publisher as the remaining subspace cannot be accounted for any incoming traffic of fl_m due to the absence of published events lying within them. So, all traffic lying within $mfgDzs$ can now be considered as the incoming traffic at a merge point m for a path p.

5.3.2.2 False Positives on Downstream Paths

Having identified the subspaces that forward traffic to the incoming port of fl_m along a specific path, we proceed to calculate the false positives lying within these subspaces that will be forwarded by the current merged filter fl_m along its downstream paths. However, before we do so, please recall that if fl_m aggregates original flows which have different outgoing ports, then the false positives for one outgoing port may be different from those of the others. As a result, we calculate the amount of false positives that may be forwarded by each outgoing port of fl_m separately and compute the sum of the individual costs of each outgoing port $\in oP$ of fl_m to obtain the total aggregation cost of fl_m for the specific path (cf. Algorithm 7, lines 15-26). So, for each outgoing port, $op \in oP$, we identify the original flows, i.e. $F_{op} \in F_m$, that should forward traffic through the current outgoing port *op*. With the information of the *dz*s of the flows that are originally supposed to forward events through the current port, it is easy to determine the false positive subspace (*fpSpace*) for *op*. So, the effective false positive subspace (*fpSpace*) for each port of an aggregated filter can be computed by removing the *dz*s of all flows belonging to F_{op} from the subspace representing $mfgDzs$. Events lying in *fpSpace* are the only unnecessary events that will be forwarded by the current outgoing port of the aggregated flow fl_m. For example in Figure 5.3(c), the gray area is the effective *fpSpace* for outgoing port 2 of fl_m. All events, published by the publisher P, that lie in this subspace account for the aggregation cost of port 2 for the specific path p.

Once we calculate *fpSpace*, we use this information to calculate the actual number of false positives along all downstream links (*dLinks*) of the outgoing port in consideration to determine the aggregation cost at this port. Here, we differentiate between the two flavors of cost calculation, i.e., load-based method and pattern-based method.

Load-based Method (FA-LB): The load-based method uses the traffic load of the publisher along the current path in consideration and the value of *fpSpace* to estimate the false positives of the outgoing port. More specifically, it collects statistics related to the total number of events ($p_{traffic}$) published by the current publisher in the advertised subspaces (*adSpace*) and estimates the false positives within *fpSpace*.

To calculate this estimated number of false positives, we quantify (qt) subspaces as a fraction of the entire event space Ω. So, in Figure 5.3, while calculating false positives forwarded by port 2, the qt value for the subspaces representing $mfgDzs$ is 1/4, that of the subspace representing the dz of F_{op}, i.e.,{0100}, is 1/16, and, therefore, that of $fpSpace = 3/16$. So, using the quantified values for the subspaces, the estimated false positives within $fpSpace$ are $(fpSpace/adSpace)$*$p_{traffic}$.

As mentioned earlier, the aggregation costs of all outgoing ports of fl_m are summed up to calculate the aggregation cost of a path, and then the aggregation costs of all paths are summed up to calculate the total aggregation cost of a merge point. So, we formally define the aggregation cost of a merge point using load-based method by extending Equation 5.1 as follows:

$$\mathcal{C} = \sum_{p \in iPaths} \sum_{op \in oP} (fpSpace^p_{op}/adSpace^p) * p^p_{traffic} * dLinks_{op} \tag{5.2}$$

Pattern-based Method (FA-PB): While the load-based cost calculation method factors in all key aspects of aggregation to compute the aggregation cost, it does not consider the actual distribution or pattern of published events. The load-based method estimates false positives by considering traffic published by each relevant publisher to be uniformly distributed over the event space. However, published events are not necessarily distributed uniformly within Ω. As a result, we introduce another flavor of cost calculation which determines the amount of false positives that could be forwarded by the aggregated filter more accurately by looking at the content of past events and determining the event distribution. Our evaluation results show that even though the pattern-based method has more overhead, it is more bandwidth-efficient than the load-based method.

More specifically, in this method, we collect published events from all publishers. For a given path p, the exact number of forwarded false positives ($pb_fp^p_{op}$) can be determined for each $op \in oP$ of fl_m depending on the calculated $fpSpace$ and the events published on that path by investigating the content of each event and determining whether it lies within the $fpSpace$ in question. So, we define the aggregation cost of a merge point using pattern-based method by again extending Equation 5.1 as follows:

$$\mathcal{C} = \sum_{p \in iPaths} \sum_{op \in oP} pb_fp^p_{op} * dLinks_{op} \tag{5.3}$$

Collecting events from all publishers, maintaining the set of all events, and considering the content of every event comes with its share of overhead. As a result, we introduce the sampling factor, denoted by sfr, which determines the fraction of events to be collected and considered for cost calculation from the set of all published events. So, if $sfr = n$, only every $1/n$th event from a publisher is collected and considered for cost calculation. Of course, here, a sampling factor of 1 implies the collection and consideration of every event from all publishers. A smaller sampling factor may reduce overhead significantly while a higher sampling factor may provide much more accuracy.

5.3.3 Resolving Dependencies Between Switches

In our filter aggregation algorithm, we have considered that, while calculating the aggregation cost at a merge point on a switch, all upstream filters are already known. However, it may so happen that one or more switches in the upstream paths of a merge point also belong to $\mathbb{ER}$ on which the final set of flows is yet to be decided. This highlights the importance of having an order of processing switches belonging to $\mathbb{ER}$ in this algorithm as each switch is dependent on other switches in the network. As a result, we start processing switches in $\mathbb{ER}$ from publishers to subscribers. However, depending upon the locations of publishers and subscribers in the network, it may so happen that two or more switches have inter-dependencies, i.e., switches in $\mathbb{ER}$ may belong to each others upstream paths. For instance, switch R_1 may be an upstream switch for one or more flows on a switch R_2 and vice versa. To this end, our algorithm enforces a random processing order on such switches by selecting one of the inter-dependent switches, say R_1, and calculating the cost of merge points at R_1 while assuming the worst case at R_2 , i.e., R_2 installs the coarsest filters. Once the order of processing switches in $\mathbb{ER}$ has been decided, the main flow aggregation decision-making process of the algorithm commences on each switch $R \in \mathbb{ER}$ in the determined order.

5.3.4 Handling Dynamics

The filter aggregation algorithm, discussed in this section, is not applied to the system for every incoming subscription and advertisement that results in the exceeding of capacity in network switches as this would prove to be expensive. As a result, it runs periodically in the system. In the meantime, when a subscription or advertisement arrives and its arrival results in exceeding of capacity by just a few flows in one or more switches, an immediate aggregation must be done to avoid false negatives in the system. For this purpose, we design two flavors of a local aggregation approach—basic local aggregation (LA-B) and cost-based local aggregation (LA-C), varying in quality and overhead—just for the affected switches to ensure dynamic behavior of the system till the filter aggregation algorithm is again applied to the system.

So, when a subscription/advertisement arrives at the controller, the usual flow installation is performed for each dz representing it. While installing a flow for a particular dz, say dz_{sub}, on a specific switch, say R, the controller discovers that the capacity of that switch is already full. As a result, to accommodate the new flow, an aggregation of at least 2 flows must be performed on R, and a local aggregation approach is employed for this purpose.

Algorithm 8 Cost calculation at a merge point m in Cost-based Local Aggregation

1: $\mathcal{C} = 0$ // Initialize aggregation cost
2: **for all** $op \in oP_{fl_m}$ **do**
3: $F_{op} = \{fl \in F_m : op \subseteq oP_{fl}\}$ // Set of flows being merged (F_m) at m that have op among their out ports
4: $tpSpace = \{dz_{fl} : fl \in F_{op}\}$
5: $fpSpace = dz_{fl_m} \setminus tpSpace$
6: $\mathcal{C}+ = fpSpace$ // Quantified value of false positive subspace

5.3.4.1 Basic Local Aggregation (LA-B)

We, first, explain the basic local aggregation approach (LA-B). The main idea behind LA-B is to simply merge two flows without conflicting relations (cf. Section 5.3.1.1) on a switch with exceeded capacity such that only the knowledge of the state local to a switch is required for aggregation. We, again, use the merge point trees local to the switch for this purpose. As only a single flow needs to be reduced, only the merge points connected to the leaf nodes (flows) in the lowest level of the tree are considered for aggregation. Please recall that filter expansion of involved flows is the least in the lowest level and increases as we go up the tree towards the root. So, the local aggregation approach selects any one of these merge points merging flows at the lowest tree level whenever a switch exceeds its capacity on advent of a subscription or advertisement. Such an approach portrays an aggregation technique with least overhead.

5.3.4.2 Cost-based Local Aggregation (LA-C)

The cost-based local aggregation approach (LA-C) builds on the same lines as LA-B and, also, only considers state local to the switch. However, it does not merely select a merge point connected to leaf nodes, but instead calculates the cost of the merge points based on local switch state and selects the merge point with the least cost. This may sound similar to the described filter aggregation algorithm except for the fact that the state on other switches and traffic statistics are not considered for a local merge to keep the overhead to a minimum. Here too, we make use of the merge point trees at the switch. However, the cost calculation of each merge point in LA-C is much simpler. The only factor taken into consideration for calculating the cost is subspace expansion of the original flows at a merge point.

Similar to LA-B, in LA-C, as only a single flow needs to be reduced, only the merge points connected to the leaf nodes (flows) in the tree are considered for aggregation, reducing computation overhead significantly. Let the set of these merge points be LM. Now, for each $m \in LM$, we calculate the aggregation cost. The local cost calculation algorithm is formally explained in Algorithm 8 and described in details as follows.

Let F_m be the set of flows being merged at m. Again, while calculating the cost ($\mathcal{C}_m$) at a merge point m, we calculate the aggregation cost for each outgoing port of the

merged flow fl_m and sum up the costs of all the outgoing ports to get the overall aggregation cost of m. For each outgoing port op, we find the set of flows $F_{op} \in F_m$ that are supposed to forward events through op. The subspaces represented by the dzs of the flows in F_{op} constitute the true positive subspace for m as these represent the original filters that should forward events through op (cf. Algorithm 8, line 4). Of course, the remaining subspace within dz_{fl_m} is the false positive subspace for op (cf. Algorithm 8, line 5). The quantified value of this false positive subspace is effectively the filter expansion measure, and this is considered as the aggregation cost of op (cf. Algorithm 8, line 6).

As mentioned earlier, the sum of the costs of all outgoing ports of fl_m provides $\mathcal{C}$ for m. Once $\mathcal{C}$ for all merge points in LM is calculated, the merge point with least aggregation cost is selected and accordingly the flows on the switch are modified. This is how the cost-based local aggregation algorithm preserves the dynamic behavior of the system by avoiding false negatives with very low overhead till the filter aggregation algorithm is performed. LA-C proves to be more bandwidth-efficient than LA-B but has a higher computation overhead than it, as shown in Section 5.4.

5.3.5 Ensuring Data Plane Consistency

The aggregation algorithms, especially filter aggregation algorithm, performs a significant amount of changes to the data plane once the most bandwidth-efficient set of flows, i.e., SF_{R_i} is determined for each switch R_i belonging to $\mathbb{ER}$. As discussed earlier, frequent and significant changes to the existing flows on the switches of the network brings forth the issue of ensuring data plane consistency. Again, in this scenario of filter aggregation, false negatives may occur due to packet drops in the network while the flows on the switches (i.e., the data plane of the software-defined network) are being modified. Event packets that are in transit while flows are being removed and added may be dropped in case they do not find a matching flow.

Of course, where TCAM table size is the primary constraint, a method similar to versioning [RFR$^+$12] cannot be applied as it is extremely resource intensive. Instead, the method used to ensure data plane consistency in Chapter 3, i.e., the use of temporary multicast flows forwarding all pub/sub traffic during the transition from one network state to the other may be used. This implies that, during filter aggregation, when flows are being modified on a switch, temporarily the use of multicast flows will result in the switch forwarding all pub/sub traffic for a short period of time. The question is, can this be avoided for the filter aggregation algorithm? Can we do better in this case?

Please recall that the algorithms in Chapter 3, largely, needed to wipe out the entire old network state and reinstall all flows based on the new content representation. The old and the new set of flows are semantically very different in such a scenario and bear no relation to each other. However, the scenario is very different in the case of filter aggregation where not all flows need to be modified on a switch. Also, the existing

flows and the new flows have the same semantics and bear flow containment relations as defined in Chapter 2.

So, in our system, we resolve the problem of data plane consistency with minimum effort by simply ordering the switch updates in a strategic manner. In fact, we ensure that the selected aggregated flow fl_m is installed first on the switch, and then, all flows covered by it that are now redundant, i.e., redundant flows in F_m, are removed. By doing so, we ensure that the events matching the original flows in F_m already have an alternate path through the aggregated flow before their former paths are removed. We deploy one aggregation at a time, i.e., only once a merged flow for a selected merge is installed and all its original flows removed, do we proceed to install the next merged flow and so on. This ensures that only one additional flow on the switch is required at a time for ensuring data plane consistency. This simple approach to tackle the consistency problem is only possible here as we can take advantage of the containment relations between the new aggregated flows and the existing flows covered by them.

Please note that it may so happen that while periodically employing the filter aggregation algorithm, a few merged flows, currently installed on the switch, need to be replaced by their original flows (i.e., unmerged) depending on the latest set of flows selected and aggregated by the latest execution of the algorithm. In such a scenario, all latest merged flows are, first, serially deployed (and the flows covered by them are removed), thus making enough space in TCAM to perform unmerges. While unmerging as well, first, the original flows are deployed and then the merged flow containing them is removed to ensure that there is always a path for relevant events. We continue to ensure that only one additional flow on the switch is required at a time for ensuring data plane consistency by performing one unmerge at a time. In this manner, we completely avoid false negatives in the network with minimum overhead by simply ordering the necessary switch updates.

5.4 Performance Evaluations

In this section, we evaluate and analyze the various aspects of the presented filter aggregation algorithm. More specifically, we conduct a series of experiments to measure and compare, primarily, the impact on overall false positives in the network and the runtime overhead of the two flavors of filter aggregation algorithm (FA), i.e., the load-based method (FA-LB) and the pattern-based method (FA-PB), with the two flavors of local aggregation, i.e., basic local aggregation (LA-B)—which we consider to be a baseline approach—and cost-based local aggregation (LA-C), to show the potential of each of the proposed methods.

5.4.1 Experimental Setup

We perform our performance evaluations mainly under two test environments (cf. Chapter 2)—1) *SDN-m*, for emulating a variety of networks and 2) *SDN-t-hswitch* consisting of the Whitebox Openflow-enabled EdgeCore switch and commodity PC hardware. To show the impact of severe TCAM limitations on the performance of the system and how the designed aggregation ensures bandwidth efficiency despite severe constraints, we constrain the TCAM capacity (i.e., *cap*) of each switch to up to 600 flows. While using *SDN-m*, we experiment with up to 300 switches and 4402 end-hosts on different topologies. In fact, to capture the false positives along every link of the network and gather the overall network false positives, we also implement our own analyzer.

We use both synthetic and real-world data for our experiments. To generate synthetic data, we use a content-based schema that uses up to 5 attributes, where the domain of each attribute varies between the range [0,1023]. Our evaluations include up to 15,000 subscriptions and up to 100,000 events. To generate synthetic data, we, again, use uniform and zipfian distributions. Also, we, again, use real-world workload in the form of stock quotes procured from Yahoo! Finance containing a stock's daily closing prices to show the performance of our system in a realistic environment.

5.4.2 Comparing Network False Positive Rate

We define the term false positive rate as the percentage of total number of events forwarded in the network that are unnecessary (i.e., network false positives). The first set of experiments compares the network false positive rate for the various aggregation methods with increasing number of subscribers where the TCAM capacity of each switch in the network is constrained. We compare the load-based method (FA-LB), the pattern-based method (FA-PB) of our filter aggregation algorithm with the two local aggregation methods, basic local aggregation (LA-B) and cost-based local aggregation (LA-C). Please note that, here, we consider a sampling factor of 1 for the pattern-based method which means that every published event is considered to determine the event distribution for cost calculation of each merge point.

Figure 5.4(a) and Figure 5.4(b) show the false positive rate when each of the aggregation algorithms are applied to a network having a regular tree topology for different workload distributions. Figure 5.4(a) depicts a scenario where workload is generated using uniform distribution whereas Figure 5.4(b) shows the behavior of the algorithms when zipfian distribution is used. In both scenarios, the local aggregation methods are heavily outperformed by the other two as a result of performing aggregation based on local switch state as compared to the two flavors of the filter aggregation algorithm which consider a holistic view of the network for filter aggregation for both distributions. The amount of false positives in the network on using, especially, LA-B for

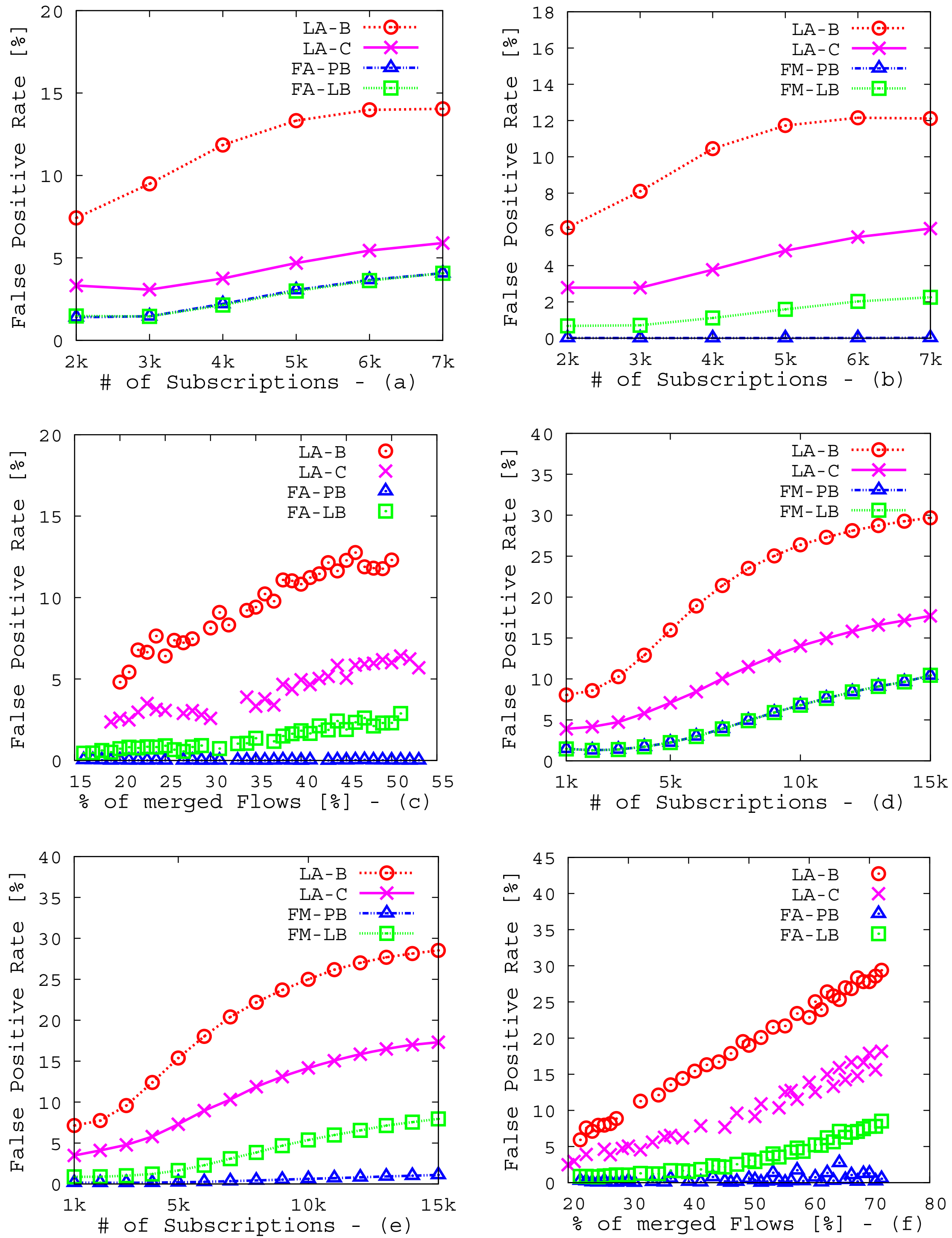


Figure 5.4: Performance Evaluations: False Positive Rate

aggregation clearly shows the importance of having a more refined algorithm for aggregation. The plots show that, in every scenario, LA-C outperforms LA-B. This is expected as LA-C selects a merge based on calculated local aggregation cost as opposed to LA-B which makes aggregation decisions with least overhead. In case of uniform distribution in Figure 5.4(a), we see that the performances of FA-LB and FA-PB are almost equivalent. The main difference between FA-LB and FA-PB is that FA-PB analyses each of the event packets to determine the amount of false positives along each path. By contrast, FA-LB determines the amount of traffic on each path and then estimates the amount of false positives while considering the traffic to be distributed uniformly over the advertised subspace for that path. As a result, the two methods behave very similarly for uniform distribution as the estimate of false positives is almost identical to the actual false positives in the network. However, for the same reason, the advantage of FA-PB over FA-LB is very apparent in Figure 5.4(b) where FA-PB clearly outperforms FA-LB as the decision-making process in FA-PB considers the exact nature of published events that follow a zipfian distribution. In fact, when using FA-PB, the false positive rate is reduced by up to 99.9% as compared to LA-B and is almost non-existent even on aggregating a large number of subscription filters in the system, highlighting the effectiveness of the filter aggregation algorithm proposed in this chapter. This is mainly because, with zipfian distribution, FA-PB can take efficient decisions to merge flows which do not experience too much traffic and, therefore, less false positives while preserving flows relevant to event traffic hotspots. In fact, we depict the false positive rate vs total percentage of merged flows in the network in Figure 5.4(c) for zipfian distribution. This graph shows the impact of flow aggregation on false positives. Of course, more the number of merged flows (i.e., aggregation) in the network, more is the false positive rate. The plots show that even when a large percentage of flows are aggregated, it results in very low false positives for FA-PB. In fact, even when over 50% of flows are merged, the false positives in the network are negligible implying that the TCAM constraint does not adversely impact the system if FA-PB is used for aggregation. The performance of FA-PB is closely followed by FA-LB, followed by LA-C, and finally LA-B.

To show the effectiveness of the proposed algorithms irrespective of the type of topology, we also conducted experiments on a random topology as depicted in Figure 5.4(d) and Figure 5.4(e). Here too, we see the same behavior of the algorithms as in the tree topology. As before, FA-PB and FA-LB perform similarly in the case of uniform distribution as depicted in Figure 5.4(d). In the case of zipfian distribution, again, on performing aggregation using FA-PB, the false positives in the network are almost negligible despite aggregating a large number of subscription filters as depicted in Figure 5.4(e). We, also, show a graph to depict false positive rate vs total percentage of merged flows in the network in Figure 5.4(f) for the random topology when zipfian distribution is used. As can be seen in the figures, the comparison of performance in terms of false positive rate of the various algorithms is similar to that for the other topology which implies that the behavior of the algorithms is not specific to a type of

topology.

5.4.3 Comparing Runtime Overhead

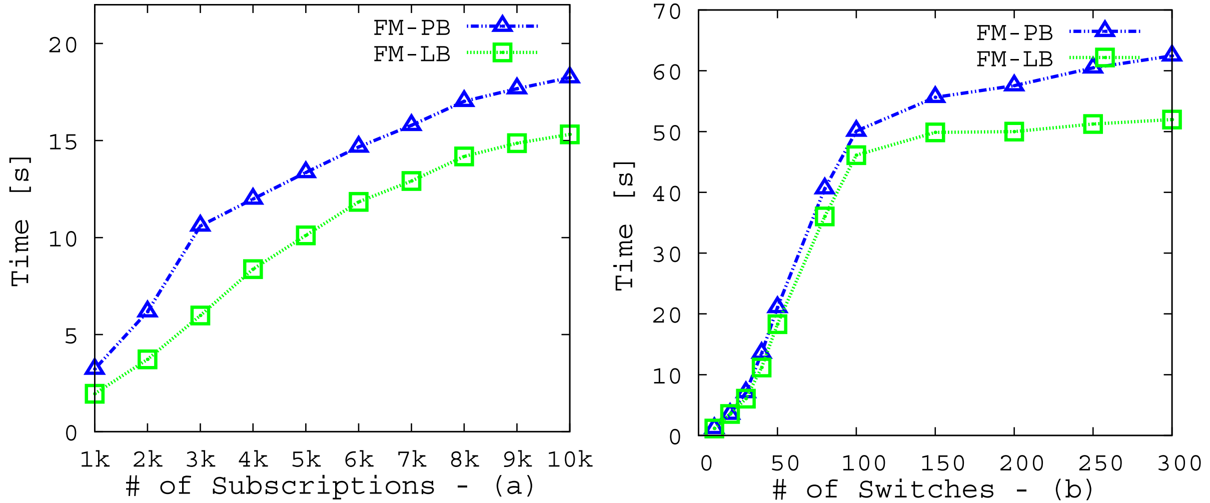


Figure 5.5: Performance Evaluations: Runtime Overhead

The effectiveness of the algorithms w.r.t. bandwidth efficiency is clear from the above discussion. However, where FA-PB outperforms the others in bandwidth efficiency, the others come with lower overhead. The higher overhead in FA-PB is not only due to the fact that published events need to be collected from the publisher but also due to a higher runtime overhead than the others. We confirm the same in our next set of experiments depicted in Figure 5.5(a) where we compare the two flavors of filter aggregation algorithm. We measure the runtime overhead with increasing number of subscriptions. Again, note, FA-PB has a sampling factor of 1 in this set of experiments. As depicted in the figure, FA-PB has a higher runtime overhead than FA-LB consistently as it additionally considers event traffic patterns for cost calculation. Our evaluations also show that the average runtime overhead for LA-C is merely *500* microseconds which further goes down for LA-B to a mere *200* microseconds on a switch. So, we see that there is a trade-off between quality and overhead as the improvement in one adversely affects the other.

We also evaluate the runtime overhead of the two flavors of filter aggregation algorithm with increasing topology size. In this experiment we keep the number of publishers and subscribers fixed and expand the topology in terms of number of switches. Of course, more the number of switches more will be the overhead for both FA-PB and FA-LB as the cost calculation has to be done over more switches with each calculation considering longer paths (more upstream filters). Such a behavior is visible in Figure 5.5(b) where

the overhead for both FA-PB and FA-LB increases with increasing number of switches. Again, FA-PB has higher overhead than FA-LB due to the aforementioned reasons.

5.4.4 Impact of Sampling Factor

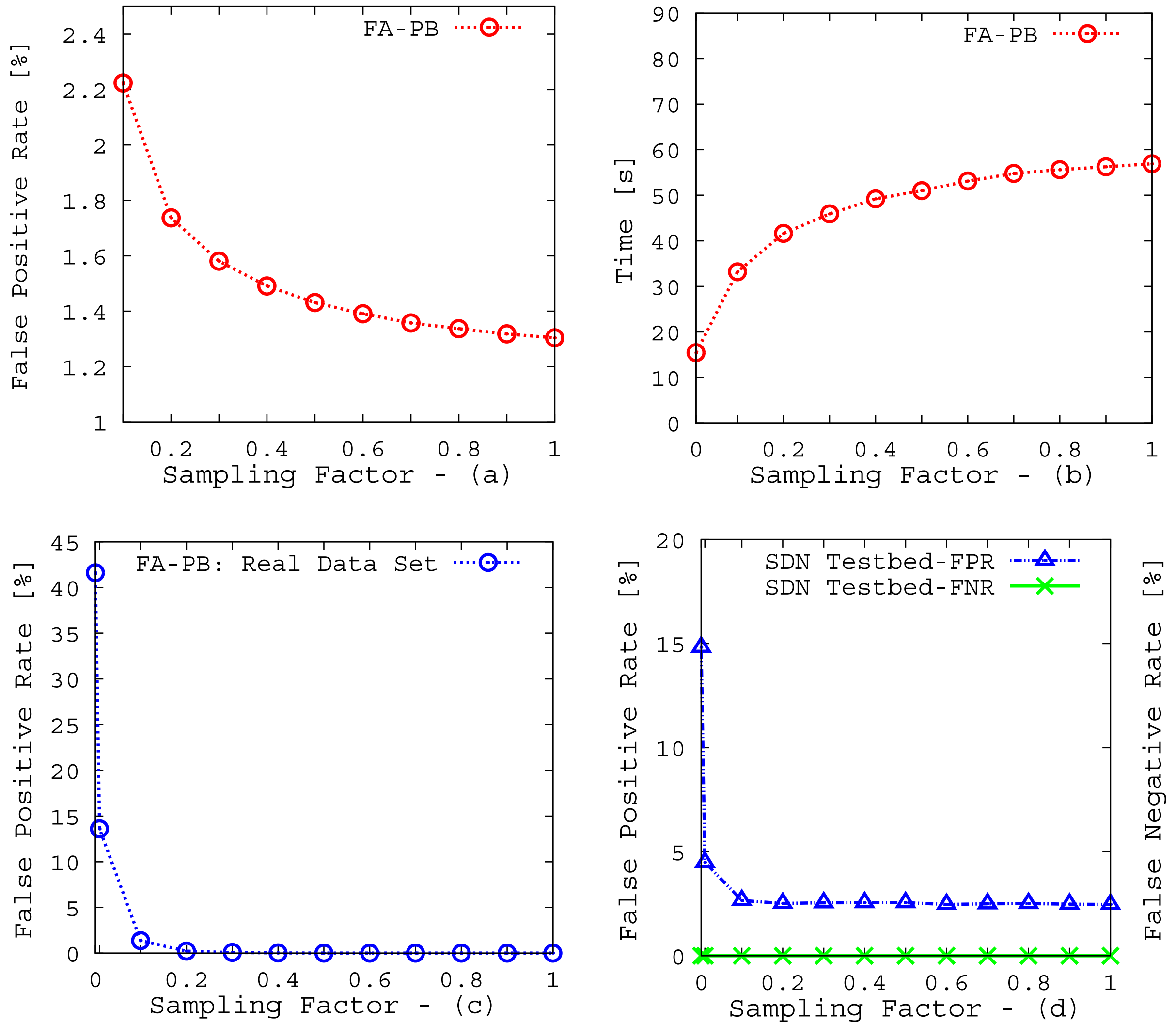


Figure 5.6: Performance Evaluations: Impact of Sampling Factor

To reduce the overhead of collecting published events and cost calculation of FA-PB, we introduced the sampling factor (i.e., sfr) in Section 5.3. In the next set of experiments, we show the behavior of our system when subjected to various sampling factors. Figure 5.6(a) plots the false positive rate with increasing value of sfr for zipfian distribution. As expected, more the value of sfr, fewer are the false positives as FA-PB is more accurate in its cost calculation when it considers more past events.

However, higher the sampling factor of FA-PB, higher is the overhead as depicted in Figure 5.6(b) where we plot the runtime overhead for increasing values of sfr.

To ensure that our aggregation algorithm is effective in realistic scenarios, we conducted experiments to show its behavior on real-world stock data. Figure 5.6(c) plots the false positive rate with increasing sampling factor for the real-world data set. The plot clearly shows that, even for a sampling factor of just 0.4, the network false positives due to aggregation are almost non-existent. These evaluation results further highlight the applicability and efficiency of the algorithm presented in this chapter.

For our next set of experiments, we measure the false positives and false negatives at the subscribers when the aggregation algorithm is deployed on the real SDN testbed *SDN-t-hswitch*. So, Figure 5.6(d) plots the false positive rate when FA-PB aggregates flows for increasing sampling factors for workload generated using zipfian distribution. The graph shows that on the real SDN testbed, the algorithm behaves as expected and the false positive rate decreases rapidly with increasing sampling factor. Moreover, we also measure the false negative rate (FNR) in the system. Figure 5.6(d) shows that, on the real testbed, our mechanisms to ensure data plane consistency successfully avoid packet drops in the network even while the data plane is being modified by the filter aggregation algorithm. As a result, there are no false negatives in the network due to the presented aggregation algorithm.

5.4.5 Dynamic Behavior

In our next set of experiments, we evaluate the performance in terms of false positive rate of our system in a dynamic environment. We progressively introduce subscriptions in the system and apply our aggregation algorithm for handling dynamics. So, in Figure 5.7(a), in general, the basic local aggregation approach (LA-B) is applied whenever switches exceed their capacity on introduction of a new subscription as explained under handling dynamics in Section 5.3. Additionally, FA-PB is employed after every 3000 subscriptions as depicted in Figure 5.7(a). The figure shows that the false positive rate gradually increases with more and more subscriptions when LA-B is used till FA-PB is performed which makes the false positives in the system almost negligible. Again, the false positive rate keeps increasing on using LA-B till the next application of FA-PB. We, also, plot the behavior of the system if only LA-B is employed. This clearly shows the amount of false positives reduced in the system at every step due to the intermittent application of FA-PB.

Figure 5.7(b) shows a similar dynamic scenario where this time LA-C is used for local aggregation whenever switches exceed their capacity. We see that even though LA-C keeps the false positive rate lower than when LA-B is used, the intermittent use of the filter aggregation algorithm, clearly improves bandwidth efficiency significantly in the system.

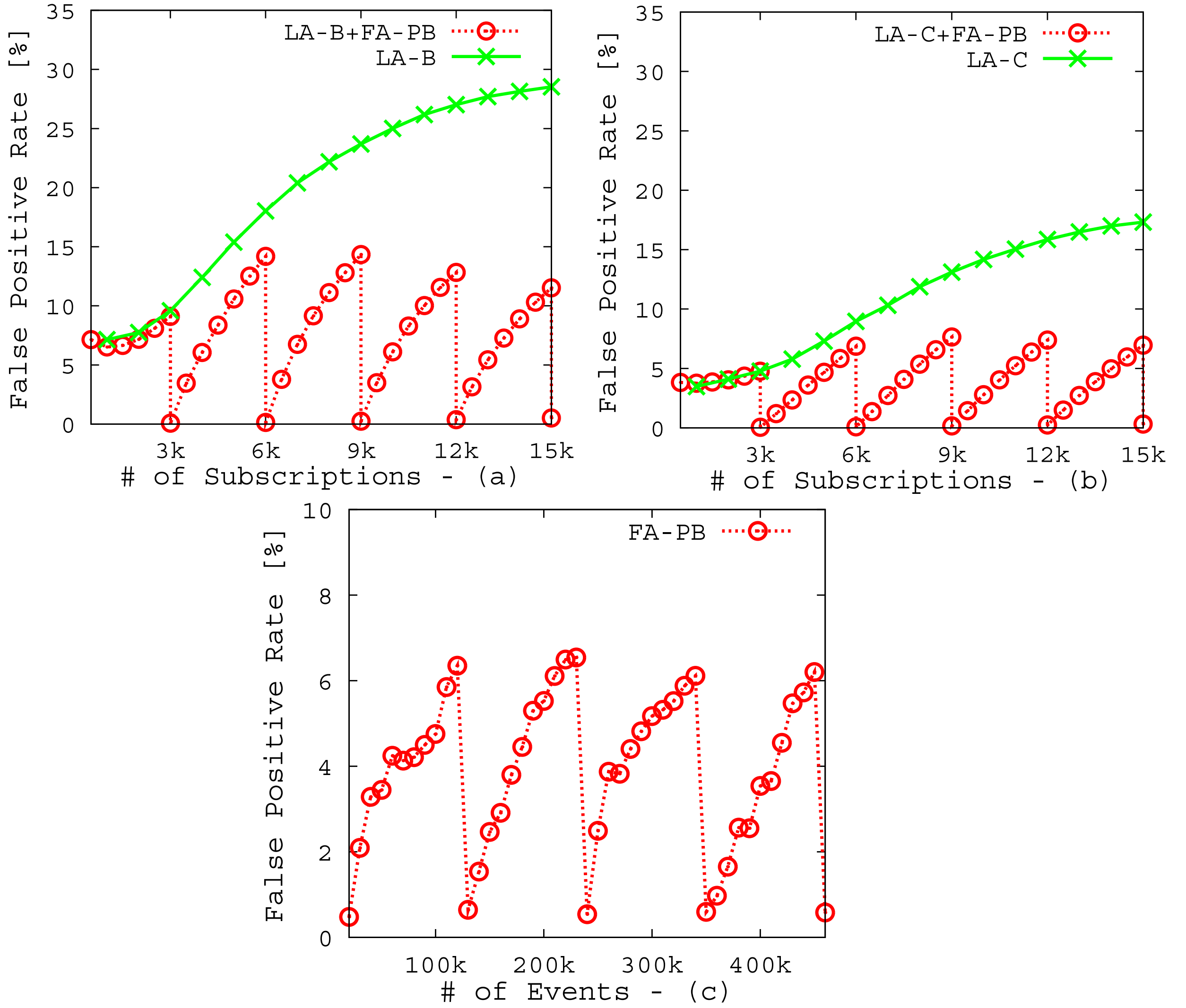


Figure 5.7: Performance Evaluations: Dynamic Behavior

In our final set of experiments, we show the impact that change in event traffic distribution has on the false positive rate of a system and the manner in which FA-PB adapts to these changes to reduce false positives. In these experiments, in a system with a constant number of subscriptions, events are generated using a zipfian distribution where the hotspots are randomly shifted by up to 10% every 10,000 events. So, Figure 5.7(c) depicts the behavior of the system when the event distribution gradually changes with time and FA-PB is employed periodically. The figure shows that, as a result of gradually shifting the hotspots, the aggregation decisions taken by the latest execution of FA-PB are no longer ideal and the false positive rate increases. The farther the hotspots are shifted from their original positions, the higher is the false positive rate until FA-PB is executed again such that aggregation decisions are taken

based on the most recent event distribution. FA-PB reduces the false positives in the system to a negligible amount which, however, again gradually starts rising with the gradual change in event distribution. This experiment highlights the importance of the periodic execution of the filter aggregation algorithm in a system with dynamically changing event distribution.

5.4.6 Discussion

Our evaluation results show the huge potential of the proposed filter aggregation algorithm in improving bandwidth efficiency of the system while being restricted in TCAM capacity. When evaluated with different data distributions and network topologies, in every scenario, despite performing significant number of aggregations in the network, the filter aggregation algorithm proves to be extremely bandwidth-efficient with a very low network false positive rate. In fact FA-PB outperforms all the other proposed aggregation techniques and reduces false positives introduced in the network due to aggregation by a baseline approach (i.e., LA-B) by up to 99.9%. The performance of FA-PB is closely followed by FA-LB, which is followed by LA-C, and finally LA-B. Of course, more bandwidth-efficient the aggregation technique, more is the runtime overhead. As a result, the runtime overhead of both variants of the filter aggregation algorithm is higher than that of both variants of local aggregation with FA-PB having the highest runtime overhead and LA-B having the lowest. We, also show the impact of sampling factor on the performance of FA-PB where an increase in sampling factor reduces false positive rate but increases the runtime overhead. The impact and practicality of the filter aggregation algorithm on a real SDN testbed and on a real data set is also shown in the evaluation results. Finally, the dynamic behavior of the proposed aggregation techniques with increasing subscriptions and changing event traffic patterns clearly impresses upon their importance and applicability in an SDN-based pub/sub middleware.

5.5 Related Work

In the past couple of decades, a significant amount of research has been dedicated to broker-based middleware implementations of content-based pub/sub [CRW01, JJE10, Müh02] that focus on achieving scalability. In this context, techniques for subscription summarization that include subscription covering [CRW01] and subscription merging [Müh02] are widely employed to realize scalable systems. Subscription summaries not only help in filtering out of events from the parts of the broker network without interested users but also ensure forwarding of new subscriptions only to brokers which previously do not receive subsuming (or covering) subscription summaries. So, these systems, primarily, use subscription summarization to reduce unnecessary message overhead in the broker network. Also, much effort [CS04, JJE10, CFMP04] has been

devoted to reduce the overhead of maintenance of these subscription summaries but from the context of efficiently handling dynamically changing subscription requests in the broker network. Implemented in the application layer, these systems do not need to address the problems of limited hardware space in the network layer to accommodate the subscription filters and subscription summarization is primarily performed to achieve bandwidth efficiency in broker networks.

In general, the problem of limited flow table entries in TCAM of SDN-compliant switches is well-known and much researched [KARW16, KLRW13, VPMB14, GXC+15]. For example, Katta et al. [KARW16] in CacheFlow use rule dependencies to cache the more popular flows on the limited TCAM, while the remaining traffic is left to rely on software. As is the case with filtering in software, here, too, the performance of the traffic forwarded by the software switch will suffer. Significant amount of work in literature deals with optimizing rule placement in a software-defined network [KLRW13, NSBT14, KHK13]. For example, OneBigSwitch [KLRW13] uses endpoint policy and routing policy to aggregate sets of rules in order to take decisions on distributing them over network switches. However, it is incapable of handling scenarios where the rule sets are larger than the aggregate table size.

Also, considerable amount of work has been done in the lines of compacting the representation of flow rules for the purposes of reducing TCAM space [LMT10, MLT12]. Although, these systems target efficient network provisioning and compressing rules on a switch, the proposed solutions are not applicable to our problem in the context of content-based routing. Also, systems, such as SmartTime [VPMB14], use an adaptive timeout technique to pro-actively evict flow rules while ensuring that there is minimum TCAM misses. Since, we consider a system that does not allow false negatives and filters can only be removed on account of an unsubscription or unadvertisement, a timeout-based heuristic is not ideal for our problem.

5.6 Conclusion

In this chapter, we design techniques to mitigate the problems associated with limited TCAM space in an SDN-based publish/subscribe middleware. We propose, implement, and thoroughly evaluate a filter aggregation algorithm that not only respects TCAM space limitations on individual switches but also successfully minimizes false positives in the network, despite merging of flows. To this end, we introduce two flavors of this algorithm and compare various aspects of their performance. Our evaluations include experiments on a real SDN testbed and with real-world workload. Evaluation results show that the designed filter aggregation algorithm reduces the false positives, introduced in the network when a baseline approach is used for aggregation, by up to 99.9%.

Chapter 6

Scaling the Control Plane

As discussed in Chapters 1 and 2, preserving the benefits of SDN in a highly dynamic environment in PLEROMA is rather challenging. For example, applications such as financial trading, traffic monitoring, online gaming and electronic auctions are not only latency sensitive but also very dynamic in terms of number of application users and their interactions. As a consequence, the control plane has to engage in very frequent network topology updates, and this is where the traditional design of SDN, consisting of a single controller instance, does not scale well. The bottleneck at a single controller instance results in increased response times to requests for network updates, rendering the middleware less responsive to dynamics.

Not surprisingly, in the past, research efforts [TG10, KCG+10, HYG12, DHM+13, DHM+14] propose a distributed implementation of the control plane, which essentially operates as a logically centralized controller. A distributed control plane hosts multiple controller instances capable of performing concurrent network updates, thus improving responsiveness and throughput of the control plane. While increasing the rate at which network reconfigurations can be realized, it has been well established in literature that a distributed control plane is subject to inconsistencies [LWH+12, BRKB13]. Inconsistencies may arise due to unsynchronized global network state views at the distributed controller instances. Every controller instance maintains a data structure representing the view of the global network state. This implies that the network acts as a shared resource. Depending on the nature of the application, network updates are made by each controller instance based on the state of its local data structure. Inconsistencies between the global network state maintained at each controller may lead to incorrect application-specific behavior. Performing updates based on a stale copy of the network view may result in routing loops and black holes in the data plane. Existing literature [LWH+12] shows the severity of degraded application performance in the absence of strong consistency.

To ensure strong consistency of network state, synchronization mechanisms must be

employed among all controller instances such that all network updates are coordinated. Synchronization involves state distribution and, depending on the desired level of consistency, various classical approaches available in the field of distributed systems may be used for the same [KCG+10, BRKB13]. However, synchronization techniques always come with a cost that may compromise responsiveness to data plane requests. For instance, according to literature [KCG+10], synchronization techniques using transactional persistent database backed by a replicated state machine yields severe performance limitations for applications requiring frequent network updates.

In the aforementioned systems, the significant overhead in synchronization cost can be attributed to the attempt of designing a general-purpose distributed control plane capable of supporting any SDN use-case. However, to render the control plane of PLEROMA more responsive, it is worth exploring how application-aware control distribution can help to reduce this overhead. So, in this chapter, we illustrate the benefit of application-aware control distribution in the context of our PLEROMA middleware, to allow for increased responsiveness while ensuring strong consistency (in the context of control plane) even in the presence of failures.

In this chapter, we propose to scale the control plane by introducing multiple controllers, which may reside on a single physical machine with a multi-core architecture (i.e., vertical scaling) or on separate physical machines in a physically distributed setting (i.e., horizontal scaling), to improve the responsiveness and throughput of network updates handled by the PLEROMA middleware. We design two approaches – shared everything and shared nothing – each reaping the benefits of vertical and horizontal scaling, respectively. Moreover, we address limitations of SDN-compliant switches w.r.t. the rate at which flow updates are performed, again by exploiting application-awareness. Our evaluations show that application-aware control distribution allows to significantly increase responsiveness to control requests for both vertical and horizontal scaling while ensuring control plane consistency.

6.1 Distributed Control Plane - System Architecture

In general, the topology reconfiguration efforts are significant in PLEROMA (cf. Chapter 2). As discussed earlier, in a scenario with frequent concurrent control requests from multiple participants, a design with a single SDN controller will result in very poor control plane responsiveness. Here, we define *response time* as the time from the issuance of a control request by a participant till the completion of all topology reconfiguration associated with this request by the control plane. For example, the response time to a subscription is the time elapsed from the issuance of the subscription until the subscriber starts receiving events. As a single controller processes each control request sequentially, the response time increases significantly in the face of high dynamics. This problem motivates us to introduce multiple controller instances in the control plane, enabling concurrent processing of control requests.

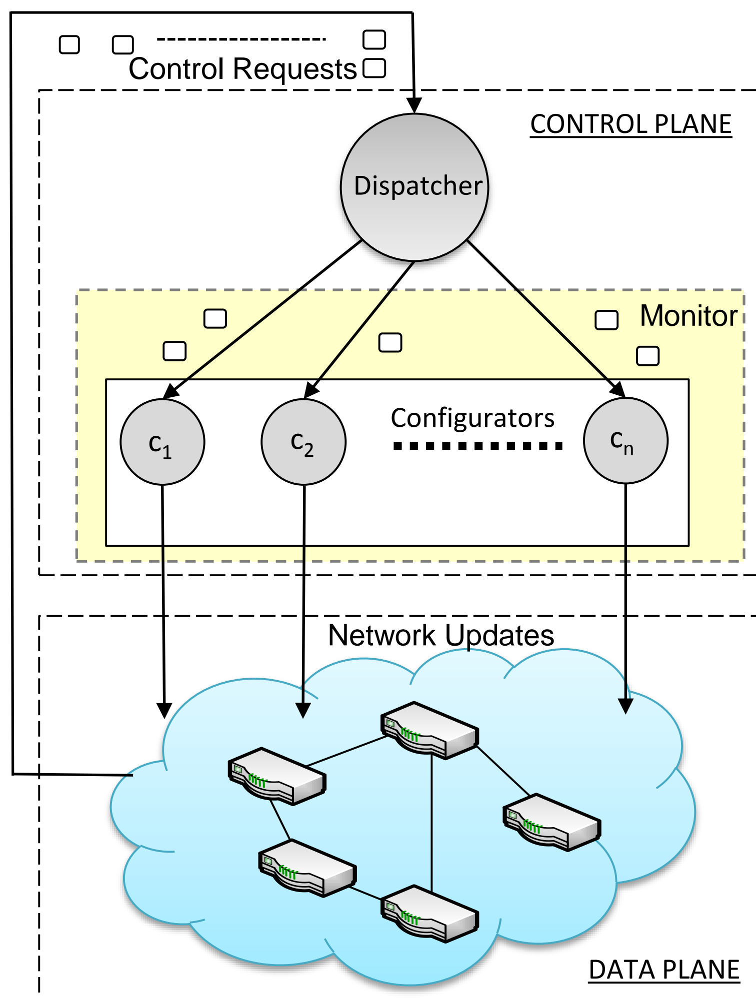


Figure 6.1: System Architecture

Figure 6.1 depicts a two-tiered architecture in the control plane; a *dispatcher* collects control requests from publishers and subscribers in a software-defined network, and a set of components, known as *configurators* (denoted by C), processes these requests and carries out network updates accordingly. SDN allows the *dispatcher* and the *configurators*, residing in the control plane, to acquire a global view of the entire network and configure it as needed. The *dispatcher* serves as the entry point to the control plane. It collects all data plane control requests and forwards them to the *configurators*. The *configurators* serve as the main workers that are capable of modifying the state of every switch in the network. Each of them receives control requests forwarded by the *dispatcher* and processes them as described in Chapter 2.

The *monitor* is an additional component connected to the *configurators* and the *dispatcher*. It plays an important role in maintaining load statistics of each *configurator*,

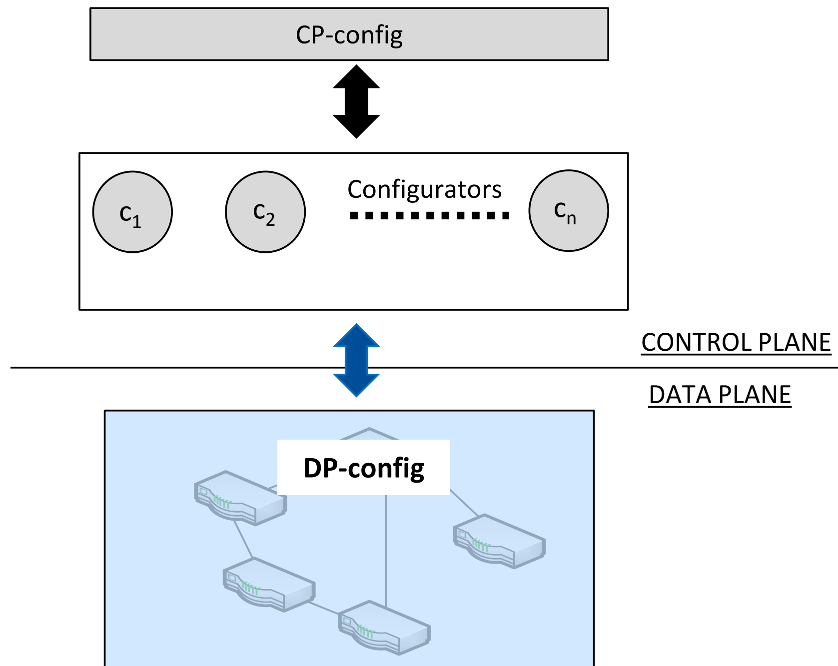


Figure 6.2: Concurrent Access to CP-config and DP-config

which contributes to improved system performance. The relevance of the *monitor*, in the context of our designed middleware, will be explained in details later in this chapter.

In this chapter, we scale the control plane both vertically and horizontally. Vertical or horizontal scaling is mainly achieved by scaling up or out the *configurators*. Here, vertical scaling means hosting multiple *configurator* instances on multiple cores of a single machine. In contrast, horizontal scaling involves hosting multiple *configurator* instances on cores of physically distributed machines. Irrespective of the scaling type, the introduction of multiple *configurators* implies concurrent processing of requests for improved responsiveness which in turn raises questions on control plane consistency.

In the subsequent sections, we first discuss control plane consistency in the context of pub/sub middleware and identify conflicting actions that may induce inconsistencies (cf. Section 6.2). Afterwards, we present approaches for vertical and horizontal scaling of the control plane that ensure strong consistency by enabling concurrency control for conflicting actions with low synchronization overhead (cf. Section 6.3).

6.2 Control Plane Consistency in Pub/Sub

Please recall from Chapter 2 that the network configuration is maintained both at the data plane and the control plane of a software-defined network. On the one hand, the network configuration at the data plane, known as *DP-config*, is maintained implicitly

as a result of pub/sub flows deployed in the TCAM of hardware switches. On the other hand, the control plane network configuration, known as *CP-config*, is maintained by the controller and serves as a reflection of DP-config. CP-config is maintained by the controller so that it does not need to query the switches in the data plane and read their states for processing every control request. When a control request arrives at a controller, since the controller assumes CP-config to be identical to DP-config, it uses CP-config to read existing flows and decide on flow changes. On taking a decision, the controller sends the new flow changes to the hardware switch, resulting in a change in DP-config. Meanwhile, the controller also performs these flow changes in the CP-config to ensure that it remains consistent with DP-config. While in Chapter 2, we mentioned that the two configurations are kept consistent in the aforementioned manner, in this chapter (cf. Section 6.4), we discuss the protocol for ensuring consistency between CP-config and DP config (even in the presence of failures) in more details.

In the context of a distributed control plane with multiple *configurators*, each *configurator* processes each control request similar to the process discussed in Chapter 2. However, we need to consider that, in order to concurrently process control requests, all *configurators* will need to concurrently read from CP-config. Moreover, they will need to perform concurrent changes to DP-config and subsequently CP-config (to keep CP-config consistent with DP-config). Such concurrent access to the configurations by *configurators* is depicted in Figure 6.2.

As a result, two important problems have to be addressed to ensure control plane consistency in PLEROMA. These problems are

(i) maintaining consistent network configuration at the control plane (i.e., CP-config) in the presence of concurrent updates by multiple *configurators*.

(ii) keeping CP-config consistent with DP-config (even in the presence of failures).

In this section and Section 6.3, we strictly focus on the first problem and address the second problem in Section 6.4. For simplicity and without loss of generality, we discuss the first problem only with respect to CP-config as consistent maintenance of CP-config, in the face of concurrency, eventually results in a consistent DP-config.

In more detail, the *configurators* execute the same control logic and operate on the same CP-config concurrently. As discussed in details in Chapter 2, in order to ***process a control request***, a *configurator* performs a ***set of actions*** consisting of ***operations*** on switches along the paths between publishers and subscribers. More specifically, at each switch along the paths, the *configurator* performs an ***action*** that consists of an ordered sequence of three ***operations***. The three operations include

(i) reading flows from a switch,

(ii) deciding on the changes to be made to the flows, and

(iii) writing these changes back to the switch.

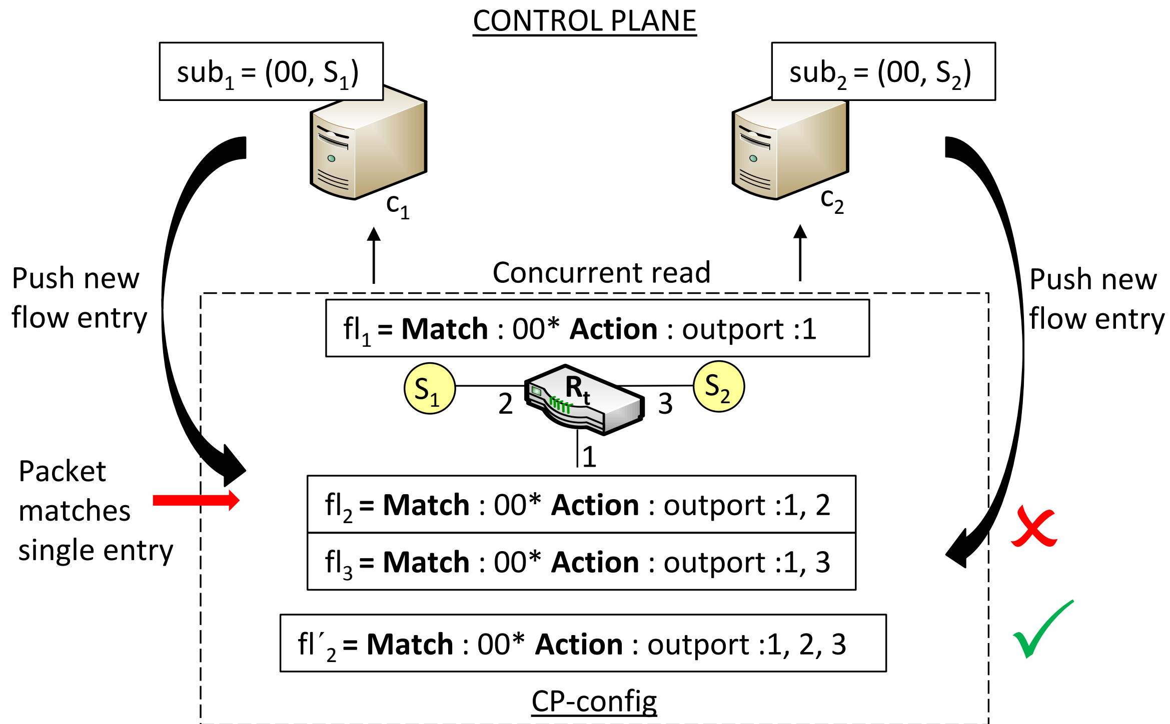


Figure 6.3: Control Plane Inconsistency

The concurrent execution of such actions by two or more *configurators* can result in their sequences being interleaved. This raises concurrency related issues resulting in false negatives or false positives in the system.

Figure 6.3, depicts an example of a simple case of false negatives at a subscriber due to the interleaving of sequences of operations constituting two actions and belonging to two *configurators*. Let us suppose that two overlapping subscription requests sub_1={00} from subscriber S_1 and sub_2={00} from subscriber S_2 are simultaneously dispatched to *configurators* c_1 , $c_2 \in C$, respectively. Both follow the aforementioned request handling process and perform actions on relevant switches. We specifically focus on the terminal switch R_t which already has a flow, fl_1, to match event packets for subspace {00} (cf. Figure 6.3). We consider a case where both *configurators* perform concurrent read on this switch in CP-config. On reading the state, c_1 and c_2 independently decide on required flow updates and replace the existing flow (fl_1) by adding two new flows fl_2 and fl_3, respectively (cf. Figure 6.3). As a consequence, there now exists two flows with the exact same match field but with different IS at R_t. Since deploying flows on CP-config implies deploying them on DP-config, now, if an event packet lying in subspace {00} arrives at R_t in the data plane, it follows the instruction set of either fl_2 or fl_3, but never both as the matching of a packet at a switch is terminated as soon as the first match is found. In either case, one of the two subscribers is affected by false

negatives compromising correctness of the system. This can be avoided if mechanisms to guarantee strong consistency at the distributed control plane are employed.

Clearly, false negatives at a subscriber in Figure 6.3 occurred because flows fl_2 and fl_3, concurrently added by c_1 and c_2, are in aforementioned partial flow containment relation (i.e., $\gtrapprox$), which essentially results in updating the same flow in R_t. In general, concurrent updates of the flows with containment relations (i.e., $\succ$ or $\gtrapprox$) have an effect of one of the updates being overwritten by the other.

While understanding the above mentioned concurrency issues, we define conflicting actions in the PLEROMA middleware as follows.

Two different actions are in conflict if (i) both of them access the same switch and (ii) both of them affect flows that are bound by the flow relations, i.e., complete containment ($\succ$) and partial containment ($\gtrapprox$).

If no precautions are taken, the concurrent execution of conflicting actions can result in inconsistencies. This is where we draw parallels between our defined action and the very widely used transaction from the field of databases [BHG86, HR83, BG81, Ske81]. In transaction processing, it is widely known that every serial execution of transactions is defined to be correct (assuming the transactions themselves are correct) [BHG86, BG81]. In fact, it has been established that 'an execution (concurrent) is serializable if it is computationally equivalent to a serial execution, that is, if it produces the same output and has the same effect on the database as some serial execution. Since serial executions are correct and every serializable execution is equivalent to a serial one, every serializable execution is also correct' [BG81]. As a result, in order to ensure correctness in our distributed control plane, all concurrent conflicting actions must be serialized.

6.3 Scaling Approaches

Having identified conflicting actions, we propose two approaches—shared everything and shared nothing—that scale the control plane both vertically as well as horizontally while avoiding concurrent processing of conflicting actions.

6.3.1 Shared Everything Approach

The shared everything approach (SEA) works on the principle that all *configurators* share CP-config among themselves. This implies that all of them read from as well as write to every switch in CP-config. Section 6.2 explained the undesirable consequences of such concurrent access of shared state which means that the SEA approach must employ certain additional mechanisms for concurrency control. SEA uses a locking mechanism that allows a *configurator* to acquire exclusive access on CP-config at various

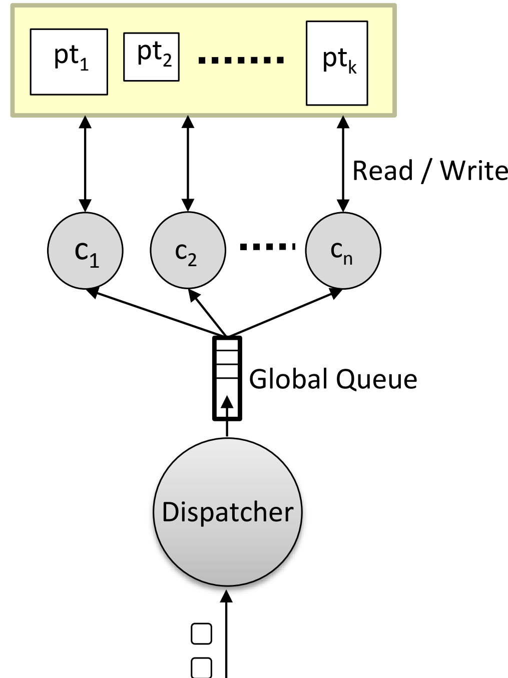


Figure 6.4: Shared Everything Approach

granularity levels. This means that no other *configurator* can access the locked part of CP-config unless the *configurator* holding the lock relinquishes it. Locks can be held at different levels of granularity in CP-config. In the absence of application knowledge, a plausible strategy is to assign locks at the granularity of switches. For example, with the advent of a subscription, a *configurator* can determine the paths between the associated subscriber and all relevant publishers, acquire locks on all switches in these paths, perform the necessary actions (i.e., read, decide on flow changes, deploy changes on the switches), and finally release the locks. Acquiring locks at switch level, however, would imply that no other *configurator* can execute an action on a locked switch even if its action does not conflict with the current action being executed. So, with respect to our definition of conflicting actions (cf. Section 6.2), locking at switch-level may not be ideal.

Here, we propose an application-aware method that uses knowledge of advertisements and subscriptions to control the granularity at which CP-config can be accessed concurrently. Since the *dz*s representing the subscriptions/advertisements (in control requests) are directly mapped to flows added to switches (cf. Chapter 2), two control requests where one *dz* covers or is identical to the other (overlapping subspaces in Ω) yield flows related ($\succ$ and $\gtrapprox$) to each other. This means that concurrent processing of overlapping control requests at a switch will result in conflicting actions and must be ordered sequentially. Control requests with non-overlapping subspaces in Ω, how-

ever, can undergo concurrent processing without any issues. For example, concurrent processing of two subscriptions $\{00\}$ and $\{000\}$ which results in state modification of at least one common switch will lead to incorrect system behavior as $\{00\} \succ \{000\}$. However, two unrelated subscriptions, $\{00\}$ and $\{11\}$, can be processed concurrently by two *configurators* without any issues as processing will not yield any related flows. This directly leads us to the idea of partitioning the event space in a disjoint way such that flows corresponding to different partitions in Ω are maintained in separate CP-configs. This enables concurrent processing of disjoint control requests that operate on different CP-configs. Locking would only be required at the level of a CP-config to ensure sequential processing of overlapping control requests as only overlapping control requests can result in conflicting actions.

So, we divide the event space into multiple disjoint, continuous partitions. A partition is nothing but a subspace in Ω and may be represented in the same way, i.e., by a dz. Disjoint event space partitioning may yield equal or unequal partitions depending on the partitioning criteria. However, it is important to note that, in any case, the partition set, denoted by $\mathbb{PT}$, is non-overlapping and fully covers Ω. Mechanisms for content or event space partitioning have been extensively researched in various fields of computer science [Van91, WQA+04] and will not be discussed further in this thesis. Henceforth, we assume that $\mathbb{PT}$ consists of k partitions and $k >> n$ where n denotes the total number of *configurators*.

The middleware maintains a set of independently configurable CP-configs (denoted by CP) having a one-to-one mapping with these partitions. This results in the creation of k CP-configs where each configuration, $cp \in CP$, is represented by the dz of the corresponding partition. Again, each switch in each cp contains only those flows that are associated with the event space partition that this configuration represents. This implies that the spanning tree maintained by CP-config is responsible for the dissemination of only a set of events that lies in its designated subspace. In the remaining part of this chapter, a CP-config ($cp_i \in CP$) is considered to be synonymous with a partition ($pt_i \in \mathbb{PT}$).

We focus on an SEA approach where locking is carried out at the level of a $cp \in CP$ which essentially means locking a set of flows across all switches that correspond to a partition in Ω. This ensures concurrent access of unrelated flows on the same switch. Each *configurator* maintains a pointer to each CP-config/ partition. Figure 6.4 illustrates the same with n *configurators*, c_1,.., c_n, operating on k partitions, pt_1,..., pt_k. SEA ensures serial processing of requests within a single partition while allowing concurrency otherwise. This implies serial execution of conflicting actions.

The *dispatcher* collects all control requests from the data plane and adds them to a global queue accessible to all *configurators*. But before adding them, it performs an additional step to prepare the requests for further processing. Let us denote the dz representing a control request by dz_c and that representing any partition pt_i by dz_{pt_i}. When a control request arrives at a *dispatcher*, it is processed by the *dispatcher* in two

ways depending on whether (i) $dz_{pt_i} \succeq dz_c$ or, (ii) $dz_c \succ \{dz_{pt_i},...,dz_{pt_j}\}$. In the first case, the *dispatcher* simply adds the request to the global queue as the request is contained by one partition and affects a single CP-config. However, the second scenario portrays a case where the control request subspace spans more than a single partition. Under such circumstances, the *dispatcher* splits up the request into multiple dzs depending upon the nature of the partitions and adds these partial requests to the queue. For example, if we consider a system with 4 partitions—00, 01, 10, 11—and a request with dz $\{001101\}$ arrives at the *dispatcher*, the *dispatcher* immediately adds the request to the global queue as $\{00\} \succ \{001101\}$. However, if the request corresponds to $\{0\}$, the *dispatcher* first splits it up into two partial requests $\{00\}$, $\{01\}$ and then adds them to the queue as $\{0\} \succ \{00, 01\}$. Consequently, two CP-configs are reconfigured for this single request. Processing of a control request is considered complete only when all its partial requests have been processed.

As soon as a request is available in the global queue, an idle *configurator* tries to dequeue it and process it. However, before dequeuing the request, it would first need to acquire an exclusive lock on the CP-config to be reconfigured for this request. Since a request is already preprocessed by the *dispatcher*, it will always correspond to a single CP-config, requiring the *configurator* to acquire the lock on this configuration alone. If it is possible to acquire the lock, the *configurator* dequeues the request from the global queue and proceeds with reconfiguration of the locked CP-config. Reconfiguration follows the usual mechanisms discussed in Chapter 2. Once all actions corresponding to this request are performed, the *configurator* releases the lock on the configuration. On the contrary, if a *configurator* is unable to acquire a lock on a CP-config for a particular request, it simply continues traversing the queue for a request belonging to a partition on which it can acquire a lock till it reaches the end of the queue. So, a *configurator* does not get blocked if requests affecting unlocked partitions are available in the queue for further processing. It should be noted that a *configurator* ensures that, if it acquires a lock on a partition, it always processes the first request waiting in the queue for that partition. This ensures fairness of request processing at least within a partition. SEA enables the *configurators* to actively look for a request to process as soon as they are idle, resulting in implicit load balancing among them. Also, multiple partitions where $k >> n$ allows for a possibility of a certain degree of concurrency despite ensuring strong consistency.

The advantages of using such an approach for increased responsiveness in a vertically scaled control plane are significant. This is because vertical scaling works with a shared memory architecture where sharing of multiple CP-configs does not cause much overhead. However, horizontal scaling implies repeated remote access of multiple CP-configs by every *configurator*. This involves transfer of large amount of data over the control network, resulting in severe performance limitations. Not surprisingly, SEA cannot match up to the requirements of a horizontally scaled control plane. To overcome these limitations and exploit the benefits of horizontal scaling, we propose

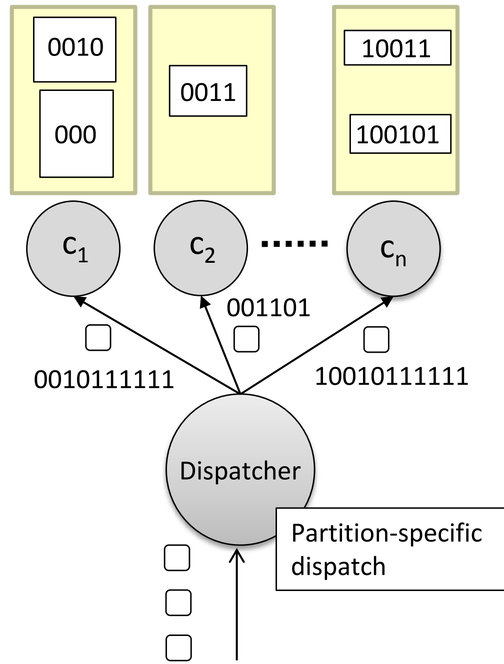


Figure 6.5: Shared Nothing Approach

the shared nothing approach which is the subject of discussion in the remaining part of this section.

6.3.2 Shared Nothing Approach

The shared nothing approach (SNA) also operates with multiple disjoint CP-configs or partitions. However, in this approach, each partition is assigned exclusively to exactly one *configurator*. To ensure consistency, each *configurator* is restricted to performing reconfigurations on its assigned partitions only. So, two or more *configurators* may process different requests concurrently as they operate on completely different subspaces in Ω, i.e., they may modify the flows on the same switch concurrently without any inconsistencies as the flows affected in each case are completely unrelated. This ensures that no two *configurators* interfere with each other while performing concurrent topology reconfigurations on the same network. As our design assumes $k >> n$, each *configurator* may be responsible for multiple partitions. It is important to note that each *configurator* needs to maintain only those CP-configs that have been assigned to it. By employing such a mechanism, we avoid all kinds of coordination overhead among *configurators* while ensuring control plane consistency in a distributed setting.

Figure 6.5 depicts a middleware where each *configurator* has one or more partitions assigned to it and each partition is represented by its corresponding dz. Such a representation enables the direct mapping of advertisements/subscriptions (represented by

one or more *dz*s) to partitions. For example, two partitions {000}, {0010} have been assigned to $c_1 \in C$, implying that c_1 only maintains CP-configs for these two partitions, affecting flows related to these subspaces.

6.3.2.1 Topology Reconfiguration

The *dispatcher* plays a significant role in this approach. It maintains a map of the *configurators* and their associated partitions and performs partition-specific dispatch of control requests. Again, the *dispatcher* first prepares a control request for further processing by splitting it into partial requests, if necessary, depending on the partitions (cf. Section 6.3.1). This guarantees the mapping of a request to a single partition enabling the *dispatcher* to directly forward a request to a *configurator* responsible for the corresponding partition. For example, as per Figure 6.5, if a request corresponds to {00}, the *dispatcher* first splits it up into three requests {000}, {0010}, {0011} and then dispatches them to c_1 and c_2 as $\{00\} \succ \{000, 0010, 0011\}$. Consequently, all three CP-configs are reconfigured for this single request.

Each *configurator* maintains a request queue for each partition it is responsible for. Processing of control requests within a *configurator* takes place sequentially. This, in turn, ensures strong consistency within each partition as all conflicting actions are serially processed. Once a request is dispatched, it gets enqueued to the relevant queue and waits for the *configurator* to dequeue it for further processing. While choosing the next queue from which to dequeue a request, a *configurator* considers the order in which requests for all its assigned partitions arrived, ensuring request processing fairness. After dequeuing a request, it proceeds with reconfiguration of a specific CP-config corresponding to the request dz. Topology reconfiguration follows the usual mechanisms discussed in Chapter 2.

The shared nothing approach enables concurrent processing of requests (i.e., concurrent execution of actions) corresponding to disjoint partitions at multiple *configurators*, thus reaping the benefits of scaling. However, the true potential of this design can be realized if the workload can be balanced between *configurators*. There may be scenarios where the workload is much higher for certain partitions which burdens a few *configurators* while others remain idle. This degrades the responsiveness of the control plane to control requests. For this reason, adaptive load balancing among *configurators* bears considerable significance and features as the subject of discussion in the remaining part of this subsection.

6.3.2.2 Adaptive Load Balancing

In the face of a dynamic incoming workload, an adaptive policy is central to the load balancing approach. We identify load of a *configurator* at a given time by request queue

lengths of all partitions assigned to it. A request queue, specific to a partition (say pt_j), consists of all control requests waiting to be processed by the *configurator* for an assigned partition. So, load at a *configurator* c_i may be defined as,

$$l_i = \sum_{j=1}^{m} QL_j \quad (6.1)$$

where m is the number of partitions assigned to c_i and QL_j represents queue length at pt_j.

When an overload condition is detected at a heavily loaded *configurator*, one or more of its assigned partitions is migrated to a *configurator* with current minimum load. This implies that the task of processing all current and future requests for the migrated partitions now lies with the newly chosen *configurator*. An overload detection is carried out by the *monitor* component. The *monitor* periodically collects load information of every *configurator* and hence can easily identify an overload condition. With every periodic collection, the *monitor* calculates the average queue length at a *configurator*, denoted by l_{avg}. If the ratio of the load at a *configurator*, i.e., l_i, to l_{avg} is greater than a threshold value, then the monitor detects an overload and proceeds with partition migration. More formally, an overload is detected if,

$$\frac{l_i}{l_{avg}} > threshold \quad (6.2)$$

where $l_{avg} = \frac{\sum_{s=1}^{n} l_s}{n}$. However, in order to avoid partition thrashing, the monitor initiates migration only if the overload condition at a *configurator* is monotonically increasing with time. Initially, the most heavily loaded partition at the overloaded *configurator* is selected for migration and the effects of migrating it to the minimally loaded *configurator* is calculated. If this results in a potential overload condition at the minimally loaded *configurator*, the monitor proceeds to calculate the feasibility of migration of the next most heavily loaded partition until a balanced migration is achieved or all partitions considered for migration.

Migration of a partition, say pt_i, essentially means transfer of state from one *configurator* to another. This state includes the CP-config, cp_i, associated with pt_i and all pending requests related to it in the queue of the overloaded *configurator*. Also, while this transfer is underway, all new requests corresponding to pt_i that arrive at the *dispatcher* need to be stalled to avoid unnecessary state transfer. Once migration is completed, the *dispatcher* forwards the pending requests and all corresponding ones associated with pt_i to the newly assigned *configurator*.

It should be noted that SNA is suitable for both vertical as well as horizontal scaling. However, in the case of vertical scaling, it may perform worse than SEA in the presence of fluctuating unevenly distributed workload. Subject to such workload, adaptive load balancing of SNA will always be outperformed by the implicit optimal load balancing achieved in SEA. This is further confirmed by our evaluation results (cf. Section 6.6).

6.4 Keeping DP-config Consistent with CP-config

So far, we have considered that CP-config is a reflection of DP-config, i.e., when an action is executed on a switch, the resulting flow changes get reflected on both DP-config and CP-config before another action which is in conflict (cf. Section 6.2) with the previous action is executed by a *configurator*. In this section, we provide details about the protocol that ensures the same.

Please recall that an action consists of operations that include reading the current switch state, making decisions to modify this state, and deploying these flow modifications to the switch maintained at both the DP-config (physical network) and CP-config. In order to ensure consistency between CP-config and DP-config, an action must be considered to be completely executed only when both DP-config and CP-config have been updated as otherwise there may be inconsistencies between the two configurations. We demonstrate the same with an example where two conflicting actions are serially processed at the control plane. So, first, a *configurator* sends flow modification requests for this action to the physical switch (i.e., DP-config) and also updates the CP-config with the same changes. Let us assume that, at this point, the action is considered complete and the next action which is in conflict with the first one is similarly executed. While the flow modifications for the two conflicting actions are performed in order at the switch in CP-config (which resides in the control plane), they may have been executed in the reverse order (or not at all) on the switch in the physical network (i.e., DP-config). This will lead to inconsistencies between CP-config and DP-config, resulting in incorrectness in the system. As a result, it is crucial to ensure that the flow changes are also executed on the switch in the physical network before the execution of an action is considered to be complete.

So, our middleware pushes out the flow modification requests, generated while executing an action, to the switch and waits until the switch acknowledges the successful completion of these updates within a given timeout. The flow monitoring functionality introduced by OpenFlow version 1.4 can be efficiently used to allow a *configurator* to be notified by a switch about flow operations (addition/modification/deletion) performed on its tables. Such switch notifications can serve as acknowledgments of completed flow table updates. On receiving an acknowledgment from the switch, the *configurator* writes these changes to the CP-config and considers the action as fully executed. The following action can now be executed without any inconsistencies. This ensures that all changes are made to DP-config in the same order as CP-config and CP-config can be considered as a reflection of DP-config.

However, if an acknowledgment does not arrive at a *configurator* within the given timeout, the *configurator* marks all the unacknowledged flow changes as undefined in CP-config and the action is considered as incomplete. An acknowledgment may not have arrived at the *configurator* due to multiple reasons. For example, the reason could be the presence of a failure such as a lost connection between the *configurator*

and the switch. In such a scenario, the *configurator* cannot be certain that the flow modifications that were sent to the switch were actually executed by the switch. As a result, the status of the flow changes are kept as undefined till an acknowledgment to repeated requests is received from the switch. Please recall that the processing of a control request consists of executing an action on each switch between publishers and subscribers relevant to this control request. So, even if one of the actions is incomplete, the processing of the control request is considered incomplete. The processing of all subsequent control requests that have containment relations with the incomplete control request must be stalled till the incomplete control request is fully processed. This implies that, for both SEA and SNA, the processing of control requests in the partition containing the incomplete request is stalled till the processing of the incomplete request is completed. Meanwhile, concurrent processing of control requests belonging to other partitions continue.

Inconsistencies between CP-config and DP-config may arise due to other failures such as switch failures. In case of a switch failure, the spanning tree maintained by CP-config has to be modified accordingly, which means that all paths need to be recalculated according to the new topology. The same has to be done in case of a switch recovery as this also involves a change in the network topology and must be reflected in CP-config to ensure consistency. The control plane itself may fail as a result of which CP-config may be lost. In such a scenario, on recovery, a *configurator* must explicitly read the current status of the switches in the network using the OpenFlow standard in order to ensure consistency between CP-config and DP-config.

6.5 Reducing Flow Operations

Increasing responsiveness of the control plane to control requests also increases the rate at which network updates are pushed to the switches by multiple *configurators*. With today's hardware switches supporting around 40-50 flow-table updates per second [HYS13], it would be really beneficial if the total number of flow updates could be reduced. However, this would have to be achieved without degrading the performance of the system, i.e., without introducing false positives and false negatives.

We claim that the number of network updates can be reduced by exploiting the knowledge of advertisements and subscriptions and their relations yet again. Using the relations, processing of control requests can be ordered to optimize the network update procedure. We explain the optimization process at a switch level w.r.t. subscriptions and identify two relations that make a difference in the ordering of control requests. If two subscriptions sub_i and sub_j, where $sub_i \succ sub_j$, independently produce two new flows fl_i and fl_j, respectively, then the two relations between the flows that ordering would benefit from are complete containment, i.e., $fl_i \succ fl_j$, and partial containment, i.e., $fl_i \gtrapprox fl_j$.

Referring to the two subscriptions in the above example and their relations, we, first,

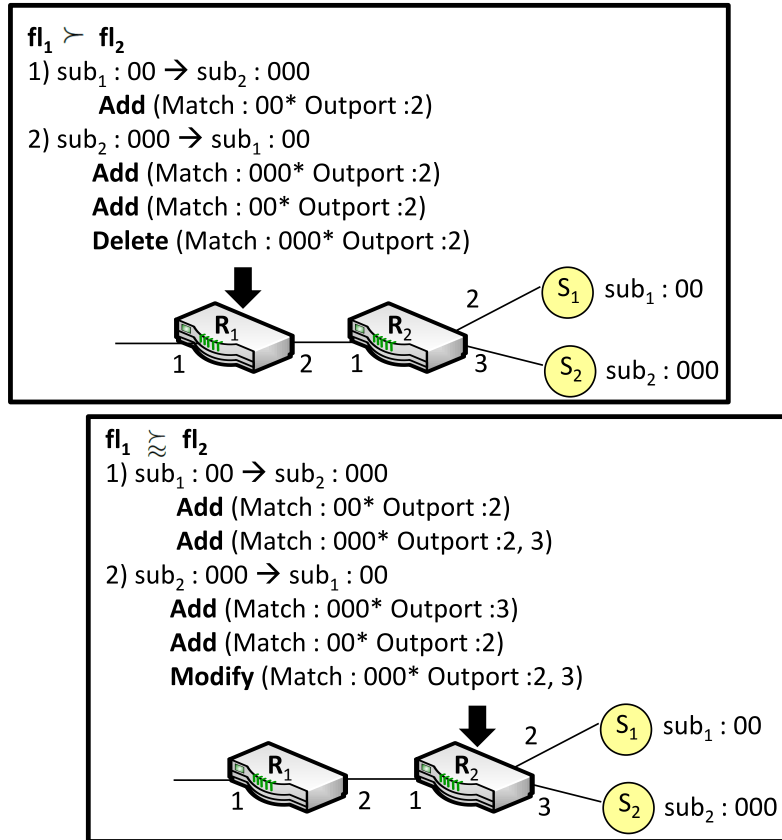


Figure 6.6: Reducing Flow Operations

look at complete containment between flows. The following updates would be done on a switch depending on the order in which the two subscriptions are processed. 1) If sub_i is processed before sub_j, sub_i first produces one add flow (fl_i) operation on the switch. When sub_j is processed, it does not produce any other flow updates on the switch as fl_i fully covers all events that need to be forwarded in response to sub_j. 2) If sub_j is processed before sub_i, sub_j also produces one add flow (fl_j) operation on the switch. After this, when sub_i is processed another flow (fl_i) add operation has to be performed to cover forwarding of all events matching sub_j and also those matching sub_i but not sub_j. Also, a delete operation has to be performed on fl_j as it is now redundant. Given the limitations of the flow table size on a switch, redundant flows cannot be afforded. This, clearly, indicates that the first ordering yields two operations less as compared to the second. Figure 6.6 illustrates the above discussion with an example where the ordering of two subscriptions sub_1 ($\{00\}$) and sub_2 ($\{000\}$) that would independently produce fl_1 and fl_2 yield different number of operations on switch R_1 as $fl_1 \succ fl_2$.

Let us now consider the second relation of partial containment between the flows. Again, we look at the number of operations required on ordering sub_i and sub_j differ-

ently. 1) If sub_i is processed before sub_j, sub_i produces one add flow (fl_i) operation. When sub_j is processed, a second flow (fl_j) add operation needs to be performed as this time the flows are only partially related and a different out port needs to be added only for sub_j. 2) However, if sub_j is processed before sub_i, sub_j produces one add flow (fl_j) operation on the switch. Now, when sub_i is processed, first a flow (fl_i) gets added for this subscription. Also, since the events relevant to fl_j are also relevant to fl_i (as $sub_i \succ sub_j$), a modify operation is performed on fl_j to accommodate the out port for sub_i. Again, the first ordering yields lesser operations as compared to the second. This is again illustrated in Figure 6.6, and this time the operations w.r.t. both orders are tracked on switch R_2 where a partial containment relation between fl_1 and fl_2 ($fl_1 \succapprox fl_2$) occurs.

It is important to note that the reordering of subscriptions does not have an impact on the correctness of the system. This is because, no matter how processing of requests is ordered, the final set of flows deployed on the switches is always the same. In Figure 6.6, at the end of processing sub_1 and sub_2, both switches have the same flows irrespective of the order in which they were processed. However, ordering may have an effect on the response time to certain requests that get scheduled later (cf. Section 6.6).

Similarly, efficient ordering of advertisements, unadvertisements, and unsubscriptions that have overlapping switches and are bound by the above relations reduce the number of network updates significantly. However, ordering of two control requests of different types should never be done. For example, the order of processing a subscription with an unsubscription must not be changed as this may result in undesirable system behavior. Both our designed approaches benefit from relevant ordering of control requests of the same type in the waiting queues of the *configurators*.

6.6 Performance Evaluations

This section is dedicated to an analysis of the design and implementation of our architecture and related approaches. A series of experiments are conducted to understand the effects of the design of the control plane on performance metrics such as (i) control plane throughput, (ii) average processing latency of control requests, and (iii) required number of flow operations on switches. We evaluate our approaches w.r.t. vertical and horizontal scaling of the control plane in order to understand the benefits of scaling up and scaling out.

6.6.1 Experimental Setup

We scale the control plane both vertically and horizontally on a testbed consisting of a small local area network which includes a cluster of physical machines capable of hosting each component of the proposed architecture. Vertical scaling is realized

by hosting multiple *configurators* on a single physical machine with 4 cores, 3.4 GHz processor and 8 GB of RAM. On the other hand, horizontal scaling is achieved by hosting multiple *configurators* on multiple physical machines where each machine in the cluster has 4 cores, 3.4 GHz processor, and 8 GB of RAM. Two separate machines host the *dispatcher* and the *monitor*.

The aforementioned setup deals with the control plane. In order to realize the data plane, our setup uses Mininet. Since we use a very large fat-tree topology with 64 hosts (publishers and/or subscribers) and 102 OpenFlow-enabled switches for all our experiments, Mininet is an ideal choice. It is also important to note that since our evaluations focus on control plane performance, they are independent of a real or emulated data plane.

We use a content-based schema that consists of up to 10 attributes for our event space, where the domain of each attribute varies in the range $[0, 1023]$. Again, experiments are performed using two different models of data distribution for generating control requests ((un)advertisement/(un)subscription). So, the uniform model generates control requests uniformly over Ω, whereas, the interest popularity model chooses 8 hotspot regions around which control requests are generated using the widely used zipfian distribution. The rate at which control requests are sent by the participants (i.e., publishers and subscribers connected to a software-defined network) to the dispatcher also follows two models of distribution, i.e., uniform and poisson. A uniform rate implies that the occurrences of incoming requests at the *dispatcher* are distributed uniformly on an interval of time. However, poisson rate involves a fluctuating workload while maintaining an average rate of incoming requests at the *dispatcher* within a given interval of time. So, there may be bursts of incoming requests from time to time along with lull periods to ensure an average rate at the *dispatcher*.

6.6.2 Vertical Scaling

In this section, we evaluate throughput and average processing latency of a vertically scaled control plane following the shared everything (SEA) and shared nothing with load balancing (SNA-LB) approaches. We partition the event space into 64 disjoint partitions on which each approach operates. Additionally, in SNA-LB, we randomly assign partitions to the *configurators* on system start-up. Also, 64 subscribers issue up to 200,000 subscriptions and unsubscriptions at various uniform and poisson rates to generate load at the control plane.

The first set of experiments measures the maximum rate at which the control plane can process control requests, i.e., throughput, with increasing number of *configurators*. It is important to note that control requests may be further broken down into partial requests to contain them in different partitions. A control request is considered to be processed only when all its partial requests have been processed. Figure 6.7(a) and 6.7(b) show that, with increasing number of *configurators*, the throughput of the control

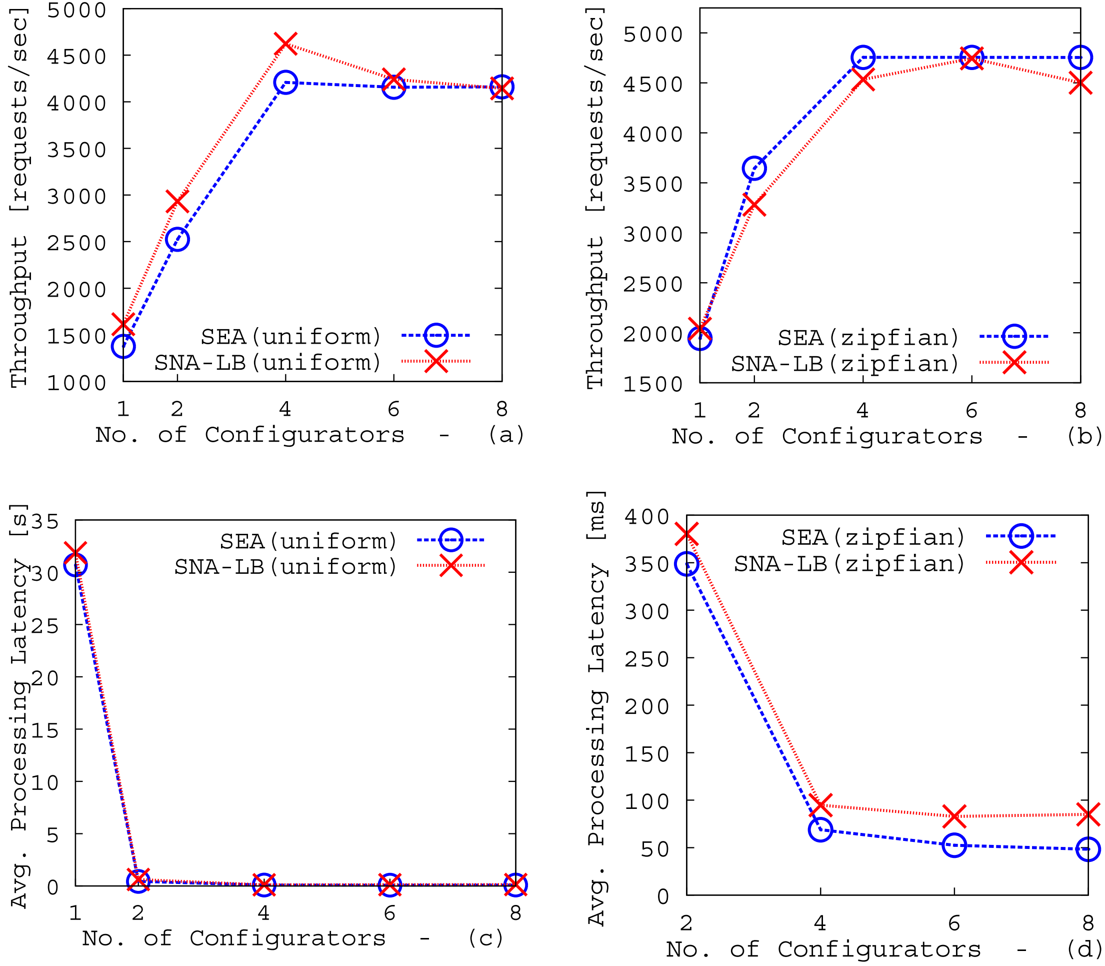


Figure 6.7: Performance Evaluations : Vertical Scaling

plane increases significantly for both uniform and zipfian data till the control plane is scaled up by 4 *configurators* for both approaches. Not surprisingly, as the *configurators* are hosted by a machine with a 4-core architecture, there is not much benefit if the control plane is scaled beyond 4. Figure 6.7(b) shows that, for control requests following zipfian distribution, the throughput of SEA is higher as compared to SNA-LB. This is because, for zipfian data, the workload is not evenly distributed among the partitions. This means that in SNA-LB, some *configurators* may be more heavily loaded while others remain relatively idle. Even though SNA-LB tries to balance this load, it does so only after a threshold limit is crossed, while SEA ensures that no *configurator* is ever idle unless there are no more requests to process. SEA implies optimal load balancing among *configurators*. In case of uniform data, where all partitions are equally loaded, Figure 6.7(a) shows that the performance of SEA is slightly worse than the other as once the benefits of load balancing are not visible, the additional synchronization

overhead required in SEA renders it less effective as compared to SNA-LB.

In the context of our work, responsiveness is directly related to the overall time it takes for a control request to be processed by the control plane (i.e., processing latency). We define processing latency as the time elapsed from the issuance of the request by a publisher/subscriber to the time when all partial requests for this request have been processed by the control plane. In this experiment, we plot the average processing latency of control requests with increasing number of *configurators* in a vertically scaled control plane. We show a comparison of both the approaches when subscription and unsubscription requests are generated using both uniform and zipfian distributions and are sent by the subscribers to the *dispatcher* at a poisson rate of 2500 requests/sec. Figure 6.7(c) and Figure 6.7(d) show that, for both uniform and zipfian data and for both approaches, the average processing latency reduces significantly with increasing number of *configurators* till 4 *configurators*. Again, scaling beyond 4 *configurators* may not have any benefits due to the reason mentioned above. Figure 6.7(c) suggests that there is not much difference in performance between the approaches for uniform data as all partitions get similar amount of workload. This implies that all *configurators* get similar amount of workload in SEA and SNA-LB. However, the difference in benefits between the approaches is visible for zipfian data and as a result we focus on comparing their performances by zooming the graph in Figure 6.7(d). In general, with dynamically changing incoming workload, SEA performs better as compared to SNA-LB as it ensures optimal load-balancing. As mentioned before, with uneven load corresponding to different partitions, and a poisson rate of incoming request, the queues formed at different *configurators* are of different lengths for SNA-LB. This implies much longer waiting times for some requests waiting at the end of long queues, resulting in a higher average processing latency.

6.6.3 Horizontal Scaling

We also evaluate throughput, average processing latency, and required number of flow operations in a horizontally scaled control plane. We especially compare the performances of shared nothing without load balancing (SNA) and shared nothing with load balancing (SNA-LB) approaches in order to show the effects of load balancing on this approach. As SEA does not scale well in a physically distributed setting, our evaluations in this section do not include this approach. As in the experiments for vertical scaling, we partition the event space into 64 disjoint partitions unless otherwise specified. Also, 64 subscribers issue up to 200,000 subscriptions and unsubscriptions at various uniform and poisson rates to generate load at the control plane.

Figure 6.8(a) and Figure 6.8(b) show the throughput of a horizontally scaled control plane for uniform and zipfian data, respectively. In both SNA and SNA-LB, the throughput increases with increasing number of *configurators* for both distributions as a horizontally scaled setup does not suffer from the limitations of a vertically scaled one

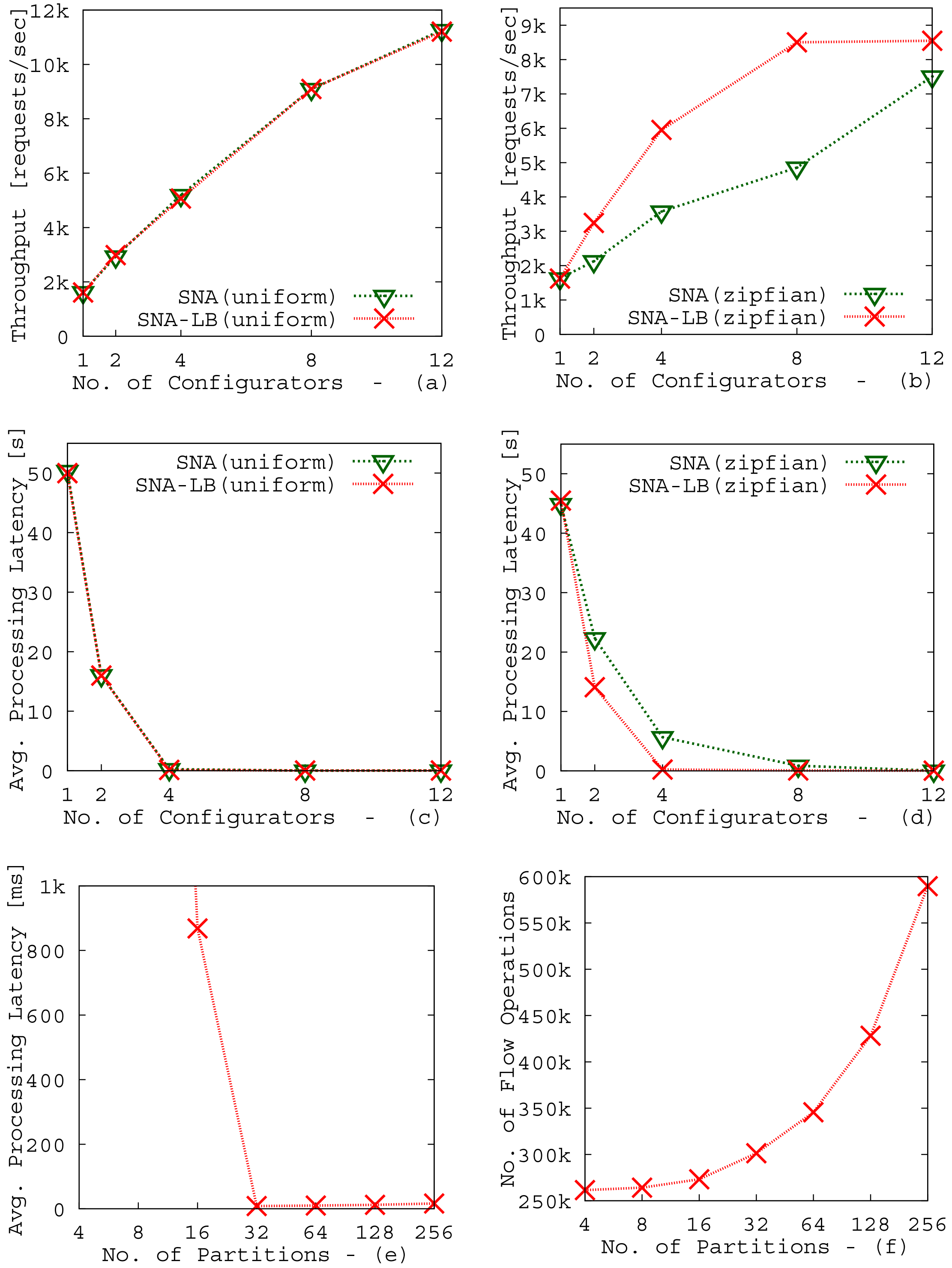


Figure 6.8: Performance Evaluations : Horizontal Scaling

in terms of number of cores. Scaling out provides a lot of flexibility and can be used effectively to increase control plane throughput as shown in the graphs. Not surprisingly, there is not much difference between the plots of SNA and SNA-LB for uniform data. However, the benefits of load balancing are again visible for zipfian data where SNA-LB outperforms SNA.

We also conducted experiments which measure average processing latency of control requests with increasing number of *configurators* when subscriptions and unsubscriptions are generated using both uniform and zipfian data and sent to the *dispatcher* at a poisson rate of 5000 requests/sec. Figure 6.8(c) and Figure 6.8(d) show behavior similar to that obtained in vertical scaling where the average processing latency reduces significantly with scaling. The plots for uniform distribution are similar for both SNA and SNA-LB, whereas SNA-LB performs better, when zipfian data is used, due to additional load balancing. This means that SNA-LB provides a possibility to migrate partitions to manage the maximum length of the waiting queues, whereas SNA has no such possibility, because of which the average processing latency for SNA-LB is mostly lower than that of SNA.

It is also interesting to observe the average processing latency of a control request with increased partitioning of the event space when SNA-LB is used. More the number of partitions, more is the possibility of load balancing in SNA-LB, when dealing with requests following zipfian distribution. If a *configurator* has a large partition with very high load, moving it to any other *configurator* will not balance the load. However, if the partitions are smaller, the possibility of the load being distributed among these partitions is more, which increases the flexibility of balancing the load between multiple *configurators*. Figure 6.8(e) shows that, for zipfian data, the average processing latency reduces significantly with increasing number of partitions up to a point. However, beyond this point further partitioning has no benefits as no further load balancing is possible for the considered workload. In fact, the graph indicates that once these benefits are no longer applicable, further partitioning may increase the average latency to some extent. This is because increased partitioning has an effect on the number of partial requests that are constructed from control requests. If the partitioning is more fine granular, the probability of a control request spanning multiple partitions is higher. This means that multiple CP-configs will be affected resulting in increased number of flow operations. Figure 6.8(f) plots the effects of partitioning on total number of flow operations. The graph clearly shows that partitioning increases the number of flow operations significantly which can have an impact on the flow updates on the network.

6.6.4 Reducing Flow Operations

In order to reduce the number of flow operations on switches, we order control requests as discussed in the previous section. However, continuous sorting of a waiting queue at a *configurator* not only poses a significant overhead but also results in starvation for

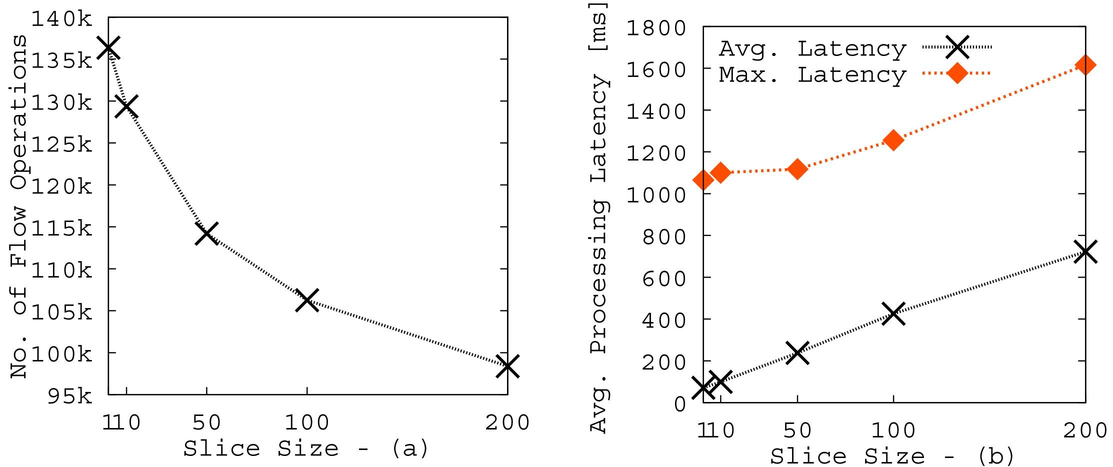


Figure 6.9: Performance Evaluations : Reducing Flow Operations

some fine-grained subscription requests that get continuously pushed down in the sorted queue. As a result, we sort only slices of contiguous subscriptions at a time and not the complete waiting queue. This set of experiments plots the number of flow operations required to process a set of 5000 subscriptions with increasing slice size. Figure 6.9(a) clearly shows that with increasing slice size, the number of flow operations reduces. However, Figure 6.9(b) shows that due to starvation of certain requests, the average latency is affected on increasing the slice size. We also plot the maximum processing latency for each slice size that contributes to increasing the average processing latency. So, there is always a trade-off between the slice size and fairness in request processing that directly affects the responsiveness to certain requests. It is important to note that a slice size of 1 implies an unsorted queue.

6.6.5 Discussion

Our evaluation results show the importance and the advantages of both vertical and horizontal scaling in the control plane. Both SEA and SNA-LB increase the throughput of the control plane manifold while reducing the average processing latency by up to 99% as compared to a centralized controller. In the context of vertical scaling, both SEA and SNA-LB perform very well with SEA having an additional edge due to its implicit load balancing property. In the context of horizontal scaling, where SEA would incur too much overhead, the importance of SNA and SNA-LB is evident. In fact, evaluation results show that adaptive load balancing can further improve the performance of SNA in most cases. Moreover, the evaluations show the impact of partition size on the performance of SNA-LB w.r.t. average processing latency and number of required flow operations. The results clearly show the trade-off between the two with increasing number of partitions. Finally, our evaluations, also, show that

reordering the handling of control requests can reduce the required flow operations in the network by up to 28%.

6.7 Related Work

Efficient maintenance and handling of dynamically changing subscriber interests has also been a subject of much research in overlay-based pub/sub middleware [CS04, JJE10, CFMP04]. For instance, Jayaram et al. [JJE10] propose mechanisms to efficiently handle subscriptions that change dynamically w.r.t. various parameters (such as location) by introducing the concept of parametric subscription. These methods, however, cannot be directly applied to the problem addressed in this chapter.

As discussed in Chapter 4, the importance of a scalable and elastic pub/sub with high throughput has been impressed upon in literature. But the question is, can these mechanisms be employed in the context of scaling the control plane in software-defined networking? Please recall, Li et al., present an attribute-based pub/sub service, BlueDove [LYK+11], that organizes multiple servers into an overlay and achieves high throughput filtering (or matching) of events by forwarding events to be matched to the least loaded servers. Likewise, Barazzutti et al. design a scalable pub/sub service, StreamHub [BFF+13] and e-StreamHub [BHM+14], where a set of independent operators take advantage of multiple cores on multiple servers to perform pub/sub operations which include subscription partitioning and event filtering. In fact, e-StreamHub supports both scaling in and scaling out depending on the load observations of the system. However, it is important to note that all these systems target parallelism of event filtering and do not need to take care of concurrency control as the servers enabling concurrent filtering of events do not share any resources.

Scaling the control plane in SDN, however, involves concurrent access to the network, acting as a shared resource, and has been the subject of much research in recent times [YG16,TG10,BRKB13,KCG+10]. Levin et al. [LWH+12] explore the trade-offs of state distribution in a distributed control plane and motivate the importance of strong consistency in their work. They investigate the impact of eventual consistency on the performance of a load-balancer implemented using SDN and infer that the lack of strong consistency severely degrades application performance. To ensure strong consistency of network state between multiple controller instances, Onix [KCG+10] provides a transactional persistent database backed by a replicated state machine. However, it claims that, for applications requiring frequent network updates, dissemination of state updates using this technique yields severe performance limitations. As a result, to accommodate such applications, Onix also proposes a mechanism for obtaining eventual consistency using a memory-only DHT which has its limitations w.r.t. consistency guarantees. Similarly, Hyperflow [TG10] only provides guarantees of maintaining weak consistency by passively synchronizing the global network views of all controllers. On

the other hand, Botelho et al. [BRKB13] show that by using a classical state machine replication technique the cost of coordination to guarantee strong consistency may become bearable for certain SDN applications, but not in general. This chapter, in contrast to the aforementioned literature, not only focuses on line-rate forwarding of events in the data plane but also on achieving high responsiveness while ensuring strong consistency on the control plane.

6.8 Conclusion

In this chapter, we have proposed an application-aware control for software-defined networks that is capable of enhancing the responsiveness of the control plane by allowing concurrent network updates while ensuring consistent changes to the data plane with low synchronization overhead even in the presence of network failures. In particular, we have designed two complementary approaches in the context of event-based middleware that take into account interests of publishers and subscribers in order to reap the benefits of horizontal and vertical scaling of the control plane. Moreover, we have proposed reordered (yet consistent) handling of control requests at the control plane to mitigate the limitations of current SDN switches w.r.t. number of supported flow updates per second. Our evaluations show that the application-aware control distribution drastically decreases the response time to control requests (up to 99% in comparison to a centralized controller) for both vertical and horizontal scaling while ensuring control plane consistency. Furthermore, reordered handling of control requests results in up to 28% less flow updates on the SDN switches.

Chapter 7

Summary and Future Work

This chapter summarizes the main contributions of this thesis and also provides a brief outlook on possible future work.

7.1 Summary

The growing amount of large-scale dissemination of information between participants of modern applications has made paradigms such as content-based publish/subscribe highly significant in today's world. For example, Google uses Cloud Pub/Sub to 'connect anything to everything' in an IoT environment [Goo]. Also, Microsoft uses Azure Event Hubs, a highly scalable pub/sub service to connect devices and applications across platforms [Eve]. Similarly, communication in Amazon Web Services IoT Platform heavily relies on the MQTT pub/sub protocol [AWS]. In fact, for the past few decades, content-based routing has been the popular choice for dissemination of content from publishers to subscribers in a loosely-coupled manner. Most traditional content-based pub/sub middleware implementations rely on an overlay network of software brokers that perform content-based filtering and routing to ensure loose-coupling between publishers and subscribers. However, due to filtering and routing being performed in the application layer, i.e., in software, their performance cannot match up to that of communication protocols implemented on the network layer w.r.t. throughput rates, end-to-end latency, and bandwidth efficiency. As a result, the following are the main contributions of this thesis.

1. We propose methods to implement the basic functionalities of the content-based publish/subscribe paradigm on the network layer by utilizing the capabilities of software-defined networking. In fact, our designed system, PLEROMA, pushes filtering and routing of events to the network layer, thus enabling line-rate forwarding of events. To this end, we provide mechanisms to represent content

filters that are capable of being installed in the TCAM of hardware switches. Our designed algorithms use these content filters to establish paths between publishers and subscribers for dissemination of published events to interested subscribers. Moreover, we provide methods to efficiently reconfigure the network in the presence of dynamic subscriptions and advertisements. We perform a series of experiments to evaluate the behavior of the designed system. Our evaluation results show that the filtering performance of the network layer (w.r.t. end-to-end latency) is significantly better than that of the application layer.

2. While PLEROMA manages to achieve line-rate performance of events, expressiveness of content filters installed in TCAM suffers due to inherent hardware limitations w.r.t. number of bits available on hardware switches to represent these filters. As a result, we explore various techniques to represent content filters expressively despite being limited by hardware in order to reduce unnecessary traffic in the network. Our designed techniques i) use workload, in terms of events and subscriptions, to represent content, and ii) efficiently select attributes to reduce redundancy in content. Moreover, these techniques complement each other and can be combined together to further reduce false positives in the system. Our detailed performance evaluations show the potential of these techniques in reducing unnecessary traffic (up to 97%) when subjected to different workloads.

3. In order to further increase expressiveness of content filters, we propose methods to strike a balance between purely application-layer-based and purely network-layer-based implementations of content-based pub/sub. In fact, we realize a hybrid content-based middleware that enables filtering of events in both software (i.e., application layer) and hardware (i.e., network layer). Moreover, we provide algorithms with various associated complexities and benefits to determine the layer in which each event gets filtered such that unnecessary network traffic can be minimized while also considering latency requirements of the middleware. In fact, depending on the performance requirements of the system, our hybrid middleware provides a full range of configurations between a pure application-layer-based implementation and a pure network-layer-based implementation. We provide a detailed performance evaluation of the proposed selection algorithms to determine their impact on the performance of the designed hybrid middleware which we further compare to a state-of-the art solution.

4. We, also, address another hardware limitation, i.e., limited number of flow table entries available to pub/sub traffic in TCAM of hardware switches. We design a filter aggregation algorithm that merges filters on individual switches to respect TCAM constraints while ensuring minimal increase in unnecessary network traffic due to the merges (i.e., reduced expressiveness of aggregated filters). Our algorithm uses the knowledge of advertisements, subscriptions, and a global view of the network state for taking aggregation decisions that would have minimum adverse impact on the bandwidth efficiency of the system. We provide different

flavors of this algorithm and thoroughly evaluate and compare their performances under realistic workload. Our evaluation results show that our designed aggregation algorithm successfully meets TCAM constraints on switches while also reducing unnecessary traffic introduced in the network due to aggregation by a baseline approach by up to 99.9%.

5. An important requirement of a pub/sub middleware is high responsiveness to dynamically changing advertisement and subscription requests (i.e., control requests). So, we present a distributed control plane that is capable of both vertical and horizontal scaling. Our scaling methods not only improve responsiveness by enabling concurrent network updates in the presence of high dynamics but also ensure consistent changes to the data plane. To this end, our proposed methods use knowledge of the application semantics that is available in the design of the data plane of the pub/sub middleware, e.g., subscriptions and advertisements, to perform concurrent and consistent network updates. Our detailed evaluations show that the designed scaling methods drastically decrease response time to control requests by up to 99% in comparison to a centralized controller while ensuring control plane consistency. Moreover, we also propose a method to consistently reorder processing of control requests at the control plane which would reduce the number of necessary flow updates in the network in order to mitigate the limitation on hardware switches w.r.t. number of supported flow updates per second. The designed method ensures that reordering the handling of control requests does not impact the correctness of the system, i.e., it does not introduce any additional false positives or false negatives in the system. Our evaluation results show that the proposed reordered handling of control requests reduces flow updates on the SDN switches by up to 28%.

In conclusion, this thesis realizes a content-based pub/sub middleware on software-defined networks. It addresses various limitations on both the data plane as well as the control plane in order to provide a high performance solution.

7.2 Future Work

There are several possible directions in which the work presented in this thesis can be extended as part of future research. The following includes a brief outlook on the most promising research directions in the context of content-based routing on software-defined networks.

- In Chapter 2 we provide methods to map content to binary strings that are capable of being installed in TCAM using spatial indexing. However, it would be interesting to explore other techniques such as bloom filters, hashes, one-dimensional representation using space-filling curves, etc., to transform content to binary strings. The practicality of these techniques and how they compare to

spatial indexing, that generates dz strings in PLEROMA, can prove to be very interesting. In fact, with the recent advent of various programming abstractions in SDN [BBBS16, BDG+14], other techniques that look into the manner in which content filters are mapped to match fields of flows in TCAM can also be explored. For example, the flexibility offered by P4 [BDG+14] to implement new protocols and headers may be harnessed to process event packets independent of any hardware target.

- While PLEROMA offers line-rate forwarding of events, it does not provide hard latency guarantees. A significant number of modern applications, however, are highly time-sensitive and demand stringent real-time guarantees such as bounded latency and jitter from the underlying network. While there has been significant research in the field of time-sensitive software-defined networks for real-time applications [NDR16, DN16], time-sensitive content-based routing on software-defined networks is yet to be explored in literature. The aforementioned work in time-sensitive software-defined network has been designed primarily for cyber-physical systems where Integer Linear Program formulations are used by the logically centralized control plane to compute transmission schedules for time-sensitive traffic. The question is, can these methods be adopted in content-based pub/sub on software-defined networks? Are additional mechanisms required to enable loose-coupling between publishers and subscribers while they remain oblivious to the underlying middleware? Can knowledge of application semantics aid the process? Considering the demanding requirements of modern applications that use content-based routing, venturing into time-sensitivity in pub/sub systems can prove to be a very promising area of research.
- We have already identified and discussed the inherent scalability limitations of a single controller instance with respect to reconfiguration efforts in the face of high dynamics in this thesis. In fact, in Chapter 6, we provide methods to scale the control plane where every control element has a view of the entire network. But what if multiple controllers were responsible for separate administrative domains where each controller performs reconfiguration in its designated domain? PLEROMA could further offer interoperability between multiple network domains. Independently managed network domains naturally arise in many business scenarios, for instance to avoid interference of manufacturing processes and enforce security policies in accessing events [PEB07, SKPR10]. In addition, partitioning of a large network can further increase the scalability by performing multiple concurrent network updates without worrying about consistency as the network is, in this case, no longer a shared resource. While this has been briefly discussed in [TKBR14, Bho13], domain-specific control should be further explored to understand the behavior of such designs w.r.t. responsiveness to control requests within a domain, responsiveness in the presence of inter-domain communication, etc. In fact, determining the participants for each administrat-

ive domain and efficiently partitioning the underlying network to be configured by the corresponding domain-specific controller can be scope for much research. Moreover, combining scaling techniques, presented in this thesis, within a domain of a multi-domain architecture could result in some interesting findings.

Bibliography

[100] SDN system performance. `http://www.pica8.com/pica8-deep-dive/sdn-system-performance/`.

[512] 512K-Day, Firstpost Article. `https://goo.gl/gC77Hk`.

[AT06] Ioannis Aekaterinidis and Peter Triantafillou. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *In Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), Lisboa, Portugal*, 2006.

[AWS] Amazon Web Services. `https://aws.amazon.com/iot/how-it-works/`.

[Azu15] Report from Open Networking Summit: Achieving Hyper-Scale with Software Defined Networking, 2015.

[Bal17] Alexander Balogh. Addressing TCAM limitations in an SDN-based pub/sub system. Master thesis, University of Stuttgart, Germany, 2017.

[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, 2016.

[BBQ+07] Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, Antonino Virgillito, and Roma Italia. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *The Computer Journal*, 2007.

[BCM+99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, 1999.

[BDFG07] Silvia Bianchi, Ajoy Kumar Datta, Pascal Felber, and Maria Gradinariu. Stabilizing peer-to-peer spatial filters. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, 2007.

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Var-

ghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 2014.

[Bet00] Katherine Betz. A scalable stock web service. In *Proceedings of the 2000 International Workshop on Parallel Processing, ICPPW 2000, Toronto, Canada, August 21-24, 2000*, 2000.

[BFF+13] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, 2013.

[BFG07] Silvia Bianchi, Pascal Felber, and Maria Gradinariu. Content-based publish/subscribe using distributed R-trees. In *Proceedings of 13th International Euro-Par Conference*, 2007.

[BFP10] Silvia Bianchi, Pascal Felber, and Maria Gradinariu Potop-Butucaru. Stabilizing distributed R-trees for peer-to-peer content routing. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1175–1187, 2010.

[BG81] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), June 1981.

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[BHM+14] Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Proceedings of 34th IEEE International Conference on Distributed Computing Systems*, 2014.

[Bho13] Sukanya Bhowmik. Distributed control algorithms for adapting publish/subscribe in software-defined networks. Master's thesis, University of Stuttgart, November 2013.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[BMVV05] Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, 2005.

[BRKB13] Fábio Andrade Botelho, Fernando Manuel Valente Ramos, Diego Kreutz, and Alysson Neves Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *Second European Workshop on Software Defined Networks, EWSDN*, 2013.

[BTBR17] Sukanya Bhowmik, Muhammad Adnan Tariq, Alexander Balogh, and Kurt Rothermel. Addressing TCAM limitations of software-defined networks for content-based routing. In *Proceedings of the 11th ACM International Conference on Distributed Event-based Systems*, DEBS 2017, 2017.

[BTGR16] Sukanya Bhowmik, Muhammad Adnan Tariq, Jonas Grunert, and Kurt Rothermel. Bandwidth-efficient content-based routing on software-defined networks. In *Proceedings of the 10th ACM International Conference on Distributed Event-based Systems*, DEBS 2016, 2016.

[BTHR16] Sukanya Bhowmik, Muhammad Adnan Tariq, Lobna Hegazy, and Kurt Rothermel. Hybrid content-based routing using network and application layer filtering. In *Proceedings of 36th IEEE International Conference on Distributed Computing Systems*, ICDCS '16, 2016.

[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, 2015.

[BTK+17] Sukanya Bhowmik, Muhammad Adnan Tariq, Boris Koldehofe, Frank Dürr, Thomas Kohler, and Kurt Rothermel. High performance publish/subscribe middleware in software-defined networks. *IEEE/ACM Transactions on Networking*, 25(3):1501–1516, 2017.

[CDNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions Software Engineering*, 27(9):827–850, 2001.

[CFMP04] Gianpaolo Cugola, Davide Frey, Amy L. Murphy, and Gian Pietro Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, 2004.

[CJ11] Alex King Yeung Cheung and Hans-Arno Jacobsen. Green resource allocation algorithms for publish/subscribe systems. In *In Proceedings of the 31st International Conference on Distributed Computing Systems*, 2011.

[CLS03] Mao Chen, Andrea LaPaugh, and Jaswinder Pal Singh. Content distribution for publish/subscribe services. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, 2003.

[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, 2011.

[CMTV07] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In *Proceedings of the International Conference on Distributed Event-based Systems*, 2007.

[Com12] ONF Market Education Committee. *Software-defined Networking: The New Norm for Networks*. Open Networking Foundation, 2012.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, 2000.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[CS04] Fengyun Cao and Jaswinder Pal Singh. Efficient event routing in content-based publish-subscribe service networks. In *Proceedings of 23rd IEEE INFOCOM Conference*, 2004.

[DHM+13] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an elastic distributed SDN controller. In *Proceedings of 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[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, 2014.

[DN16] Frank Dürr and Naresh Ganesh Nayak. No-wait packet scheduling for IEEE time-sensitive networks (TSN). In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France*, 2016.

[DXG+11] C Dong, Q Xiuquan, J Gelernter, L Xiaofeng, and M Luoming. Mining data correlation from multi-faceted sensor data in the internet of things. In *China Communications*, 2011.

[ECG09] Christian Esposito, Domenico Cotroneo, and Aniruddha Gokhale. Reliable publish/subscribe middleware for time-sensitive Internet-scale applications. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2009.

[Edg] Hardware Switch Edge-Core AS5712-54X. `http://www.edge-core.com/`.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[ES14] Mohamed El-Shamouty. Efficient content-based routing using OpenFlow. Bachelor thesis, University of Stuttgart, Germany, 2014.

[Eve] Microsoft Event Hubs. `https://azure.microsoft.com/en-us/services/event-hubs/`.

[Goo] Google Cloud Pub/Sub. `https://cloud.google.com/pubsub/`.

[Gru14] Jonas Grunert. Increasing bandwidth efficiency of content-based routing in software-defined networks. Bachelor thesis, University of Stuttgart, Germany, 2014.

[GSAA04] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, 2004.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, 1984.

[GXC+15] Zehua Guo, Yang Xu, Marco Cello, Junjie Zhang, Zicheng Wang, Mingjian Liu, and H. Jonathan Chao. Jumpflow. *Computer Networks*, 92(P2):300–315, December 2015.

[Heg16] Lobna Hegazy. Evaluation and analysis of realizing broker-based content routing protocols in SDN. Master thesis, University of Stuttgart, Germany, 2016.

[Hof09] Todd Hoff. Latency is everywhere and it costs you sales -how to crush it. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`, 2009.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), December 1983.

[HYG12] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*, 2012.

[HYS13] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proceedings of 2nd ACM SIGCOMM Workshop on Hot Topics in SDN*, 2013.

[JCL+10] Hans-Arno Jacobsen, Alex King Yeung Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The PADRES publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*. 2010.

[JF08] Zbigniew Jerzak and Christof Fetzer. Bloom filter based routing for content-based publish/subscribe. In *Proceedings of the 2nd International Conference on Distributed Event-based Systems*, 2008.

[JJE10] K. R. Jayaram, Chamikara Jayalath, and Patrick Eugster. Parametric subscriptions for content-based publish/subscribe networks. In *Proceedings of 11th International Conference on Middleware*, 2010.

[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*, 2013.

[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*, 2014,.

[JMVM09] Hojjat Jafarpour, Sharad Mehrotra, Nalini Venkatasubramanian, and Mirko Montanari. MICS: An Efficient Content Space Representation Model for Publish/Subscribe Systems. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, 2009.

[Jol86] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[JPMH07] Michael A. Jaeger, Helge Parzyjegla, Gero Mühl, and Klaus Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, 2007.

[JZER+09] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. LIPSIN: line speed publish/subscribe internetworking. *ACM SIGCOMM Computer Communication Review*, 2009.

[KARW16] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, 2016.

[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 USENIX Conference on OS Design & Implementation*, 2010.

[KDR15] Thomas Kohler, Frank Dürr, and Kurt Rothermel. Update consistency in software-defined networking based multicast networks. In *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2015, San Francisco, CA, USA*, 2015.

[KDT13] Boris Koldehofe, Frank Dürr, and Muhammad Adnan Tariq. Tutorial: Event-based systems meet software-defined networking. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, 2013.

[KDTR12] Boris Koldehofe, Frank Dürr, Muhammad Adnan Tariq, and Kurt Rothermel. The power of software-defined networking: line-rate content-based routing using Openflow. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '12. ACM, 2012.

[KHK13] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *Proceedings of the IEEE INFOCOM Conference, Turin, Italy*, 2013.

[KLRW13] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, 2013.

[KORR12] Boris Koldehofe, Beate Ottenwälder, Kurt Rothermel, and Umakishore Ramachandran. Moving Range Queries in Distributed Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS).*, Berlin, 2012.

[KPP04] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer, Berlin, Germany, 2004.

[KTKR10] Gerald G. Koch, Muhammad Adnan Tariq, Boris Koldehofe, and Kurt Rothermel. Event processing for large-scale distributed games. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, 2010.

[LCZT07] Yijuan Lu, Ira Cohen, Xiang Sean Zhou, and Qi Tian. Feature selection using principal feature analysis. In *Proceedings of the 15th ACM International Conference on Multimedia*, MM '07, 2007.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A network on a laptop: Rapid prototyping for software-defined networks. In *Proceedings of 9th ACM Workshop on Hot Topics in Networks*, 2010.

[LMT10] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM razor: a systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Transactions on Networking*, 2010.

[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 Hot Topics in Software Defined Networks*, 2012.

[LYK+11] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2011.

[Mar07] Richard Martin. Wall street's quest to process data at the speed of light. *Information Week*, 2007.

[MC02] René Meier and Vinny Cahill. STEAM: event-based middleware for wireless ad hoc network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems, Workshops (ICDCSW '02), Vienna, Austria*, 2002.

[MFB02] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter similarities in content-based publish/subscribe systems. In *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, ARCS '02, 2002.

[MG04] Arnaz Malhi and Robert X. Gao. PCA-based feature selection scheme for machine defect classification. *IEEE T. Instrumentation and Measurement*, 2004.

[Mis13] Gagan Bihari Mishra. Providing in-network content-based routing using OpenFlow. Master thesis, University of Stuttgart, Germany, 2013.

[MJ14] Vinod Muthusamy and Hans-Arno Jacobsen. Infrastructure-free content-based publish/subscribe. *IEEE/ACM Transanctions on Networking*, 2014.

[MLJ10] Vinod Muthusamy, Haifeng Liu, and Hans-Arno Jacobsen. Predictive publish/subscribe matching. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, 2010.

[MLT12] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 20(2), April 2012.

[MPP15] Gero Mühl, Helge Parzyjegla, and Matthias Prellwitz. Analyzing content-based publish/subscribe systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway*, 2015.

[MSRS09] Anirban Majumder, Nisheeth Shrivastava, Rajeev Rastogi, and Anand Srinivasan. Scalable content-based routing in pub/sub systems. In *Proceedings of the 28th IEEE International Conference on Computer Communications, joint conference of the IEEE Computer and Communications societies (INFOCOM)*, 2009.

[Müh02] Gero Mühl. *Large-Scale Content-Based Publish-Subscribe Systems.* PhD thesis, TU Darmstadt, November 2002.

[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, 2013.

[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 2016, Brest, France*, 2016.

[NSBT14] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Optimizing rules placement in openflow networks: Trading routing for better efficiency. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, 2014.

[Ope] Open vSwitch. `http://openvswitch.org/`.

[Ope13] Open Networking Foundation. OpenFlow management and configuration protocol (OF-CONFIG v1.1.1). Technical report, March 2013.

[PB02] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Procs. of the 22nd International Conference on Distributed Computing Systems*, 2002.

[PC05] Olga Papaemmanouil and Ugur Centintemel. Semcast: Semantic multicast for content-based data dissemination. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2005.

[PEB07] Lauri I. W. Pesonen, David M. Eyers, and Jean Bacon. Encryption-enforced access control in dynamic multi-domain publish/subscribe networks. In *Proceedings of the inaugural international conference on distributed event-based systems (DEBS)*, 2007.

[Pic] PicOS Version 2.6. `http://www.pica8.com/documents/pica8-datasheet-picos.pdf`.

[PRGK09] Jay A. Patel, Étienne Rivière, Indranil Gupta, and Anne-Marie Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, 53(13):2304 – 2320, 2009.

[PW02] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '02, 2002.

[RFR+12] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012.

[RLW+02a] Anton Riabov, Zhen Liu, Joel L. Wolf, Philip S. Yu, and Li Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[RLW+02b] Anton Riabov, Zhen Liu, Joel L. Wolf, Philip S. Yu, and Li Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS, 2002.

[sen] Intel Research Berkeley Lab Sensor Data Set. `http://www.cs.cmu.edu/~guestrin/Research/Data/`.

[Ske81] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, 1981.

[SKPR10] Björn Schilling, Boris Koldehofe, Udo Pletat, and Kurt Rothermel. Distributed heterogeneous event processing: Enhancing scalability and interoperability of cep in an industrial context. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, New York, NY, USA, 2010.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, 2001.

[Sri17] Deepak Srinivasan. Ensuring data plane consistency in SDN-based publish/subscribe systems. Master thesis, University of Stuttgart, Germany, 2017.

[sto] Correlation in Stock Exchange Data. `http://www.investopedia.com/articles/technical/02/010702.asp`.

[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, 2003.

[TA90] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Operating Systems Review*, 24(3), 1990.

[Tar13] Muhammad Adnan Tariq. *Non-functional requirements in publish, subscribe systems*. PhD thesis, University of Stuttgart, 2013.

[TBF+03] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2Nd International Workshop on Distributed Event-based Systems*, DEBS '03, 2003.

[TG10] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for OpenFlow. In *Proceedings of Internet Network Management Conference on Research on Enterprise Networking*, 2010.

[TKBR14] Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, and Kurt Rothermel. PLEROMA: A SDN-based high performance publish/subscribe middleware. In *Proceedings of 15th International Middleware Conference*, 2014.

[TKK+11] Muhammad Adnan Tariq, Boris Koldehofe, Gerald Georg Koch, Imran Khan, and Kurt Rothermel. Meeting subscriber-defined QoS constraints in publish/subscribe systems. *Concurrency and Computation: Practice and Experience*, 2011.

[TKKR09] Muhammad Adnan Tariq, Boris Koldehofe, Gerald Koch, and Kurt Rothermel. Providing probabilistic latency bounds for dynamic publish/subscribe systems. In *Proceedings of the 16th ITG/GI Conference on Kommunikation in Verteilten Systemen (KiVS)*, 2009.

[TKKR12] Muhammad Adnan Tariq, Boris Koldehofe, Gerald G. Koch, and Kurt Rothermel. Distributed spectral cluster management: A method for building dynamic publish/subscribe systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2012.

[TKR13] Muhammad Adnan Tariq, Boris Koldehofe, and Kurt Rothermel. Efficient content-based routing with network topology inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, 2013.

[tOMG] OMG: Object Management Group. Common object request broker architecture specification. `http://www.omg.org/spec/CORBA/`.

[Van91] George Vaněček, Jr. BRep-Index: A multidimensional space partitioning tree. In *Proceedings of 1st ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, 1991.

[VPMB14] Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. Effective switch memory management in openflow networks. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, 2014.

[VRKS06] Spyros Voulgaris, Etienne Rivière, Anne-Marie Kermarrec, and Maarten Van Steen. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *the 5th International Workshop on P2P Systems*, 2006.

[WQA+04] Yi Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J. Wang. Subscription partitioning and routing in content-based

publish/subscribe systems. In *Proceedings Of International Symposium on Distributed Computing*, 2004.

[YG16] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, 2016.

[ZJ13] Kaiwen Zhang and Hans-Arno Jacobsen. SDN-like: The next generation of pub/sub. *CoRR*, 2013.

[ZKV13] Ye Zhao, Kyungbaek Kim, and Nalini Venkatasubramanian. Dynatops: A dynamic topic-based publish/subscribe architecture. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, 2013.