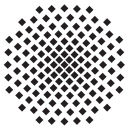


Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Addressing TCAM limitations in an SDN-based Pub/Sub System

Alexander Balogh

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	M.Sc. Sukanya Bhowmik
Commenced:	2016-10-24
Completed:	2017-04-24
CR-Classification:	C.2.1,C.2.4

Abstract

Content-based publish/subscribe is a popular paradigm that enables asynchronous exchange of events between decoupled applications that is practiced in a wide range of domains. Hence, extensive research has been conducted in the area of efficient large-scale pub/sub system. A more recent development are content-based pub/sub systems that utilize software-defined networking (SDN) in order to implement event-filtering in the network layer. By installing content-filters in the ternary content-addressable memory (TCAM) of switches, these systems are able to achieve event filtering and forwarding at line-rate performance. While offering great performance, TCAM is also expensive, power hunger and limited in size. However, current SDN-based pub/sub systems don't address these limitations, thus using TCAM excessively.

Therefore, this thesis provides techniques for constraining TCAM usage in such systems. The proposed methods enforce concrete flow limits without dropping any events by selectively merging content-filters into more coarse granular filters. The proposed algorithms leverage information about filter properties, traffic statistics, event distribution and global filter state in order to minimize the increase of unnecessary traffic introduced through merges.

The proposed approach is twofold. A local enforcement algorithm ensures that the flow limit of a particular switch is never violated. This local approach is complemented by a periodically executed global optimization algorithm that tries to find a flow configuration on all switches, which minimized to increase in unnecessary traffic, given the current set of advertisements and subscriptions. For both classes, two algorithms with different properties are outlined.

The proposed algorithms are integrated into the PLEROMA middleware and evaluated thoroughly in a real SDN testbed as well as in a large-scale network emulation. The evaluations demonstrate the effectiveness of the approaches under diverse and realistic workloads. In some cases, reducing the number of flows by more than 70% while increasing the false positive rate by less than 1% is possible.

Acknowledgements

At this point, I would like to express my deep gratitude by acknowledging all those, without whom this thesis would not be possible. First and foremost, I wish to thank Prof. Dr. Kurt Rothermel for giving me an opportunity to do my thesis in the Department of Distributed Systems and in an exciting and interesting topic.

I am also immensely grateful to my supervisor, Sukanya Bhowmik for her permanent involvement and support throughout the duration of this project. Not only did I profit greatly from her expertise but also from her constant positive attitude that helped me through this thesis. I am wishing her all the best for her further career, especially for the near-term completion of her Ph.D.

Moreover, I wish express my thankfulness towards Dr. Muhammad Adnan Tariq, for providing me with his outstanding expert knowledge in the topic of pub/sub concepts, as well as critically questioning my work and supporting me with his crucial feedback.

I'm also grateful for my friends, for their support, encouragement and for being understanding of my absence during the more stressful stages of this thesis.

To all the people in the distributed systems lab: It was great sharing laboratory with you during last six months. Thanks for providing a fun work environment and many interesting discussions.

Furthermore, I want to express my very profound gratitude to my parents and to my brother for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Last but not least, I want to thank all the researches cited in this thesis for doing the hard work and creating the mountain of knowledge on which this work builds upon.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The *publish/subscribe* (pub/sub) paradigm is a well-established communication paradigm that enables information to flow from information producers (*publishers*) to information consumers (*subscribers*) in a manner, that is decoupled in time and/or space [1]. More specifically, subscribers specify what kind of information they are interested in, i.e. they subscribe to certain information. Publishers on the other hand disseminate information by publishing events. The pub/sub system (also referred to as notification service [2] or event-based system [3]) acts as an intermediary between publishers and subscribers and is responsible for routing published events to all subscribers whose subscriptions match the given event. Therefore, neither publishers, nor subscribers need to be aware of each other. Subscribers express their interest according to a filter model (also called subscription model). A filter is a pattern defined over the event space that matches a subset of all possible events. Many different filter models exist, such as channel-based, topic-based, type-based and content-based, varying in expressiveness and complexity. This thesis focuses on content-based pub/sub systems due to the expressiveness and widespread usage of this filter model.

Pub/sub systems find application in a wide range of domains. This includes applications for information dissemination (financial data, news, RSS-data, etc.), system and network monitoring, Internet of Things (IoT) and Enterprise Application Integration (EAI) [3, p. 4 ff.]. Many of these applications put challenging requirements on the pub/sub middleware implementations. In particular:

- Supporting a large and constantly changing set of publishers and subscribers
- Delivering a large number of events with low end-to-end latency (sometimes even with real-time characteristics) and high throughput
- Keeping bandwidth consumption at a minimum

This puts great demands for scalability and performance on the design of the pub/sub system. In order to fulfill these requirements, many pub/sub systems employ a distributed architecture where the physically centralized broker is replaced through a set of brokers (also termed routers) connected through an overlay network. However, the broker network is still logically centralized, i.e. it appears as a single entity to the publishers and subscribers. In order to avoid unnecessarily forwarding events within the overlay, content-filters are installed in the broker network. Content-filters define what events get forwarded to which broker. The

bandwidth efficiency of content-based pub/sub systems largely depends on the expressiveness of the content-filters. Implementing content-based pub/sub systems via an overlay broker network doing event routing and filtering in the application layer has been a popular model for implementing pub/sub systems in the past [4][2][5][6, 7][8]. However, this approach comes with two major flaws. Firstly, while the brokers may use information from the underlying network in order to optimize routing paths, the incompleteness of these information leads to topology mismatches between the logical overlay network and the underlying physical network. This topology mismatch results in imperfect routing decisions causing unnecessary bandwidth consumption. Secondly, event filtering in the application layer and de-/serializing events between network and application layer at each hop comes at the expense of end-to-end latency and throughput.

To overcome these limitations, systems that leverage the power of *Software-Defined Networking* (SDN) have recently been proposed in literature [9][10][11]. SDN enables applications to easily shape networks by injecting custom control logic into the network layer. The aforementioned systems make use of this by installing content-filters directly in the *ternary content-addressable memory* (TCAM) of switches, thus effectively pushing event-routing and filtering logic from the application layer into the network layer. The fast hardware-based routing and filtering of events directly within the network layer allows these systems to achieve event-forwarding at line-rate performance.

SDN makes this possible by untangling the, formerly tightly integrated, control plane of the network layer from the data plane and making it accessible to applications by providing straightforward APIs. The control plane is implemented as a logically centralized *controller*. The controller has a consistent global view on the network, thus eliminating the problem of topology mismatch between underlay and overlay network mentioned above. However, the control plane may be physically distributed for scalability reasons [12]. As Zhang et al. [13] assert, the dichotomy of control and data plane in SDN also fits nicely with the pub/sub model. Subscriptions and advertisements are handled by the control plane while events get forwarded in the data plane. OpenFlow [14] is the de facto standard for realizing SDN and supported by many hardware switches. OpenFlow introduces the notion of *flow entries* that can be installed in TCAM by the controller. A flow entry specifies what packets it matches (based on layer 2-4 headers) and what should be done with a matched packet. In order to implement a SDN-based pub/sub system aforementioned systems transform content-filters into OpfenFlow flow entries.

Unfortunately, hardware-based matching in the network layer does not come without drawbacks. TCAM is a scarce resource. In most switches, the number of available flow entries is limited to only a couple of thousand entries [15] that may even need to be shared between multiple applications. Moreover, TCAM is also significantly more expensive and has notably higher power consumption and heat generation compared to DRAM/SRAM [16]. Therefore, applications should use TCAM sparingly. Limiting TCAM usage is a well discussed topic in

literature [15][17][18][19]. However, in the context of SDN-based pub/sub systems, TCAM limitations didn't find much consideration yet. Reducing the size of the TCAM available for the pub/sub system effectively reduces the amount of content-filter information that can be encoded in flows / TCAM, thus reducing the expressiveness of the filters. As mentioned above, less expressive filters lead to higher bandwidth usage due to unnecessarily forwarded traffic. This trade off between TCAM size and expressiveness / bandwidth-efficiency has already been explored in [20] by looking at it from the perspective of single flow entries. The authors outline how filter information can be efficiently encoded within the limited memory available for a single flow entry. However, the proposed algorithms do not reduce the overall amount of flow entries in TCAM.

Currently, no SDN-based pub/sub system with pure network-based filtering enforces any limits on the amount of installed flows. Hence, current systems may deploying a large amount of flow entries in TCAM rendering them impractical in environments where TCAM resources are scarce. The goal of this thesis is to extend PLEROMA [9, 10], a SDN-based pub/sub system such that concrete float limits can be enforced. PLEROMA uses the concept of spatial indexing in order to convert content-filters into a binary representation which is encoded into the IP-header match field of OpenFlow flow entries. Likewise, content information of events is encoded in their IP-destination header. This allows fast prefix-based matching of events against content-filters in TCAM of OpenFlow-enabled switches. The main contribution of this work are algorithms that reduce the required number of flows by intelligently combining (*merging* flow entries (respectively the content-filters they represent) while trying to keep the increase of bandwidth waste due to loss of expressiveness at a minimum. To achieve this goal, the proposed algorithms exploit information about subscriptions, advertisements, network state and event distribution.

1.1 Thesis Organization

The remaining part of this thesis is structured as follows:

Chapter 2 provides more background on the aforementioned topics pub/sub and SDN. It gives an overview over different types of pub/sub systems, the concepts of SDN and how they can be applied to a real network using the popular OpenFlow standard.

Chapter 3 discusses the current state of the art regarding pub/sub systems by doing a brief survey over current pub/sub implementations. Furthermore, it gives an overview over approaches for reducing TCAM usage in various application contexts.

Chapter 4 contains the problem statement and introduces PLEROMA [9, 10], a content-based pub/sub system using SDN. This thesis is an extended work of PLEROMA and therefore heavily utilizes the notations and algorithms introduced in this chapter.

Chapter 5 introduces algorithms for enforcing flow limits on PLEROMA. This chapter contains the main contribution of this work.

Chapter 6 describes how the algorithms previously introduced in Chapter 5 have been realized and evaluated on a real SDN testbed as well as on an emulated network environment. The conducted experiments mainly focus on the amount network false positives and performance. Above that, this chapter offers analysis and discussion of the findings.

Finally, Chapter 7 provides brief summary of the work and a conclusion. Furthermore, it proposes an outline for possible future works.

Chapter 2

Background

This chapter provides the necessary background to this thesis. In particular, the publish/-subscribe paradigm and software-defined networking are discussed.

2.1 Publish/Subscribe Paradigm

Pub/sub systems are inherently driven by information and not by the identities of the producers and consumers of said information [21]. Producers of information initiate communication by publishing events that are delivered by the pub/sub middleware to potentially many subscribers that are unknown to the publisher. Therefore, pub/sub follows a one-to-many communication model. As aforementioned, a key aspect of pub/sub is that it allows producers to share information with consumers in a loosely coupled manner. Following Eugster et al. [1], this includes in particular:

- **Space decoupling**: publishers and subscribers do not need to know each other. All communication between the two groups is mediated by the pub/sub middleware.
- **Time decoupling**: the life cycle of individual publishers and subscribers does not depend on each other. Publishers and subscribers can join and leave the system at any time without affecting others in any way. Some pub/sub implementations also offer the possibility for subscribers to consume information, which has been published while they were disconnected from the system.
- **Synchronization decoupling**: events are sent asynchronously. Publishers are not blocked till a published event is consumed. Subscribers are notified asynchronously and consume events independent from other subscribers.

Therefore, the pub/sub paradigm allows the dissemination of information between a large and dynamic set of autonomous and heterogeneous components [22].

In the following, some key aspects of the pub/sub communication paradigm are presented.

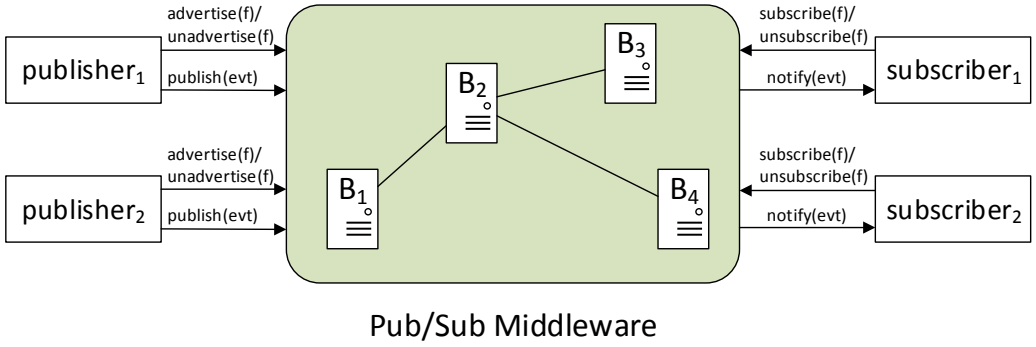


Figure 2.1: A pub/sub system

Components As mentioned before, a pub/sub system knows the roles of *publishers* and *subscribers*. Publishers are the source of information. They disseminate information by publishing *events*. Subscribers on the other hand are consumers of information. They can specify in what kind of information they are interested. The concept of publishers and subscribers is orthogonal to physical hosts or applications. An application can act as a publisher and as a subscriber at the same time. The *pub/sub middleware* acts as an intermediary between publishers and subscribers. It is responsible for delivering published events to interested subscribers. The pub/sub middleware itself may be implemented as a centralized component or as a distributed system. A popular model to implement a distributed middleware is as a set of *brokers* connected by an overlay network [4]. Implementations come in different topologies and can range from static networks to dynamic peer-to-peer based approaches [23]. Figure 2.1 shows a pub/sub architecture. The middleware is implemented as a network of brokers but acts as a single entity towards publishers and subscribers (i.e. it is logically centralized).

Operations In a pub/sub system, publishers and subscribers only interact with the pub/sub middleware, not directly with each other. The pub/sub paradigm requires only a small set of operations. Subscribers express their interests with the *subscribe(f)* operation where f is a filter that describes the set of events that the subscriber is interested in. Subscribers can revoke their subscriptions with the corresponding *unsubscribe(f)* operation. Publishers on the other hand can publish information in form of *events* (also called event notifications) using the *publish(evt)* operation [24, p. 245 ff.]. The structure of events is defined by the data model of the pub/sub system. Events can, among others, be expressed as attribute/value pairs, objects and semi-structured data like XML [25]. Additionally, some systems offer a *advertise(f)* operation and the corresponding *unadvertise(f)* operation for publishers. Advertisements specify the set of possible events that the publisher may publish and are again expressed in terms of a filter. Advertisements are primarily used to implement an efficient event delivery mechanism in the pub/sub middleware [24, p. 245 ff.].

Filters Filters are an integral part of pub/sub systems. A filter is a pattern defined over the event space that can be matched against events. As such, a filter can be expressed as a boolean function that takes an event as input and returns true if the event matches the filter and otherwise false [3, p. 13]. Another way to look at filters is that they select a subset of all possible events, which contains all events that are matched by the filter (or alternatively, the filter defines a subspace of the event space). How exactly filters are specified, i.e. the *filter model* depends on the structure of the events, i.e. the *data model*. Possible types of filter models are further described in the next section.

2.1.1 Pub/Sub Models

Channel-based and topic-based pub/sub In a channel based model, publishers publish events to named channels. Channels are categorized in a flat collection. Subscribers on the other hand can subscribe to arbitrary channels. This model is used for example by the CORBA Event Service [26]. Filters cover all events that belong to a particular category. The topic-based approach extends the channel concept by introducing a hierarchical arrangement of topics. Filters match topics and all subtopics. For example, a publisher can subscribe to the topic "news.sport" and will receive events from the topics "news.sport.football", "news.sport.basketball", etc. Topics are also often referred to as subjects [24, p.246 f.].

Type-based pub/sub In a type-based model, events are objects situated in a type hierarchy. This allows filtering based on type or subtype relationships. As such, type-based filtering is similar to topic-based approaches. Both depend on the arrangement of events into categories. An object-based approach can extend the type-based approaches to include filtering based on the attribute values of events [24, p.247]. This makes them similar to content-based approaches.

Content-based pub/sub In a content-based pub/sub model filters can be defined as constraints over the actual content of events, i.e. the information carried by the event. This allows very expressive and flexible filters. The specific filter model depends on the structure of the events, i.e. the data model. The following lists some possible combinations of data and filter models:

- **XML/XPath**: events are modeled in XML. One advantage is, that already existing technologies for processing XML can be leveraged. For example, filters can be defined as XPath queries [27].
- **Tuples/Templates**: events are modeled as tuples, i.e. an ordered list of attributes (e.g. ("weather", "sensor4", 15°C, 70%)). Corresponding filters are expressed as templates. A template has the same number of attributes and can specify concrete values or wildcard (*, don't care), e.g. ("weather", "sensor4", *, *) will match all weather data events of sensor4. This concept is for example used in LINDA [28].

- **Attribute-Value Pairs/Conjunctive Attribute Filters**: events are modeled as attribute-value pairs (e.g. (type="weather", sensor="sensor4", temperature="15°C", humidity=70%)). In contrast to tuples, attribute-value pairs allow optional attributes. A filter can be defined as the conjunction of multiple attribute filters. Attribute filters define a boolean condition over a single attribute and typically support a range of operations. String typed attributes can for example be tested for equality, matching prefix/suffix or containment. Numeric values can be evaluated against concrete values or value ranges. A possible filter matching the above event is $\{(\text{type} = \text{"weather"}) \wedge (\text{temperature} \in [10,30]) \wedge (\text{humidity} > 50\%)\}$ [3, p.36 ff.].

2.1.2 Event Delivery Mechanisms

Multicast and Subscription Clustering One way to deliver events is by grouping subscribers together in multicast groups. In order to deliver an event, it just needs to be send to the correct multicast group (or to multiple groups), using IP-multicast where it is available. The members of a multicast group can either be the subscribers themselves or brokers, which deliver the received events to all connected subscribers. In channel-based pub/sub, the mapping from subscriptions to groups is straightforward - every channel gets a corresponding group. A possible multicast strategy for topic-based pub/sub is to map every topic and topic-subtree to a group. Implementation of multicast for a content-based pub/sub system is much more complex and a popular research topic (e.g. [29][30][31][32]). A popular theme is in order to use a clustering approach to group similar subscribers (or brokers) together. This can be done for example based on similarity of subscriptions or geographical proximity [33].

Flooding Flooding works by propagating events to all subscribers and filtering locally at the subscribers. In case of a distributed implementation of the pub/sub middleware as a network of brokers, events can also be propagated by flooding through the broker overlay network. The advantages of flooding are that it is easy to implement and that it maps well to multicast [24, p.250 f.]. Since events are sent to every subscriber, flooding often produces a lot of unnecessary traffic, especially in the presence of expressive and diverse subscriptions. However, if most events should be propagated to most subscribers anyway, flooding is a very efficient implementation strategy [8].

Content-Based Routing In content-based routing, events delivered by selectively forwarding them through a network of content-based routers. The routers make routing decisions based on the content of the events, i.e. based on content-filters installed on the router. If the routers are situated in the application layer and connected through an overlay network, they are commonly referred to as brokers. There are various different approaches that try to balance the accuracy and performance of content-filters against the overhead of maintaining the content-based routing tables. Generally, if new a new subscription is added to the system, brokers need to be made aware of that. A simple approach floods every subscription (respectively the corresponding filters) to ever broker. More advanced techniques avoid forwarding all

filters, for example by not forwarding a filter if an identical filter is already in place (*identity-based routing*), or if a new subscription is already covered by an existing filter (*covering-based routing*), i.e. if all events matched by the new filter will also be matched by an existing one [3, p. 80 ff.]. Another technique is, to merge multiple filters into a more coarse granular filter and propagate the aggregated filter instead of the individual filters (*merge-based routing*) [8]. Finally, some systems also propagate advertisements to the brokers. Therefore, brokers need to only forward subscriptions to places where overlapping advertisements exist [4]. A concern of content-based routing is also, to avoid cyclic forwarding and the delivery of duplicates. This can be easily achieved using acyclic topology. However, acyclic topologies are often plagued by single point of failures and performance bottlenecks [23].

2.2 Software-Defined Networking

The core idea of Software-Defined Networking (SDN) is to provide a programmable network. One way to look at network devices is the perspective of control plane and data plane. The control plane is the decision module of a switch or router. It manages the network topology, access control and is responsible for establishing routes, i.e. routing protocols that write the routing tables of a router, like the Border Gateway Protocol (BGP) [34], are implemented in the control plane. The data plane is responsible for forwarding packets.

In traditional network devices, control and data plane are tightly coupled and integrated into the network device. The control plane logic, implemented a low-level language in the firmware of the switch / router, is typically quite complex due to the big large amount of protocols that it needs to support and often a vendor specific and closed platform. This makes it hard and expensive to implement new network functionality, even for the vendor [35].

This leads to the fact that many systems rely on the 'one size fits all' solutions, provided by traditional network devices. However, many systems could greatly profit from control plane logic customized to their specific applications and needs. A popular example of that is Google, which has successfully used custom control plane in order to achieve high bandwidth utilization and fault tolerance in the WAN connecting its datacenters by deploying a large SDN infrastructure [36].

SDN makes this possible by providing clean APIs for implementing custom control logic in high level languages. For this purpose, the control plane is extracted from the network devices and implemented in a separate (logical) centralized component named *controller*. The controller has an integrated view of the whole network and can make routing decisions based on that knowledge. Figure 2.2 shows a SDN architecture. The controller itself implements the kernel and the communication with the data plane. It can be a single entity as depicted or distributed but *logically centralized* component, e.g. for scalablility reasons [12]. There are many popular open source controller implementations available, for example Floodlight [37] and NOX [38]. The controller provides a so called *northbound interface* that can be used by the control code modules in order to interact with the network. The actual form of the northbound interface is not standardized and depends on the controller implementation. Controllers may offer a web API, a modular plugin system or control logic can be implemented directly in the source

code of an open source controller [39]. But the general idea is always, that control logic can be implemented in a modular way. Therefore, one can use modules that implement standard network routing protocols like BGP or OSPF as well as modules that implement more specific algorithms customized to once needs, e.g. in order to implement a SDN-based firewall or pub/sub system.

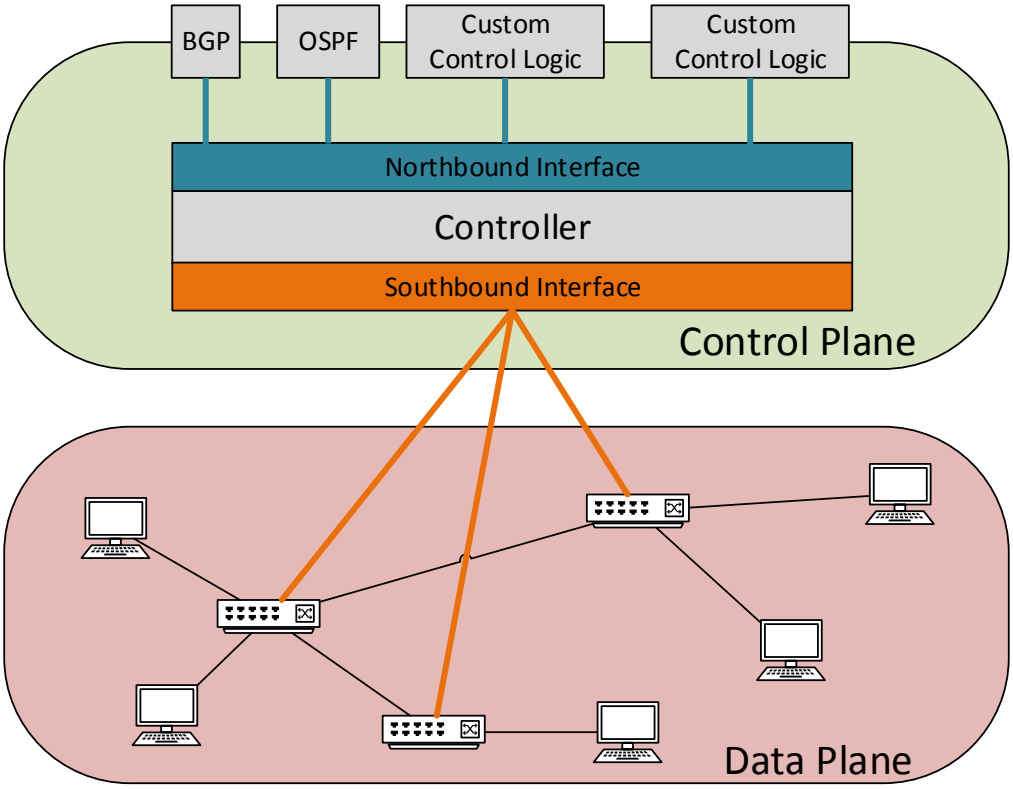


Figure 2.2: Architecture of a Software-defined network

The *southbound interface* of the controller is used for the communication between controller and SDN-based switches. The controller is connected to each switch over each southbound interface. Over this connection, the controller can gather network statistics, inject packets into the network and modify the state of the switches. More specifically, SDN uses the concept of *flow entries* that can be installed in *flow tables* of switches. Similar to routing tables, flows in flow tables match incoming packets and then perform an action specified by the flow entry (e.g. forwarding, dropping or modifying the packet). One can differentiate two routing approaches. In a proactive routing approach, all necessary flows are installed from the very beginning. In a reactive routing approach on the other hand, switches forward packets that aren't matched by any flows to the controller. The controller then determines the route, installs the according flows and injects the packet back into the network. Following similar packets do not need to

be sent to the controller again. OpenFlow [14] is the de facto standard for the southbound interface. It is developed by the Open Networking Foundation and backed by many big players such as Cisco, Verizon, Deutsche Telekom, Microsoft, Google and Facebook. The following section describes the OpenFlow protocol in detail.

2.2.1 OpenFlow

As aforementioned, OpenFlow [14] is the de facto standard for the southbound interface of a SDN architecture. OpenFlow specifies a protocol for the communication between controller and switches and how OpenFlow-enabled switches need to behave in the context of an OpenFlow-based network. This section describes the parts of the OpenFlow standard, more specifically OpenFlow version 1.3.x, that provide the relevant context for this thesis.

In OpenFlow, each switch has at least one flow table. The controller can install, modify and delete flow entries inside a flow table. An incoming packet matches exactly zero or one flow entries in a table. Once a packet is matched, the actions defined by the respective flow entry are executed by the switch. In case of multiple flow tables a pipeline-based processing is possible. However, only one flow table is used in the context of this thesis. Figure 2.3 depicts the general structure of a flow entry.

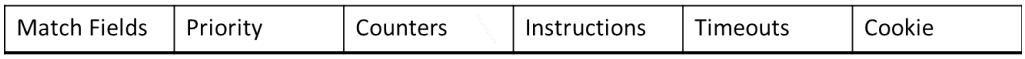


Figure 2.3: Structure of an OpenFlow flow entry

The **match fields** are used to match packets against flow entries. One can match against a specific ingress port and a set of several layer 2-4 header, e.g. source and destination IP, source and destination port, VLAN ID, ... For each match field, the flow can specify a concrete value or wildcard (often denoted as *), i.e. the flow does not care about the value of said match field. One can for example implement routing by matching against certain IP-addresses or match all traffic to the http standard port 80 regardless of destination IP and MAC.

Priority is an integer value that acts as a tie breaker in case an incoming packet matches against multiple flow entries. The packet is handled by the flow entry with the highest priority.

Counters are updated, whenever the packet is matched by this flow entry.

The **instructions** describe what the switch should do if a packet matches this flow. Most importantly, one can define a set of actions that should take place when the flow is matched. Possible actions are for example to drop the packet, forward it through one or multiple ports and/or modify the packet headers.

The **timeouts** field can be used to define an optional timeout (total time or idle time) after which the flow is automatically deleted.

Finally, the **cookie** is a value that can be freely set by the controller for its own internal purposes.

One can specify a table miss flow entry that matches all flows with lowest priority. Otherwise, the specified or default table miss action (e.g. drop or forward to controller) is taken if a packet matches none of the flow entries.

Furthermore, OpenFlow specifies a large number of optional and required statistics that switches need to keep track of. This includes statistics on flow tables, flow entries, ports, groups, etc. Notable examples are received packets and received bytes counters for ports and flows. The statistics can be queried by the controller on demand.

2.2.2 Content-Addressable Memory

Classical random access memory (RAM) allows the lookup of data stored at a specified memory address. Content-addressable memory (CAM), also called associative memory, inverts this principle. It allows searching for a specific data word (content) in a tabular memory and returns the memory address of the matching data and/or additional data associated with it. CAMs are highly optimized for these kind of search operations. In fact, CAM is able to search the complete memory in a single clock cycle by comparing the queried data against every CAM entry fully in parallel. Therefore, CAMs allow extremely fast search operations, however this comes at the expense of increased cost and power consumption due to a more complex architecture using a much larger number of circuits [40].

Binary CAM allows searching for arbitrary binary words, e.g. '10011010'. Ternary CAM (TCAM) additionally allows to mask or wildcard some bits, denoted as '*'. While searching, it does not matter if the state of that bit is 0 or 1, e.g. searching for '11*00' will search for both '11000' and '11100'. This makes more sophisticated searches possible while maintaining the same performance of simple binary CAMs. However, this comes at the expense of additional cost and power consumption [41].

Find application where fast searching is necessary, e.g. in pattern matching [41] or fast routing table lookup [42]. In the context of SDN, TCAM memory allows fast matching of packets against flow entries.

Chapter 3

State of the Art

3.1 Pub/Sub Systems Survey

3.1.1 SIENA

SIENA (Scalable Internet Event Notification Architectures) [4] is a content-based pub/sub system that aims to scale to wide-area networks while maintaining expressiveness. Events are expressed as set of typed attribute value pairs. SIENA supports a predefined set of common primitive types such as string, temporal and numeric types among others. Subscriptions can be expressed as conjunctive attribute filters, i.e. a set of predicates over single attributes (e.g. amount > 30) that must be all true in order to match an event. Furthermore, subscribers can define *patterns* of events that they want to receive by given an ordered list of filters. For example, if the pattern specifies that f_1 must be matched before f_2, all events that come before any event matching f_1 will be dropped. After an event matching f_2 has been received, everything but events matching f_2 will be dropped.

Events are routed through a network of brokers placed in the application layer. In order to only forward events to places where it is necessary, SIENA employs content-based filtering at the brokers, i.e. brokers match incoming events against a local subscription based routing table and forward events accordingly. To establish routing tables, two approaches are supported. The first approach floods subscriptions to all brokers. The second approach introduces an advertise-operation. Advertisements are flooded in the system and subscriptions are only forwarded to brokers with matching advertisements. Both approaches additionally only forward subscriptions that are not covered by existing subscriptions (cf. covering-based routing, 2.1.2).

All in all, SIENA is able to provide an expressive filter model and even more advanced event selection mechanism, such as matching event patterns, while maintaining scalability. However, its end to end delay is inherently limited due to the application layer based filtering approach that is uses.

3.1.2 LIPSIN

LIPSIN (Line Speed Publish/Subscribe Inter-Networking) [43] is a topic-based pub/sub system, suitable for large-scale networks. LIPSIN achieves line-rate performance by employing a novel network-based event-forwarding strategy, based on multicast. The architecture separates a control and a data plane. The data plane's responsibilities are packet forwarding, error correction and traffic scheduling. The main purpose of the control plane is to make routing decision. Therefore, a topology manager is implemented in the control plane that has a consistent global view of the network. The global view is build during an initial bootstrapping phase and continuously maintained. Furthermore, the control plane contains a rendezvous system that brings together subscribers and publishers.

In order to achieve delivery of events at line-rate performance, LIPSIN assigns a binary link identifier to each link in a specific manner. The control plane then creates spanning trees for topics and encodes all links of the tree in a bloom filter [44]. This bloom filter can be matched against any link id using a fast binary AND operation. The result will tell us if the link is contained in the bloom filter or not. When an event is published to a topic, the bloom filter associated with this topic is encoded in the packet header of the event. Therefore, all routing information is contained in the packet. Forwarding nodes simply need to compare the ids of their links against the bloom filter of an incoming packet and forward it over all matched links.

LIPSIN provides a scalable pub/sub approach that is able to achieve dissemination of events at line-rate performance. However, it uses a topic-based pub/sub model that lacks the expressiveness of content-based systems. Furthermore, the matching of links against the bloom filters is not perfect and will introduce false positives in the system.

3.1.3 Gryphon

Gryphon [45] is a content-based pub/sub system that is focused on scalability, availability and security. Gryphon supports a centralized pub/sub middleware or can scale out to a large distributed broker network. Gryphon uses an efficient application layer based approach to match events against subscriptions, outlined by Aguilera et al. in [46]. Subscriptions are arranged as leafs in parallel match tree. The inner nodes of the tree are attribute filters, i.e. they test a single attribute of an event against a value, a value range, etc. Events are matched against the nodes starting from the root. The next node is chosen depending on the outcome of the test at the current node. A single subscription can be matched by following a path from the root to a leaf. At any point, all subscriptions in the subtree, with the current node as its root, match all before-tested conditions. With this technique, events can be matched against all subscriptions in sub-linear time (in the number of subscriptions).

A simple way to implement routing through a network of brokers would be to match events against all subscriptions at each hop, using the aforementioned matching technique. However, Gryphon uses an more efficient solution, described in [47]. The general idea is to annotate the match tree with routing information at each broker. Assuming that an event has been

matched up till a certain node, the broker knows over which links the event definitely should be forwarded, over which it maybe should be forwarded and over which it definitely should not be forwarded. The deeper the matching algorithm descends into the merging tree, the more links can be classified as definitely forward / definitely not forward. Therefore, the match tree is only needed to be matched partially at every broker, thus increasing the performance. By sending events to all "definitely forward" and "maybe forward" links, no false negatives can occur. Sending events over "maybe" links can however introduce false positives, i.e. unnecessary traffic.

Gryphon presents an interesting and high-performance approach for content-based routing in an application-layer broker overlay for content-based pub/sub. However, it cannot match the performance of a hardware-based approach.

3.1.4 Rebeca

Rebeca [8, 48] is a content-based pub/sub system using a broker overlay architecture. Rebeca supports advertisement propagation and identity and covering-based routing among others. Since its initial development in 1999, Rebeca has provided as base for various research efforts. The implementation of merge-based routing [8, 49] is of particular interest to the topic of this thesis. In merge-based routing, brokers combine multiple filters to a new aggregated filter and propagate only the aggregated filter to other brokers. Mühl et al. explore perfect merging, as well as imperfect merging, although latter only briefly. In the case of perfect merging, the aggregated filter matches only events that are matched by at least one of the individual filters. An imperfect merge on the other hand results in an aggregated flow, that matches some events, which none of the original filters match.

Although the reduction of filter table sizes is one of the goals of merge-based routing in Rebeca, Rebeca does not address the problem of enforcing concrete limits on the number of filters, because it does not face the problem of limited hardware. Therefore, its actual implementation of merge-based routing is rather limited. It allows only i) merging of filters with the same destination and ii) perfect merges. However, in the context of limited TCAM space, flow limits quotas cannot be met reliably by performing only such restrictive merges. Especially, if multidimensional event spaces and diverse and expressive subscriptions make perfect merges unrealistic.

3.2 Addressing TCAM Limitations

3.2.1 Hybrid Content-Based Filtering

In [17], Bhowmik et al. face the problems of limited TCAM capabilities in a SDN-based pub/sub system. In order to reduce the number of flows that need to be installed in TCAM, they propose a hybrid approach where some of the content-filters are installed in the network layer and some in the application layer. The approach tries to combine the positive attributes

of both approaches while trying to minimize the negative aspects. Application-based filtering is slow but highly expressive. On the other hand, network-based filtering is very fast but less expressive due to the constraints on the amount of memory available to filters and therefore forwards more unnecessary traffic. Bhowmik et al. propose means for selecting which filters should be installed in the network layer and which should be installed in the application layer, such that bandwidth usage and end to end delay is minimized.

CacheFlow [50, 51] provide a general solution for transparently caching flows in the application layer and thereby creates the illusion of a large TCAM size. CacheFlow selects the flows that are cached in the application layer based on popularity of flows and dependencies between flows.

Obviously, hybrid approaches that offload some of the flows to the application layer can reduce and limit the amount of used TCAM. However, this comes at the expense of increased end to end delay for all those events that are routed over the application layer. Since the goal of this work is to not compromise on delay and throughput, the proposed hybrid approaches are not applicable.

3.2.2 Flow Entry Eviction

A common approach deal with a limited number of flows is to remove old flow entries in order to install new ones. This can either happen once the TCAM is exhausted and a new flow should be installed or pro-actively before the flow limit is reached.

One popular way to achieve this is by the means of flow entry timeouts. OpenFlow offers a timeout feature, which makes it possible to automatically discard flows after a certain idle time or total time. Furthermore, applications can implement their own timeout mechanism and remove timed out flows themselves. SmartTime [19] proposes a heuristic for choosing adaptive idle timeout values such that flow table misses are minimized. A similar approach is proposed by Zhu et al. with Intelligent Timeout Master [52], which aims to predict good timeout values using a history based approach. Furthermore, timeout values are adjusted according to the current load factor of the flow table in order to prevent exhausting the available TCAM.

FlowMaster [53] tries to detect stale flows using a Markov based learning predictor and pro-actively deletes those flows. Additional to the aforementioned timeout mechanisms, SmartTime also provides FIFO and random based flow eviction in case of imminent TCAM exhaustion [19].

However, in the context of SDN-based pub/sub, removing flows and the therewith associated flow table misses would lead to dropped events (i.e. false negatives) or forwarding of the unmatched event to the application layer. Both is not compatible with the goals of this thesis.

3.2.3 Optimized Rule Placement

Some research has been done regarding the optimal placement of rules in a SDN environment. For this purpose, [18] differentiates routing and endpoint based policies, that both need to be encoded in flow. Routing policies define the paths that packets should take through the network. Endpoint policies, e.g. load balancing and access control, on the other hand only care about the packets received at an endpoint. Based on that, Kang et al. try to distribute the rules over multiple switches along a path such that the amount of needed flows is minimized. Palette [54] follows a similar approach. vCRIB [55] can route traffic purposefully over a longer path in order to ensure sufficient TCAM capacity to apply all endpoint policies.

However, the proposed techniques don't help to reduce the TCAM usage of the problem at hand, because SDN-based pub/sub uses only routing based policies that need to be installed at every switch along the path. Furthermore, these algorithms do not work if the number of rules exceeds the combined TCAM capacities of all switches. Nevertheless, rule placing optimization could be useful for balancing the TCAM usage of SDN-based pub/sub with other applications that share the same TCAM.

Chapter 4

PLEROMA

This section introduces PLEROMA [9, 10], a SDN-based pub/sub system, in detail. This work extends the PLEROMA system, since PLEROMA has proven to achieve event forwarding at line-rate performance under scale but does not address TCAM limitations. This section explains the parts of the PLEROMA system necessary to understand the extensions proposed by this work.

PLEROMA uses a content-based filter model over a d-dimensional event space. Events are expressed as attribute value pairs, e.g $event = \{price = 25, weight = 200, height = 2\}$. They are represented as a point in the event space. Participants can express their advertisements and subscriptions as conjunctive attribute filters, e.g $adv = \{price \geq 20 \wedge 500 \leq weight < 1000\}$. An advertisement / subscription represents a subspace of the event space. PLEROMA requires that publishers send only events that are covered by a previously announced advertisement.

PLEROMA converts advertisements and subscriptions to a binary representation called *dz-expression*. The dz-expressions are then encoded in the match fields of OpenFlow flow entries and deployed into the TCAM of OpenFlow-enabled switches. Events are converted to dz-expressions as well and encoded in the packet header of the packet transmitting the event. While routing through the network, the dz-expression of events can be matched against the dz-expression encoded in flow entries allowing fast hardware based prefix filtering and routing in TCAM.

4.1 Converting Events and Filters to Dz-Expressions

In order to convert content filters and events to binary dz-expressions PLEROMA uses *spacial indexing*. The concept of spatial indexing is widely used in the context of geospatial databases to allow fast geo-based queries [56]. The idea of simple spatial indexing is to recursively divide the event space into regular subspaces along each dimension. Each dz-expression represents a subspace of the event space. Longer dz-expressions define smaller subspaces. Figure 4.1 shows a simple two dimensional event space. Both dimensions, dim_1 and dim_2, allow values in range $[0, 100]$. The first subfigure shows the event space, identified by the empty dz-expression {*}. The next step shows all subspaces defined by dz-expressions of length 1 (({0}, {1}). The next step all subspaces defined by dz-expressions of length 2 ({00}, {01}, {10}, {11}) and so forth. A very important property of this spatial indexing technique is prefix based containment. A subspace $space_A$ identified by dz_A is contained by another subspace $space_B$ identified by dz_B

iff dz_B is a prefix of dz_A. For example, in figure 4.1, the subspace identified by dz-expression {00} contains all subspaces, that are identified by dz-expressions starting with 00, e.g. the subspaces defined by {000} and {001}. We use the notation $dz_B \succ dz_A$ iff dz_B covers dz_A, i.e. dz_B is a prefix of dz_A. We say that two dz-expressions are related iff they are overlapping, i.e. they are either equal or $dz_B \succ dz_A$ or $dz_B \prec dz_A$.

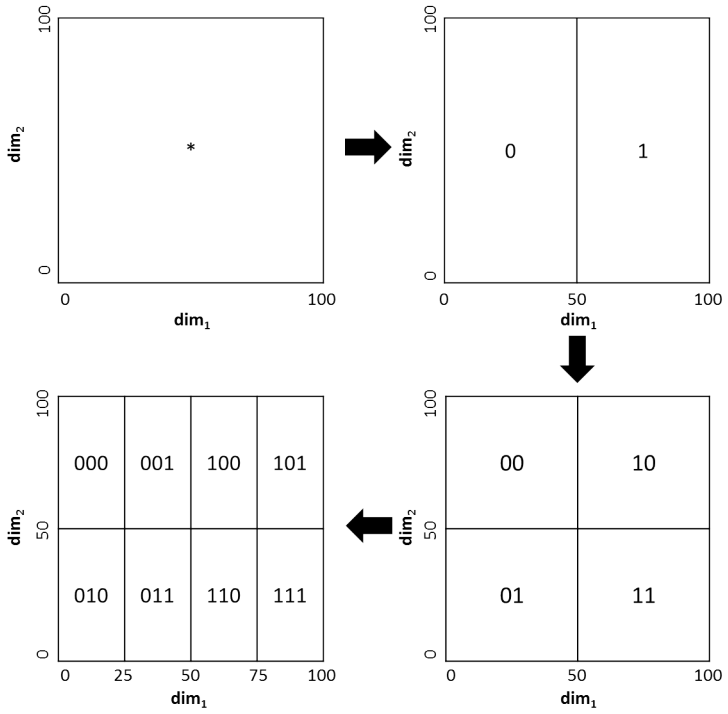


Figure 4.1: Decomposition of the event space into subspaces identified by dz-expressions

Events are approximated by the smallest subspace that they fall in. Subscriptions and Advertisements are represented as a single dz-expression or multiple dz-expressions, for example in the event space illustrated in figure 4.1, the subscription $\{dim_1 = [0, 25] \wedge dim_2 = [0, 50]\}$ can be represented as dz-expression {010}. On the other hand, the subscription $\{dim_1 = [0, 25] \wedge dim_2 = [0, 100]\}$ can only be expressed by two dz-expressions: {000, 010}. But not all subscriptions can be expressed so nicely. For example, subscription $\{dim_1 = [0, 10] \wedge dim_2 = [50, 100]\}$ cannot be perfectly expressed using only dz-expressions up to length 3 as depicted in the figure. It has to be approximated by dz-expression {000}. With unlimited long dz-expressions, arbitrary small subspaces can be defined, that better and better approximate advertisements / subscriptions and especially events, which are just points in the event space. However, the length of the dz-expression is constraint, since only a limited amount of bits can be encoded in the flow entries. Bhowmik et al. proposed several methods, e.g. a workload-based indexing approach that try to provide a more expressive binary encoding of advertisements / subscriptions than the simple spatial indexing aforementioned [20]. The algorithms proposed by this work assume simple spatial indexing. However, the proposed concepts can be directly adapted to other approaches, especially if they honor the

containment constraints.

4.2 Encoding Dz-Expressions in Flows and Packets

PLEROMA encodes the dz-expressions in the IP header match field which allows prefix based matching in order to support IP address matching based on Class-less Interdomain Routing (CIDR). It's possible to encode the dz-expression both in IPv4 and IPv6 match fields. For IPv4, the 225.128.0.0/9 address space is used (i.e. 225.128.0.0/225.255.255.255). The first 9 bits of the IP address are fixed. The dz-expression is appended after the first 9 bits. When not all of the 32 bit of the IPv4 address are used, the remaining bits are masked, i.e. they are set to match anything (wildcard / *). Figure 4.2 shows the fixed prefix and the conversion of several dz-expressions to IPv4 address ranges, displayed both in binary and CIDR-notation. The dz-expression part of the result is marked in red in the binary representation. Using CIDR-notation, dz-expression {110} is converted to the address space 225.224.0.0/12. Dz-expression {1101000}, which is contained by {110}, has the CIDR-representation 225.232.0.0/16. Note that all IP-addresses in range 225.232.0.0/16 are also contained within the 225.224.0.0/12 space. However, none of the IP-addresses that belong to dz {0011} (225.152.0.0/13) are part of the subnet specified by dz-expression {110}. This confirms exactly the expectations of prefix based matching. Respectively, the dz-expression of an event is encoded in the IP-destination header of the packet that transmits it.

Instead of IPv4, it's also possible to use IPv6 based matching. In this case, the IP-multicast address space (ff00::/8) is used. Other than that, the procedure is equivalent to the IPv4 encoding [11].

	Binary Notation	CIDR-Notation
IPv4 Fix Prefix	11100001.1*******.********.********	225.128.0.0/9
dz={110}	11100001.1110****.********.********	225.224.0.0/12
dz={1101000}	11100001.11101000.********.********	225.232.0.0/16
dz={0011}	11100001.10011***.********.********	225.152.0.0/13

Figure 4.2: Conversion of dz-expressions to IPv4 addresses

4.3 Installing Flow Entries

Hosts can announce new (un-) advertisements / (-un) subscriptions by sending them to a special IP-address IP_{fix}. No flow entry that matches IP_{fix} will be installed on the switches, thereby sending all packets that have this destination to the controller. In order for events being routed from publishers to subscribers, flows need to be installed along the paths from

publishers to subscribers in the network. PLEROMA maintains a dynamic set of spanning trees. Each tree is responsible for a disjoint event subspace and connects every publisher and subscriber that publishes or receives events of that subspace. Publishers and subscribers can be part of multiple such dissemination trees. Flows whose dz-expression belongs to a certain subspace can only be installed to the switches contained in the respective spanning tree. Therefore, it's not possible that events are routed in cycles.

This work uses a simplified model of PLEROMA, where only a single spanning tree is used. The tree is thereby responsible for the whole event space and connects every publisher and subscriber. All installed flows only forward events within his tree. However, the tree itself can change over time when hosts join or leave the system. Whenever the controller receives a new subscription (respectively advertisement) it performs the following steps:

1. It looks up all advertisers (subscribers) that have overlapping advertisements (subscriptions) with the new subscription (advertisement). To achieve this, the controller keeps track of all advertisements and subscriptions in the system.

2. For each identified relevant publisher (subscriber), the controller determines the route from publisher to subscriber (there is only one route in a tree).

3. Along each of these routes route, the controller installs flows such that events matching the advertisement and subscription are forwarded from the publishers to the subscribers.

In order to establish the flows along each route, PLEROMA tries to install a new flow on each switch of the route whose dz-expression is the overlap between advertisement and subscription of this path. If there are overlapping flows already installed on a switch, adjustments to the new flow and/or existing flows may be necessary. The process is described in detail by algorithm 1. In essence, if a flow f_i partially covers another flow f_j, then f_j must, additional to its own egress ports, also forward over the egress ports of f_i. f_j has to be deployed with a higher priority. Hence, events that match f_j (and therefore also f_i) are forwarded by f_j. Events that match only f_i are forwarded by f_i.

4.4 Problem Statement

As previously discussed, existing SDN-based pub/sub systems achieve line-rate performance and scalability by installing content-filters in the TCAM of network switches. However, these systems in general and PLEROMA in particular lack means to limit the amount of flow entries installed in TCAM. Nevertheless, constraining the number of flow entries is necessary because TCAM is a scarce and expensive resource. Therefore, existing systems need to be extended such that the number of flow entries installed on each switch can be constraint to an upper limit. Reducing the number of entries available to the system typically results in a loss of expressiveness, thereby increasing unnecessary traffic in the system. Therefore, the main objective of this thesis is to propose algorithms that reduce the number of flows until a given limit is respected, while keeping the increase of bandwidth wastage at a minimum. Bandwidth wastage is measured in terms of network false positives. Whenever an event is forwarded over a

Algorithm 1 Installing a new flow on a switch

```
1: procedure InstallFlow(flow_new, switch)
2:     if ∃flow ∈ switch.flows : flow fully covers flow_new then
3:         return
4:     end if
5:     for all flow_existing ∈ switch.flows do
6:         if flow_existing partially covers flow_new then
7:             flow_new.egressPorts ← flow_new.egressPorts ⋃ flow_existing.egressPorts
8:         end if
9:         if flow_new partially covers flow then
10:            flow_existing.egressPorts ← flow.egressPorts_existing ⋃ flow_new.egressPorts
11:        end if
12:    end for
13:    delete newly fully covered flows
14:    install flow_new and modify all changed existing flows in TCAM
15: end procedure
```

link, although it shouldn't, it counts as a network false positive. The proposed algorithms are integrated into the PLEROMA pub/sub system and evaluated under a realistic workload.

Chapter 5

Enforcing Flow Limits on Switches

This chapter is dedicated to algorithms that enforce flow limits on switches within the PLEROMA system. The approach thereby is twofold. Firstly, this work proposes a *global optimization algorithm*, that takes the network topology and the current set of advertisements and subscriptions (respectively the corresponding flow entries on each switch generated by PLEROMA) as input and produce a set of flow entries for each switch, such that:

1. The flow limit constraints of each switch are respected on each switch
2. All events are correctly delivered, i.e. no false negatives can occur
3. The amount of unnecessary traffic (false positives) is minimized, i.e. the flow configuration is bandwidth-efficient

Since the global optimizer needs to produce the final flow set for every switch and has a significant complexity, its execution time is in the order of (tens of) seconds. If advertisements and subscriptions change rarely, one can get by with using only this algorithm, by running the global optimizer whenever a new advertisement or subscription would violate the flow limit constraints.

However, many pub/sub systems are highly dynamic and have to deal with constantly changing subscribers, publishers, subscriptions and advertisements. In this situation, it's not feasible to run the global optimizer every time a change would violate flow limit constraints due to its slow execution time. This motivates a second class of algorithms termed *local flow limit enforcers*. These algorithms will be executed whenever a change on a single switch would violate its constraints and only perform merges on said switch. While these algorithms should also make good decisions regarding the bandwidth-efficiency, their main design criterion is execution time. This is because they need to run very often and also because new advertisements/subscriptions should come into effect as fast as possible. Therefore, the execution time of these algorithms is essential for the scalability of the system.

Both classes of algorithms cooperate well. The local flow limit enforcer ensures that flow limits are respected at all time while the global optimizer can be executed periodically in the background in order to optimize the current set of flows. While the optimizer is executed, the system remains completely functional. Even new advertisements/subscriptions can be added during the execution. They just need to be re-added after the global optimizer terminates since they are not included in the start state of the optimizer.

The remaining part of this chapter is structured as follows: Firstly, some general concepts and data structures that are relevant for all algorithms are introduced. Followed by an investigation of the forces that are at play when merging flows in a bandwidth-efficient manner. Then, two flavors of the global optimization algorithm are presented, as well as two different local enforcers. Finally, two complementary components are introduced. Firstly, a component responsible for consistently deploying the flow changes induced by the aforementioned algorithms without violating flow limit constraints and without temporally spiking false positives (or false negatives). Secondly, a component that allows the pub/sub systems to determine the ratio of network false positives on its own. With the help of this component, the system is able to autonomously adapt parameters or trigger execution of the global execution algorithm when the false positive ratio surpasses certain thresholds.

5.1 General Concepts

5.1.1 Merging Flows

All algorithms proposed in the thesis work on the basis of merging flows. This means, that they start with a set of flows, that violates flow limit constraints, and then reduce the number of flows by selectively combining them. The merge operation takes k flows $f_1, ..., f_k$ as input (with $k \geq 2$) and produces a single flow f_m as output such that f_m *fully covers* all flows $f_1, ..., f_k$ (cf. chapter 4).

Note that it is only possible to merge flows that match on the same ingress port, since OpenFlow only supports matching either a specific ingress port or *any* ingress ports (wildcard).

Performing a $merge : f_1, ..., f_k \rightarrow f_m$ is done as follows:

1. Set the ingress port of f_m equivalent to the ingress port of $f_1, ..., f_k$ (they must be identical).

2. Set the dz-expression of f_m to the longest common prefix of all dz-expressions of $f_1, ..., f_k$. If there is no common prefix set the dz-expression of f_m to wildcard (*), i.e. match every event.

3. Set the egress ports of f_m to the union of the egress ports of $f_1, ..., f_k$

As an example, merging the two flows $f_1 = (1 \rightarrow 0000 \rightarrow 2,3)$ and $f_2 = (1 \rightarrow 0011 \rightarrow 3)$. The resulting merged flow will be $f_m = (1 \rightarrow 00 \rightarrow 2,3)$. The dz-expression of f_m is $\{00\}$ is the longest common prefix of the dz-expressions of f_1 and f_2: $\{0000\}$ and $\{0011\}$. The merging of these two dz-expressions in a two-dimensional event space is visualized in figure 5.1. The areas highlighted in green show the parts of the event space covered by the dz-expressions of f_1 and f_2. The area framed in red is the part of the event space covered by the dz-expression of f_m. On can easily see, that the dz-expression of f_m covers more than just the sum of those belonging to f_1 and f_2. We call this an imperfect merge.

A perfect merge on the other hand is a merge, where f_m covers exactly the same part of the event space as the sum of all the participating flows $f_1, ..., f_k$ and every participating flow has also the same egress port(s). This means, that all events matched by f_m would have been also matched by at least one of the flows $f_1, ..., f_k$ and is forwarded to exactly the same egress ports. For example, merging $f_1 = (1 \rightarrow 000 \rightarrow 2,3)$ and $f_2 = (1 \rightarrow 001 \rightarrow 2,3)$ is a perfect merge whose result is the flow $f_m = (1 \rightarrow 00 \rightarrow 2,3))$. Figure 5.2 depicts this merge in a two-dimensional event space from the dz-expression perspective.

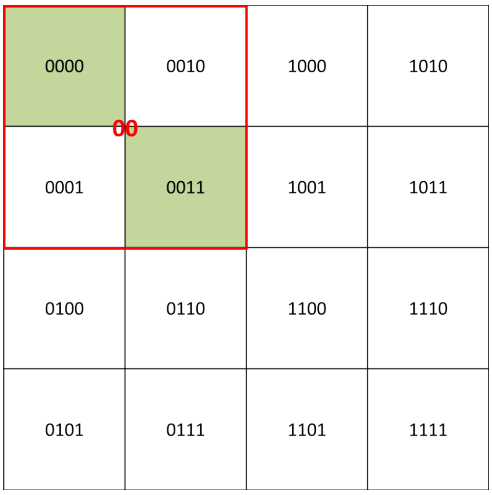


Figure 5.1: Merging of dz-expressions 0000 and 0011

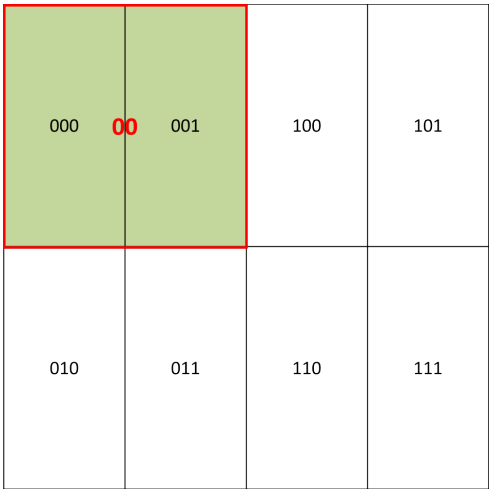


Figure 5.2: A perfect merge of dz-expressions 000 and 001

5.1.2 Network False Positives

This section investigates the reasons why and how the merging of flows introduces new false positives into the system. Consider the merging of $f_1 = (1 \rightarrow 000 \rightarrow 2,3)$ and $f_2 = (1 \rightarrow 011$

$\rightarrow$ 3) into $f_m = (1 \rightarrow 0 \rightarrow 2{,}3)$. Figure 5.3 depicts this situation. Events that are matched by none of the original flows $f_1, ..., f_k$ but are matched by f_m will produce a false negative on every outgoing link that f_m forwards to. In this specific example, every event that is located in the red shaded part of the event space will trigger two false positives at this switch. E.g. $event_1$ will trigger two false positives because f_m forwards it over port 2 and 3, while without the merging the packet would have been dropped, since neither f_1, nor f_2 match it.

The situation gets trickier, when an event would have been matched by one or more of the original flows. In case of $event_2$, no false positives will be introduced by the merge because the egress ports of flow f_1 that also matches $event_2$ are identical to the egress ports of f_m. Therefore, $event_2$ will be forwarded over port 2 and 3 no matter if the flows get merged or not. However, for $event_3$ the situation is different. Without merging, it would only be forwarded over port 3. Yet, after merging, it is forwarded over port 2 and 3, thereby introducing one new false positive into the system.

False positives introduced at one switch can lead to further false positives downstream.

Figure 5.3: False positives introduced due to merging

5.1.3 Merge Tree

This section introduces the *merge tree* data structure that is widely used by the algorithms of this work, in order to efficiently identify possible merges and perform merges. A merge tree is a radix tree [57] with flows as values and their respective dz-expression as keys. A radix tree is a compact form of a prefix tree (also called trie)[58] where empty intermediary nodes are merged into their parent nodes. Figure 5.4 depicts such a tree. The value of each node (pictured as circles) is a dz-expression. More specifically, it's the longest common prefix of the value of its children. The boxes represent the flows that have been inserted into the tree structure. A node can either be empty intermediaries as it is the case for nodes '00' and '000', or they can have a flow associated with them, e.g. nodes '01' and '011'. Each of the nodes represents a *merge point*. When merging at a specific node / merge point, all flows of

the subtree whose root is the merge point get merged, i.e. they get removed from the switch and replaced by a single new aggregated flow f_m. The dz-expression of f_m is the value of the merge point. For example, merging at node '000' will merge flows ($1 \rightarrow 00001011 \rightarrow 2$) and ($1 \rightarrow 000111 \rightarrow 4$) into $f_m = (1 \rightarrow 000 \rightarrow 2{,}4)$. Merging at node '01' would merge all five flows contained in this subtree. Figure 5.5 shows the same merge tree from figure 5.4 after merging at the merge points '000' and '011'.

The *cardinality* of a merge point is defined as the number of flows that it merges, i.e. by merging a merge point with cardinality k, the number of flows on that switch is reduced by $k-1$ (k flows get deleted and one new flow is added). Since it is only possible to merge flows with the same ingress port (as discussed above), the system maintains a separate merge tree for every ingress port at each switch. Therefore, flows can be merged freely at every merge point in the tree.The maximum depth of a merge tree corresponds to the maximum length of a dz-expression.

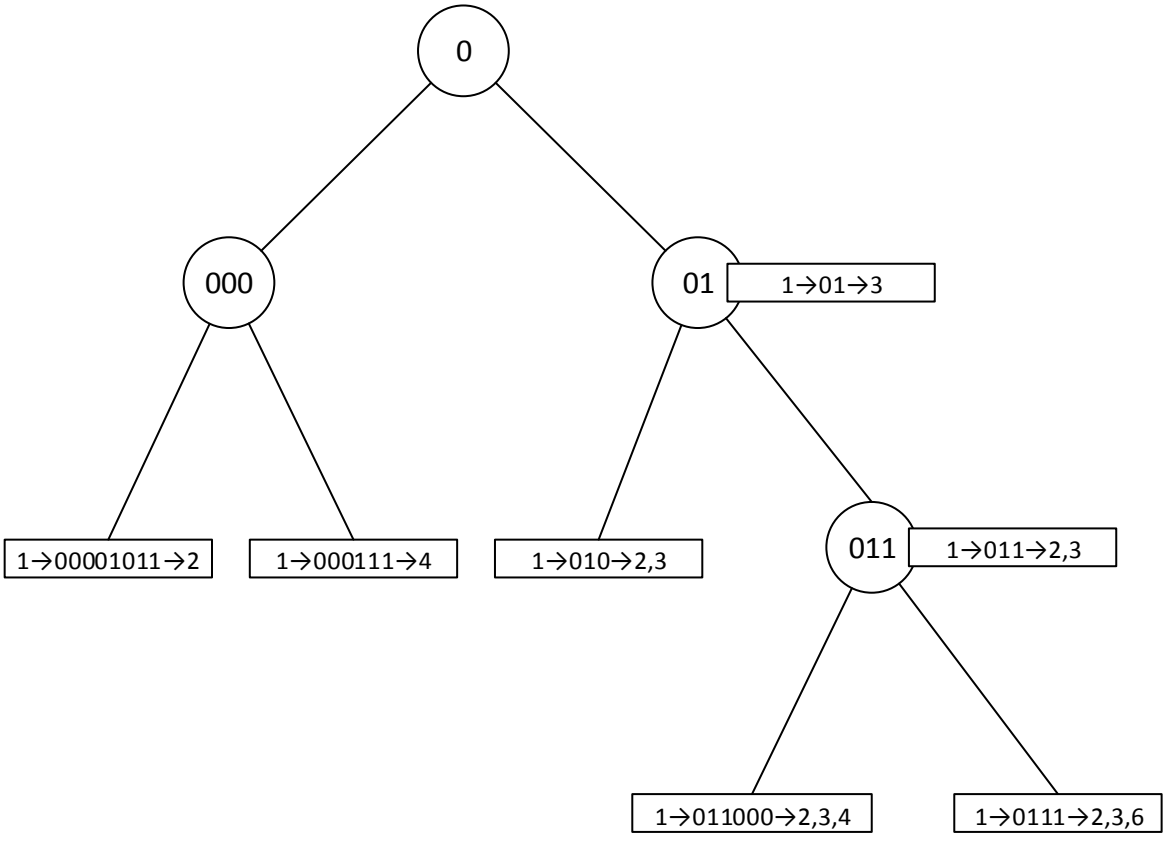


Figure 5.4: A merge tree containing several flows

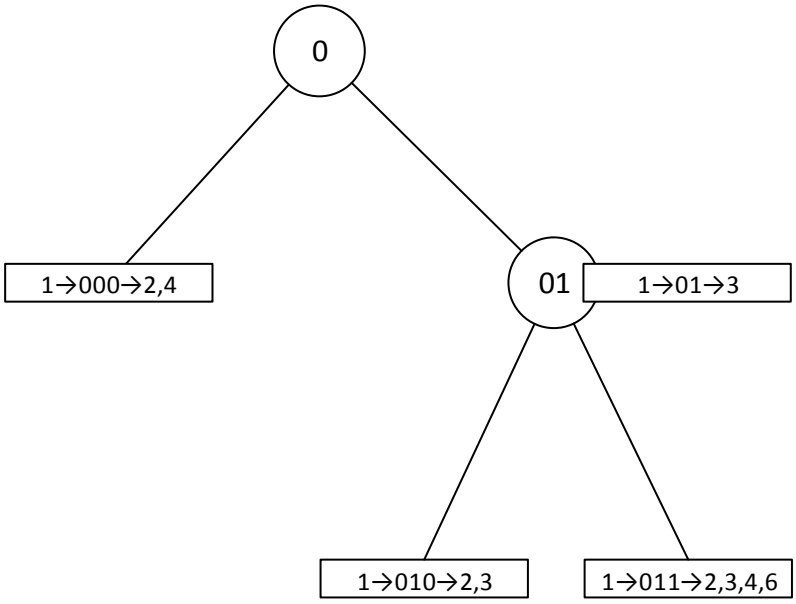


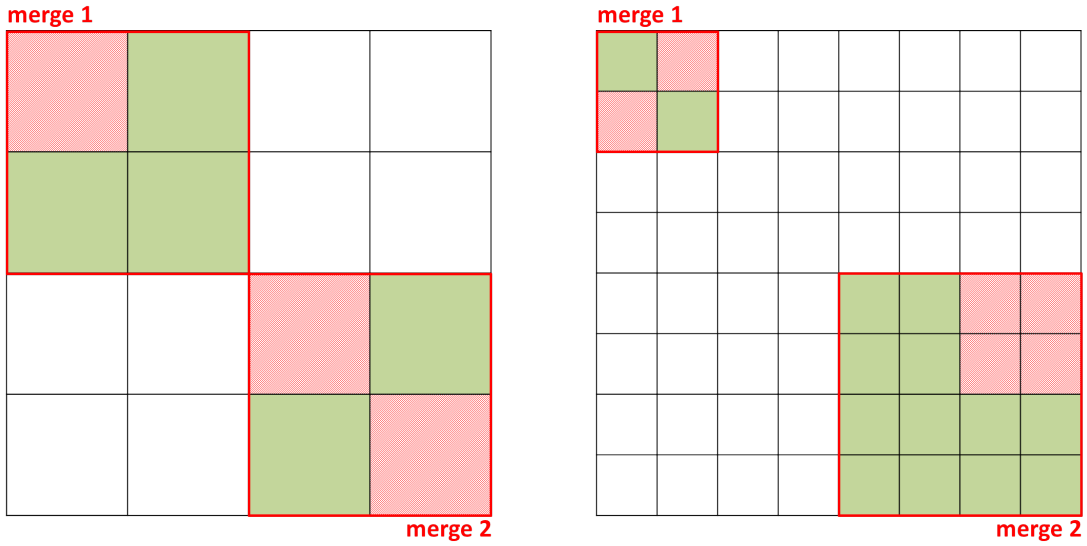
Figure 5.5: Merge tree after performing two merges

5.2 Forces

This section investigates the forces that are at play when trying to bandwidth-efficiently merge flows. This means it identifies the criteria that influence the design of the algorithms and describes how they influence the design

One major factor when looking at the causes of false positives is **space expansion**. As described in section 5.1.2, f_m may cover some parts of the event space that none of the original flows covered before. Events that fall into this space will definitely introduce false positives. We call the increase of covered event space due to a merge *space expansion*. It seems like a sensible idea to minimize the size of this expansion. In order to investigate the influence of space expansion in isolation, we assume all flows participating in the merge have the same egress ports. Figure 5.6 depicts multiple merge options. The green area is respectively the portion of the event space covered by the original flows. The red outlined part of the event space covered by the resulting flow of the merge and the red shaded area is the problematic 'wrongfully' covered area. Two factors are at play here. Figure 5.6a shows the effect of relative size expansion. Merge number one seems to be preferable, since only a quarter of the space covered by the merge result is 'wrongfully' covered; compared to merge number two where the ratio is split half and half. However, more important than relative size expansion is the total space expansion as figure 5.6b illustrates. While merge number one has a bigger relative size expansion, merge number two seems to be preferable. The reason is, that in case of merge

one only $\frac{2}{64}$ of the total event space is 'wrongfully' covered. The result of merge number two on the other hand 'wrongfully' covers $\frac{4}{64}$ of the event space.



(a) Influence of relative space expansion

(b) Influence of total space expansion

Figure 5.6: Influence of space expansion

Apart from the expansion of covered event space, the **port expansion** is also important. As discussed before in section 5.1.2, flows that participate in a merge can have different egress port sets. Events that fall into an area of the event space that is matched by f_m but none of the original flows always results in false positives being forwarded over all egress ports of f_m. But even if the event would have been matched by f_i, false positives are still produced on the egress ports that are part of f_m but not of f_i. Therefore, we consider the space expansion for each egress port individually.

Another major force is the **event distribution**. Minimizing space and port expansion works well if the events are distributed uniformly over the event space. However, if the event distribution is skewed, a small 'wrongfully' covered area can be responsible for a lot more false positives than one would expect due to its size. This is illustrated in figure 5.7. Past events are depicted as small stars. The majority of events occur in the lower right and the upper left corner of the event space. Under these circumstances merge number two will introduce a lot more false positives compared to merge number one. Although its size expansion is much smaller, it causes exactly the part of the event space where most events occur to be 'wrongfully' covered. Assuming that the event distribution does not change, merge number one is therefore preferable. Skewed event distributions are not uncommon. For example a fire detection system will frequently receive temperature data from sensors as events. Most events

will be in the range of normal room temperature and hopefully only very few events will be in the 'fire range'.

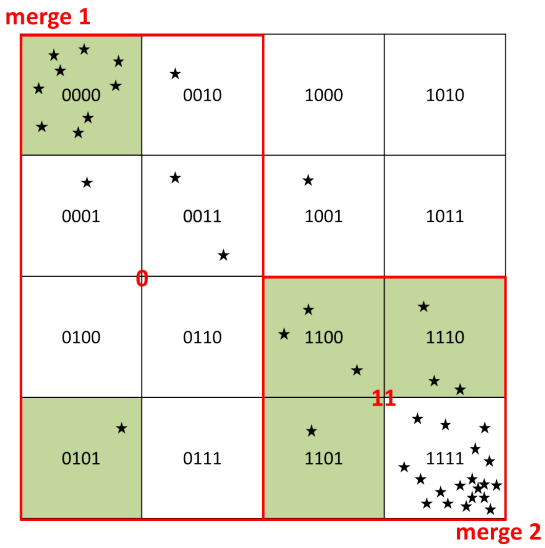


Figure 5.7: Influence of the event distribution on merge decisions

Till now, all forces were focusing on the state of the switch on which the merging takes part. However, we can not make good merge decisions in isolation since the quality of a merge decision is heavily influenced by the flows deployed on other switches of the network. This mutual influences between merge decisions on different switches adds major complexity to the bandwidth-efficient merge problem. There are two factors that need to be considered with regard to the **global flow state**. Firstly, it's important what events can even reach the switch. Therefore we look into the *upstream paths* of a switch, i.e. all paths from publishers to the switch. Events that get filtered out at one of the upstream switches never reach the switch on which we currently want to merge. In this regard, it's also important what kind of events the publisher can even produce (advertisements). In the situation depicted in figure 5.8 the two flows on each green switch can be merged into $f_m = (1 \rightarrow 00 \rightarrow 2)$ on all green switches without introducing any false positives at all. All events that would fall into 'wrongfully' covered subspace of the merge result (0001 / 0010) get filtered out at the red switch and will never reach the green switches. Secondly, it's important how far false positives will be propagated into the network if they are not stopped at the current switch in question. Therefore, we look into the *downstream paths* of a switch, i.e. all paths from the switch to the subscribers. False positives that are passed through at the current switch may be completely or partially filtered out again at some downstream switch. Or they are forwarded all the way to the subscriber. In figure 5.9 either the two purple outlined flows or the two blue outlined flows of the green switch should be merged. Both merge results will forward events from the publisher that the subscriber is not interested in. However, the events wrongfully forwarded by the result of the purple will be filtered out at the next switch (the red switch). In contrast,

an event wrongfully forwarded by the blue merge result will be propagated all the way to the subscriber, causing four instead of one network false positives on its way.

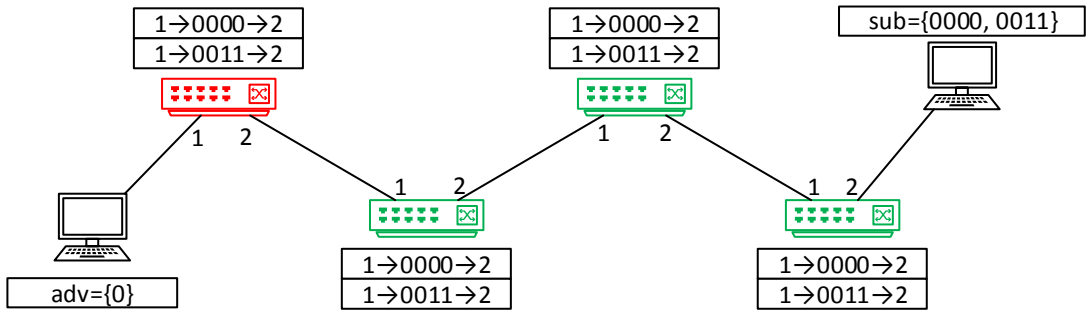


Figure 5.8: Influence of the upstream flows on merge decisions

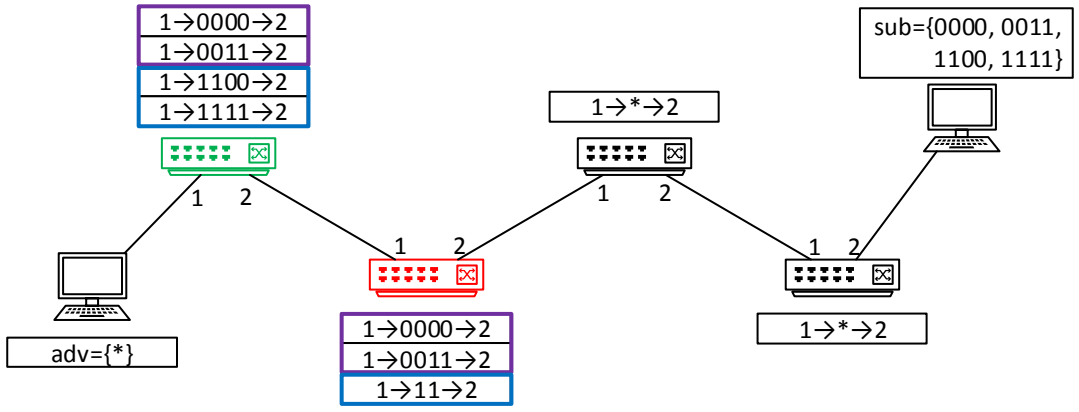


Figure 5.9: Influence of the downstream flows on merge decisions

Lastly, especially for the local flow limit enforcement algorithms, **execution time** is a major design factor. These algorithms are possibly executed whenever a flow is added to a switch due to a new advertisement / subscription and many pub/sub systems are quite dynamic by nature. Therefore, the execution time of these algorithms is an integral factor to the scalability of the whole system. Furthermore, new advertisements / subscriptions should come into effect as fast as possible. For the global algorithms, execution time is less important. But still, shorter execution time means that the global algorithms can be executed more frequently and that the system can adapt faster to changes.

5.3 Local Flow Limit Enforcement Algorithms

This section finally introduces two alternative algorithms of the local flow limit enforcement class. The purpose of these algorithms is to ensure that the flow limit constraint is never violated on any switch. The local flow limit enforcement algorithm is triggered whenever a flow should be installed on a particular switch. This means that the addition of a single new subscription or advertisement can trigger the execution of the local enforcement algorithm many times throughout the network. While the minimization of false positives introduced through merging is an important goal of these algorithms, their design is heavily constraint by the high requirements regarding the execution time of the algorithms. Because of their frequent execution and the desire that new subscriptions and advertisements come into effect as fast as possible, a short execution time is vital for the scalability of the system. Hence, the algorithms proposed in this section are rather simple.

5.3.1 Baseline Approach

This section proposes a simple flow limit enforcement algorithm. The algorithm is triggered whenever a new flow should be installed on a switch. Firstly, the algorithm checks if adding the new flow would violate the flow limit constraint of the switch. If not, the flow can be deployed and the algorithm terminates. If the flow limit is violated, it can only be violated by one, because the enforcement algorithm is triggered for every new flow. Therefore, it is only necessary to perform a single merge of two flows. The new flow is first added to the proper merge tree of the switch so that it is considered in the merge decision as well. Then, a random merge point is chosen among the leaf merge points of all merge trees (one merge tree per ingress port, cf. 5.1.3) of the switch. Leaf merge points are merge points that have no other merge points as children. They have a cardinality two or three (depending if there exists a flow with the same dz-expression as the merge point or not). Figure 5.10 shows a merge tree for the ingress port one. The leaf merge points are m_4, m_5, and m_6. After a merge point has been chosen, the new flow (if it was not part of the merge) and the flow changes of the merge are implemented in the switch (see section 5.5 for how to deploy flow changes without violating flow limit constraints).

5.3.2 Minimum Space Cost Function

This section presents an alternative algorithm that is an extension to the base line approach. The overall structure of the algorithm remains the same. However, instead of choosing a random merge point of cardinality two or three, all merge points with that cardinality are evaluated with a simple cost function. The merge point with the lowest cost is picked and gets merged.

Algorithm listing 2 describes the employed cost function. The idea is to minimize the space expansion on each egress port (cf. 5.2). For this purpose, the size of the 'wrongfully' covered area of the event space is computed as fraction of the size of the total event space for each

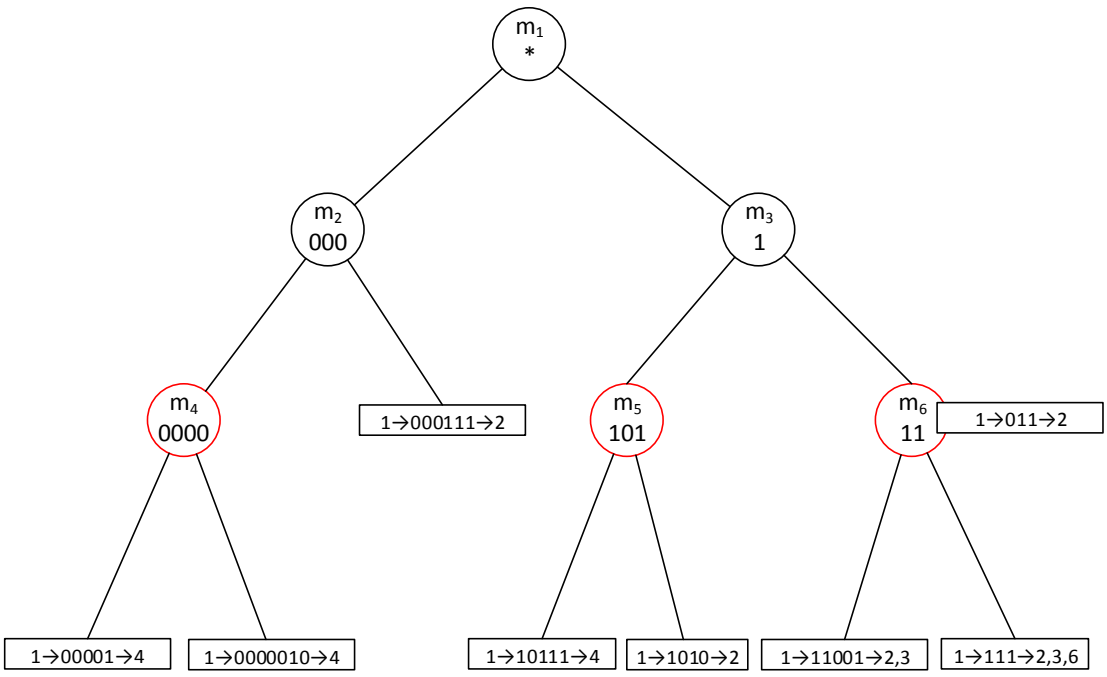


Figure 5.10: Merge points considered by local flow limit enforcement algorithm

egress port of f_m. The final cost of the merge point is the sum of the size expansion on all ports.

Note that the algorithm does not take into account the two of the major forces identified beforehand. Namely, the event distribution and the global flow state. This makes the execution of the algorithm fast but may lead to bad merge decisions.

5.4 Global Optimization Algorithm

A pub/sub system is fully functional and adheres to all flow limits by just using one of the local flow limit enforcement algorithms introduced above. However, motivated by the possibly bad decisions made by the local flow limit enforcement algorithm, this section proposes a global optimization algorithm. The optimizer can be triggered at any point by the system. The algorithm is global in a sense that it changes flows on every switch. Given the current subscriptions and advertisements, it tries to find the best possible combination of flows on each switch, possibly drastically changing the currently deployed flows on each switch. In contrast to the local algorithms this algorithm considers all forces mentioned in section 5.2.

The optimizer assigns costs to possible merge points with a cost function. There are two different cost functions proposed by this thesis. The main difference is how they incorporate

Algorithm 2 Minimum Space Cost Function

```
1: procedure ComputeMinimumSpaceCost(mp)                    ▷ mp is a merge point
2:     Flows ← flows that are merged by mp
3:     f_m ← resulting flow of merging mp
4:     cost ← 0.0
5:     for all port ∈ m.egressPorts do
6:         Flows_port ← {f ∈ Flows | port ∈ f.egressPorts}
7:         originalSpace_port ← event space covered by Flows_port
8:         newSpace_port ← event space covered by f_m
9:         spaceExpansion_port ← originalSpace_port − newSpace_port
10:        cost ← cost + size(spaceExpansion_port) ▷ size gives the size of the given space as
   fraction of the total event space
11:    end for
12:    return cost
13: end procedure
```

the event distribution. The *OpenFlow traffic based* approach relies mostly on OpenFlow counters to estimate the event distribution and assumes a uniform distribution in places where the information gathered by OpenFlow are not sufficient. The *event history based* cost function collects an event history in order to determine the event distribution. This allows building a better model of the event distribution, but comes at the expense of the additional overhead of collecting the history.

In the following the general structure and common parts of the optimization algorithm are introduced. Afterwards, both cost functions are described in detail.

5.4.1 Basic Algorithm

The optimizer operates on the complete flow set, i.e it first recreates the flow set that the system would like to deploy in the absence of flow limits on every switch. The flows are stored in merge trees. Then, the algorithm processes switch by switch. It reduces the amount of flows on the currently processed switch by selectively merging flows till the flow limit constraint is met. After that, the resulting flow set on this switch is finalized and the algorithm moves on to the next switch.

Processing Order of Switches

The merging decisions on one switch heavily depend on the decisions made on other switches. Since the algorithm processes each switch only once (per optimizer execution), the order in which the switches are processed is very important. When processing a switch, the algorithm can only rely on the flow state of the already processed switches, because their final state is certain. In contrast, the algorithm can only make assumptions about the final state of the

currently unprocessed switches. The optimizer, as described in algorithm listing 3, tries to process the switches in such an order, that it has the maximum possible knowledge about the upstream state when processing a switch. This means, that on every path that enters the switch, as many upstream switches as possible should already be processed. This allows the algorithm to judge, what events can reach the switch. To achieve this, the algorithm selects switches in rounds. In the first round all switches that are directly connected to a publisher are processed. In the second round all switches that are one hop away from a publisher are processed and so on. Of course, one switch might be directly connected to one publisher and much much farther down in the path from another publisher to some subscriber.

To establish ordering inside a group and resolve conflicts, a simple heuristic used. The heuristic chooses the switch with the least number of unprocessed upstream switches, i.e. for every path from some publisher to some subscriber through the switch, identify the upstream switches and take the one with the lowest number of unprocessed upstream switches. The idea of this heuristic is again to minimize upstream uncertainty.

Figure 5.11 depicts a scene with three switches. Black arrows show the paths on which events travel from publishers to the subscriber in the system. Both the red and the green switch are directly connected to a publisher and thereby in the group that is processed first. Since the red switch has no unprocessed upstream dependencies it is processed before the green one which has one unprocessed upstream dependency (namely the red switch). After the red switch is processed, the green switch gets processed. The blue switch is processed. In this simple example all upstream switches have already been processed when the algorithm processes a switch. Of course, 100 percent upstream certainty is not always possible, as figure 5.12 shows for example. No matter if the algorithm starts with the green or with the blue switch, there will always be two unprocessed upstream switches.

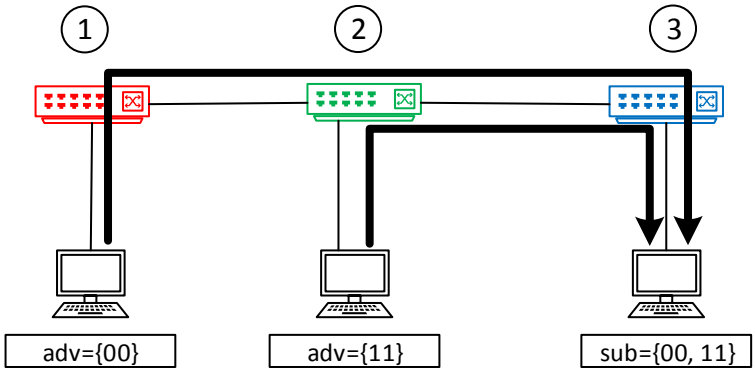


Figure 5.11: Choosing the processing order of switches

Processing a Single Switch

Algorithm 3 Global Optimization Base Algorithm

```
1:  procedure GlobalOptimization(mp)                    ▷ mp is a merge point
2:      U ← Set of all switches                         ▷ unprocessed switches
3:      i ← 0
4:      while U ≠ ∅ do
5:          currentGroup ← {switch ∈ U | switch is i hops away from a publisher}
6:          ProcessSwitchGroup(currentGroup)
7:          U ← U \ currentGroup
8:          i ← i + 1
9:      end while
10: end procedure
11:
12: procedure ProcessSwitchGroup(group)    ▷ process a group of switches all i hops
    away from a publisher
13:     while group ≠ ∅ do
14:         switch ← switch with minimum upstream dependencies
15:         ProcessSwitch(switch)
16:         group ← U \ {switch}
17:     end while
18: end procedure
```

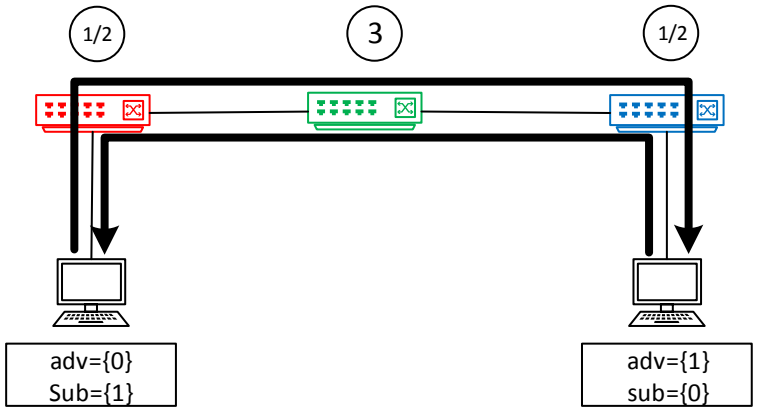


Figure 5.12: Processing order without complete upstream knowledge

This section discusses how the optimizer makes merge decisions on a single switch (cf. algorithm listing 4). Firstly, the algorithm computes the cost of each merge point. The cost function is the central piece of the global optimizer logic. The two cost functions that are proposed by this work are discussed in the following sections. After the cost assignment, the algorithm proceeds by deciding which merge points to merge. For this purpose, the *benefit* of each merge point is determined. The benefit equals the number of flows by which the flow count of the switch is reduced when merging at this merge point, i.e. the cardinality of the merge minus one. Based on that, the optimizer computes the *cost per benefit* = $\frac{cost}{benefit}$ of each merge point. The merge points are then selected using a greedy approach that merges the merge points in the order of increasing cost per benefit, i.e. the merge point with the lowest cost per benefit is merged first. However, merge points that would lower the number of flows more than necessary are not considered. This approach works well, because any merge points with a higher cardinality can also be achieved by a combination of merges with lower cardinality at the same cost.

Algorithm 4 Optimization of a Single Switch

```
1:  procedure ProcessSwitch(switch)
2:      M ← Set of all merge points of all merge trees of switch
3:      switch.flowCount ← bla
4:      switch.flowLimit ← maximum number of flows allowed on this switch
5:      for all m ∈ M do
6:          m.cost ← CostFunction(m)
7:          m.benefit ← |m| − 1            ▷ # of reduced flows = cardinality of m minus 1
8:          m.costPerBenefit ← m.cost / m.benefit
9:      end for
10:     while switch.flowCount > switch.flowLimit do
11:         m ← m ∈ M s.t. m.costPerBenefit is minimal ∧ switch.flowCount −
    m.benefit ≥ switch.flowLimit
12:         switch.flowCount ← updated flow count
13:     end while
14: end procedure
```

5.4.2 Traffic Based Cost Function

The traffic based cost function as described in algorithm listing 5 takes a merge point as input. The idea of the function is to directly reflect the number of false positives that would be introduced by merging at this merge point. More specifically, the computed cost equals the estimated number of false positives that would occur, because of this merge, based on past traffic statistics, collected with OpenFlow. Thereby, a sliding window approach is employed. Thus, only relatively up to date traffic statistics are used, allowing the system to adapt to changing traffic patterns by periodically executing the optimization algorithm.

The cost function follows a path based approach. For every path from a publisher that enters the switch through the ingress port of the merging flows, the cost is calculated individually

and then added together. The basic idea for computing the cost for a specific path (i.e. the false positives that are caused by events coming over this path) is to multiply the following three factors:

- Total amount of events published by the publisher
- Fraction of the total events that will cause false positives
- Number of network false positives a single false positive event causes, i.e. over how many links the event gets forwarded after leaving the switch

The following paragraphs describe in detail, how the traffic based cost function determines these three factors.

To determine the total amount of events published by the publisher, OpenFlow counters are utilized. In particular OpenFlow requires, that a switch needs to maintain a counter for each of its ports, which counts the number of received packets over this port. Every publisher is connected to the pub/sub system through exactly one port of an OpenFlow enabled switch. The publisher is exclusively responsible for all incoming traffic on this specific port. Therefore, the optimizer can determine the total amount of events send by the publisher within a certain window by periodically querying for the value of this counter.

In order to estimate the ratio of events that will cause false positives the algorithm determines the subspace of the event space that is covered by advertisements (*advertised space*) and the subspace of the advertised space whose events will cause false positives (*fp-space*). The algorithm assumes, that publishers publish events uniformly over their advertised space. Under this assumption, it can easily compute the fraction of events that will cause false positives by dividing the size of the fp-space by the size of the advertised space. To determine the fp-space, the algorithm first identifies the subspace of the advertised space whose events are forwarded all the way from the publisher to the switch, i.e. which events that are published by the publisher reach the switch (*input space*). Therefore, the algorithm follows the path from publisher to the switch, observing which events are filtered out at each step. The fp-space is the intersection between the 'wrongfully' covered space of the merge result (cf. 5.2) and the input space. Of course, this intersection needs to be done on a per egress port basis, as described in the port expansion section, because the 'wrongfully covered' space depends on the egress port.

Lastly, the algorithm needs to estimate the number of network false positives that a single false positive event can cause. Since the optimizer tries to process switches from publishers to subscribers, naturally not many of the downstream switches will be processed when processing a switch. This makes it hard to estimate how far false positive events forwarded by the current switch will spread into the network. Therefore, the algorithms makes a worst case estimation at this point. It assumes, that the event will be forwarded over every reachable downstream link.

Example: Figure 5.13 depicts a scenario for computing the costs of a merge point that merges two flows, namely $f_1 = (1 \rightarrow 0000 \rightarrow 2{,}3)$ and $f_2 = (1 \rightarrow 0011 \rightarrow 2)$. The result of this merge is $f_m = (1 \rightarrow 00 \rightarrow 2{,}3)$ and it takes place on switch number 2. There are two paths that connect relevant publishers with switch number 5.

Algorithm 5 Traffic Based Cost Function

procedure CostFunction(m) ▷ m is a merge point
  $f_m \leftarrow$ flow at merge point
  $dz_m \leftarrow$ dz-expression at merge point
  $switch_m \leftarrow$ Switch to which f_m should be deployed
  $P \leftarrow$ Set of all publishers
  $cost \leftarrow 0$
  for all $publisher \in P$ **do**
    $path \leftarrow$ Path from $publisher$ to $switch_m$
    if $path$ enters $switch_m$ through $f_m.ingressPort$ **then**
      $cost \leftarrow cost +$ CostOnPath($path$)
    end if
  end for
end procedure

procedure CostOnPath($path$) ▷ compute cost on a particular path
  $Flows_m \leftarrow$ flows that are merged by m
  $inputSpace \leftarrow \bigcap_{switch \in path} \left(\bigcup_{flow \in \{f \in switch.flows \mid f.ingressPort \in path \wedge \exists p \in f.egressPorts : p \in path\}} flow.dz \right)$
  $publisher_{path} \leftarrow$ publisher associated with $path$
  $advertisedSpace \leftarrow \bigcup_{adv \in publisher_{path}.advertisements} adv.dz$
  $traffic \leftarrow$ measured incoming traffic at the connection port of $publisher$
  $costOnPath \leftarrow 0$
  for all $port \in f_m.egressPorts$ **do**
    $Flows_{port} \leftarrow \{f \in Flows_m \mid port \in f.egressPorts\}$
    $affectedLinks_{port} \leftarrow$ number of links reachable in the subnet connected to $port$
    $wrongfullyCoveredSpace_{port} \leftarrow dz_m - \bigcup_{f \in Flows_{port}} f.dz$
    $fpSpace \leftarrow inputSpace \bigcap wrongfullyCoveredSpace$
    $fpRatio \leftarrow \frac{size(fpSpace)}{size(advertisedSpace)}$
    $costOnPath \leftarrow costOnPath + (traffic * fpRatio * affectedLinks_{port})$
  end for
  return $costOnPath$
end procedure

The first path starts at the upper publisher with advertisement {00}. The upstream path from the perspective of the switch contains the switches 1, 2 and 4. The advertised space of this path is {00}. Next, the algorithm computes what is forwarded over the path, i.e. the input space. Switch 2 and 4 forward the complete advertised space, i.e. {00}. However, switch 1 only forwards events matching {0000,0011}. This means, that events covered by 0010 or 0001 are filtered out at switch 1 and cannot reach switch 5, where the merge takes part. The input space of this path is therefore {0000,0011}. Next, the algorithm determines the fp-area for each egress port of f_m. For port 2, the area wrongfully covered by f_m is {0010,0001}. There is no overlap with the input space and therefore no false positives coming over this path will be forwarded. For egress port 3 the wrongfully covered subspace additionally contains {0011}, since f_2 does not forward over port 3. The overlap between input and wrongfully covered space, i.e. the fp-space for this port is therefore {0011}. The number of affected links amounts to 1. This means that every event sent by the publisher and covered by {0011} will cause exactly 1 network false positive. Since the fp-space {0011} covers on quarter of the total advertised space {00} we estimate, that 25% of the published events will cause false positives. Assuming that the publisher sent 1000 during the current window, the cost on that path is calculated as $1000 * 25\% * 1 = 250$.

The input space of the second path, starting from the lower publisher and containing the switches 3 and 4, is {00}. The fp-space on port 2 equals {0001,0010} which covers 25% of the advertised space {0}. The worst case number of affected links when forwarding over port 2 amounts to 3. Assuming the publisher has published 200 events in the current window, the costs for port 2 for this path amount to $200 * 25\% * 3 = 150$. For egress port 3, the fp-space is 0010,0001,0011 which covers 37.5% of the advertised space. This results in costs of $200 * 37.5\% * 1 = 75$. Ergo, the total cost on this path amounts to $150 + 75 = 225$. The total cost of the merge point is calculated as the sum of the cost over each path and equals 475 for this particular merge point.

5.4.3 Event History Based Cost Function

This section introduces the second cost function as described in algorithm listing 6. Instead of assuming that each publisher publishes events with a uniform distribution, this approach collects a history of actual events. The event history allows the optimizer to determine the actual event distribution. The consideration of the history is based on a sliding window approach, i.e. only the events published within the current window are considered in the history. In order to build the event history, each published event is also sent to the controller (or some separate event history component). This of course introduces a lot of traffic to the controller / the event history component. It may not be feasible to collect every single event in the history. Therefore, a configurable *sampling factor*, a value between 0 and 1, is introduced. The sampling factor equates to the probability of an event being sent to the collection service. The optimizer estimates the event distribution based on the event history samples and the sampling factor. The tradeoff between performance and the accuracy of the estimated distribution when choosing the sampling factor is discussed later in the evaluation section.

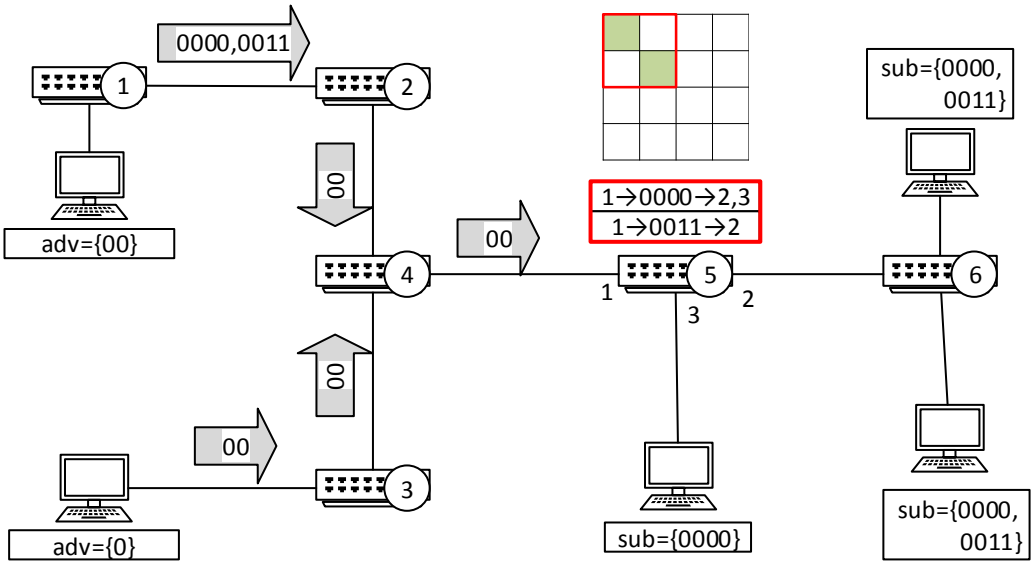


Figure 5.13: Computing the cost of a merge point

Other than that, the general structure of this cost function is similar to the traffic based function. The cost function again follows the path based approach and computes the cost on each path as the product of total number of events, the ratio of how many of those will cause network false positives and how many network false positives a single event can cause.

In order to estimate the total number of events, published by the publisher of some path, the algorithm just counts the events in the event history of said publisher (adjusted for the sampling factor if necessary). It no longer needs to rely on OpenFlow statistics.

In order to compute the ratio of events that will cause false positives, the traffic based approach compared the size of the fp-space with the size of advertised space. Assuming a uniform event distribution and a significant number of events, this will approximate the real false positive ratio well. This approach doesn't need to make assumptions about the distribution since it knows the actual event history. To determine the false positive ratio, the algorithm can simply count the number of events that are covered by the fp-space and divide that by the total number of published events.

The calculation of the number of affected links, over which a false positive event is forwarded, remains the same.

Let's now look again at the same example scenario as previously in the section introducing the traffic based cost function, depicted in figure 5.13. The event based cost function follows the same path based approach, starting with the upper path. For egress port 2, the fp-space is empty and the associated cost is thereby 0. For egress port 3, the fp-space is again 0011. Now,

Algorithm 6 Event History Based Cost Function

procedure CostFunction(mp) ▷ mp is a merge point
  $f_m \leftarrow$ flow at merge point
  $dz_m \leftarrow$ dz-expression at merge point
  $switch_m \leftarrow$ Switch to which f_m should be deployed
  $P \leftarrow$ Set of all publishers
  $cost \leftarrow 0$
  for all $publisher \in P$ **do**
    $path \leftarrow$ Path from $publisher$ to $switch_m$
    if $path$ enters $switch_m$ through $f_m.ingressPort$ **then**
      $cost \leftarrow cost +$ CostOnPath($path$)
    end if
  end for
end procedure

procedure CostOnPath($path$) ▷ compute cost on a particular path
  $Flows_m \leftarrow$ flows that are merged by m
  $affectedLinks_{port} \leftarrow$ number of links reachable in the subnet connected to $port$
  $inputSpace \leftarrow \bigcap_{switch \in path} \left(\bigcup_{flow \in \{f \in switch.flows \mid f.ingressPort \in path \wedge \exists p \in f.egressPorts : p \in path\}} flow.dz \right)$
  $publisher_{path} \leftarrow$ publisher associated with $path$
  $publishedEvents \leftarrow$ events published by $publisher_{path}$ in current window
  $receivedEvents \leftarrow \{e \in publishedEvents \mid inputSpace$ contains $e \wedge dz_m$ contains $e\}$
  $traffic \leftarrow |receivedEvents|$
  $costOnPath \leftarrow 0$
  for all $port \in f_m.egressPorts$ **do**
    $Flows_{port} \leftarrow \{f \in Flows_m \mid port \in f.egressPorts\}$
    $receivedEvents_{fp} \leftarrow \{e \in receivedEvents \mid \nexists f \in Flows_{port} : f$ contains $e\}$
    $fpRatio \leftarrow \frac{|receivedEvents_{fp}|}{traffic}$
    $costOnPath \leftarrow costOnPath + (traffic * fpRatio * affectedLinks_{port})$
  end for
  return $costOnPath$
end procedure

instead of dividing the size of the fp-space by the advertised space as the traffic based cost function does, the history based function consults the event history. It matches every event published event against the advertised space and the fp-space. If for example the advertised space covers 500 events and the fp-area (which is a subspace of the advertised space) covers 100 of those events, the algorithm computes the fp-ratio as $frac100500 = 20\%$. The cost for this port and path are thereby $500 * 20\% * 1 = 100$. The cost of the other path is computed equivalently.

5.5 Flow Limit Compliant Flow Deployment

Previous sections discussed how to deal with situations in which the number of flows on a switch exceed the allowed flow limit by selectively merging them. However, the flow limit constraints also need to be considered when changing an existing flow set without necessarily increasing the total number of flows.

The problem can be defined as follows: On a switch, we want to move from the current set of flow entries $F_{current}$ to a new set of flows F_{new}. Set F_{new} may contain a combination of new flows and flows that were already present in set $F_{current}$. Neither of the flow sets violates the flow limit constraint of the switch. In order to migrate from $F_{current}$ to F_{new} the system needs to delete some flows from $F_{current}$ (F_{delete}) and add some new flows (F_{add}), i.e. $F_{new} = (F_{current} - F_{delete}) \cup F_{add}$.

During the migration from $F_{current}$ to F_{new}:

- The flow limit constraint should not be violated
- No false negatives should be introduced
- False positives should be kept at a minimum

This is trivial to achieve when the sum of the number of existing flows and the number of new flows is lesser or equal than the flow limit ($|F_{current}| + |F_{add}| \leq flow\ limit$). In this case, the system can just deploy all flows from F_{add} first and afterwards delete the flow entries that need to be deleted.

However, this work focuses on the flow changes induced by the local enforcement and especially the global optimization algorithms. In these cases, often both $F_{current}$ and F_{new} contain as many flows as the flow limit allows. The proposed algorithm is tailored to deploy flow changes induced by the global and local algorithms introduced in this work and will not necessarily be applicable for the migration between two arbitrary flow sets.

A first naive approach would be to first delete all flows of F_{delete} and afterwards add the flows of F_{add}. While this proposal honors the flow limit, it also opens the system to the possibility of false negatives. For example, the system wants to merge two flows f_1, f_2 with dz-expressions 0000 and 0011 into a new flow f_m with dz-expression 00. It is $F_{delete} = \{f_1, f_2\}, F_{add} = \{f_m\}$. If we first remove f_1 and f_2 (or just one of them) before deploying f_m, events that arrive at

the controller after the removal and before the deployment, that would have been matched by f_1 or f_2, will be dropped instead.

The dilemma is solved by keeping space for one additional flow per switch, i.e. setting the flow limit constraint on each switch by one lower than it is in reality. In case of the changes induced by the local enforcement algorithms the problem is easily solvable now. The pays out as follows: $|F_{current}| = flow\ limit$ and one new flow f should be added to the switch. The local flow enforcement algorithm picks a merge point m, that merges two or three flows and provides the merge result f_m. f may or may not be involved in this merge point. The deployment algorithm first deploys f_m into the empty extra spot. Then it removes the flows that have been merged into f_m. Finally, if f did not participate in the merge, f is deployed.

In case of changes induced by a global optimization algorithm the situation is more complex, but the same logic can be applied. A naive approach would be to deploy a wildcard flow that forwards every event over every port but the one it was received over into the empty flow entry spot. Then remove all flows of F_{delete} and deploy all flows of F_{add}. Finally, remove the wildcard flow again. Although this approach cannot cause any false negatives, it would drastically increase the false positive rate during the migration phase. An important insight is that changes induced by the optimizer either replace multiple flows through the resulting flow of merging, or expand a formerly merged flow, i.e. replace a merged flow by (some of) the parts it was aggregated from. This motivates algorithm 7:

In phase one deploy all the merges one by one. For that purpose first install some flow f_m resulting from a merge, then remove all flows fully covered by f_m (the flows that have been merged). This operation temporally requires space for one flow entry more than the current number of flows on the switch. But after the completion of the operation, the number of flows on the switch will be lower than before.

In phase two all unmerges are performed one by one. To perform an unmerge, first deploy all formerly merged flows f_1,...,f_k, then remove the corresponding f_m. This operation increases the number of flows on the switch and temporarily needs space for one more flow than the present in the resulting flow set.

5.6 Self-Evaluation Component

This section introduces a component that enables the pub/sub system to estimate the current network false positives rate during runtime. The system can use this information to autonomously react if the false positive rate changes significantly. It can for example trigger the global optimizer or adjust the sampling factor of the event history collection. The proposed method works based on the collected event history as described in section 5.4.3. If the history based algorithm is used, the history is available anyhow.

The heart of the component is a lightweight software based network emulation. It supports forwarding of events based on the same flow entries deployed in the real network. The emulated switches perform simple prefix tree based matching in software. The emulated network mirrors

Algorithm 7 Consistent Deployment of Flow Change Sets

```
1:  procedure DEPLOY(F_add, F_delete)
2:      DEPLOYMERGES(a, b)
3:      DEPLOYUNMERGES(a, b)
4:  end procedure
5:
6:  procedure DEPLOYMERGES(F_add, F_delete)
7:      for all flow_add ∈ F_add do
8:          covered ← {flow ∈ F_delete | f_add fully covers flow}
9:          if |covered| > 0 then
10:             deploy flow_add to switch
11:             remove all flow ∈ covered from switch
12:             F_add ← F_add \ {f_add}
13:             F_delete ← F_delete \ covered
14:         end if
15:     end for
16: end procedure
17:
18: procedure DEPLOYUNMERGES(F_add, F_delete)
19:     for all flow_delete ∈ F_delete do
20:         covered ← {flow ∈ F_add | f_delete fully covers flow}
21:         deploy all flow ∈ covered to switch
22:         remove flow_delete from switch
23:         F_add ← F_add \ covered
24:         F_delete ← F_delete \ {f_delete}
25:     end for
26: end procedure
```

the actual network. As depicted in figure 5.14, whenever the controller deploys a flow into the real network, it also deploys the same flow in the emulated network.

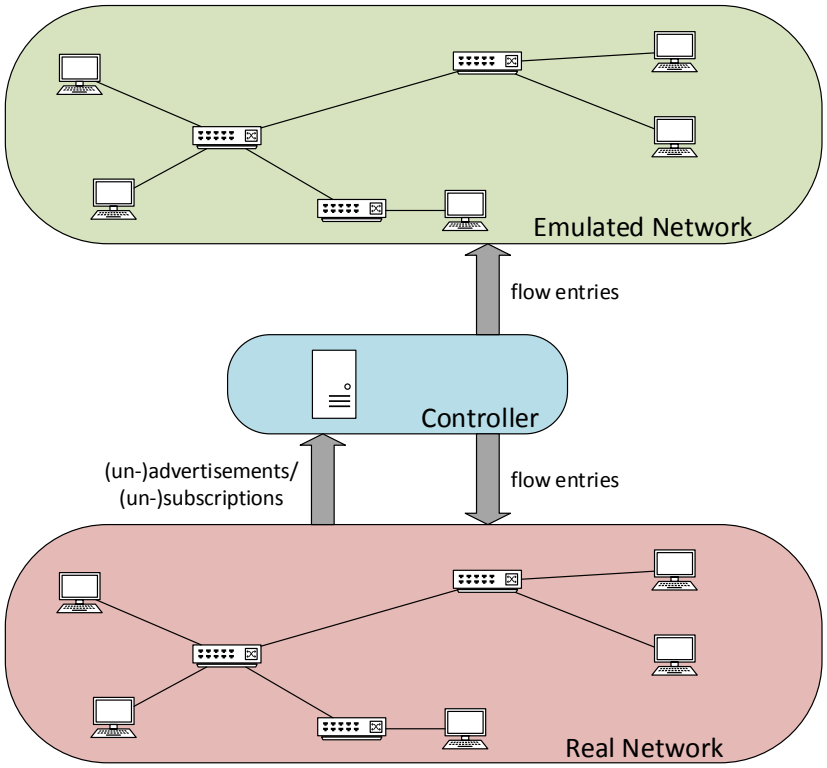


Figure 5.14: Flows are deployed in a mirrored emulated network for self-evaluation

In order to determine network false positives, the system replays the event history in the emulated network. Since it has full control over the network, it is easy to trace the route of each event through the network. The system can check if an event, sent over some link, is a false positive by checking if the link is part of any route from the originator of the event to any of the subscribers that are interested in that event. With this technique, the exact number of false positives based on the event history can be determined. Of course, the overall accuracy of the result depends on the completeness of the event history, i.e. the used sampling factor. This is equivalent to the tradeoff between sampling factor and accuracy of the optimizer described in section 5.4.3.

The network emulation can easily be run in a service separate from the controller on another machine. The overhead in the controller is minimal. It just needs to forward the flow changes to the network emulation component. Though, the usage of this component is optional.

Chapter 6

Evaluation and Analysis

This chapter evaluates and analyses the properties of the proposed algorithms such as their efficiency in minimizing network false positives and their execution time under a realistic, large scale work load. Furthermore, the effect of the adjustable parameters, especially the sampling factor is investigated. For this purpose, multiple experiments are conducted, using a real SDN testbed as well as an emulated network infrastructure.

6.1 Evaluation Setup

The pub/sub system uses the open source Floodlight controller [37] in combination with OpenFlow 1.3 to control the flow entries as switches.

Some of the experiments were conducted on a real SDN testbed using a Whitebox EdgeCore switch. The topology of the testbed consists of a binary tree of depth three. Each switch of the lowest layer is connected to two hosts. Hence, the network consists of 7 switches and 8 hosts. The hosts are executed on commodity hardware. The SDN controller is executed on a machine with a 3.1 GHz processor and 40 cores.

In order to perform experiments with bigger topologies, the popular network emulation tool Mininet [59] is used. Mininet allows for lightweight emulation of large scale networks on commodity hardware. The virtual switches provided by Mininet are OpenFlow enabled. Therefore, the pub/sub system can deploy its flows in a same way that it deploys the flows to a real system. The system isn't even aware if it runs on Mininet or a real testbed. In order to precisely measure the network false positives, the self-evaluation component introduced in section 5.6 is utilized. These experiments use two machines. The controller is hosted on a machine with 4 3.2 GHz cores and 8 GB RAM. Mininet runs on a separate machine using 2 3.5 GHz cores and also 8 GB RAM.

For the large scale experiments, two different topology types were used. Firstly, a regular tree structure with a depth of 4 and a fanout of 5, i.e. starting from the root switch, each switch has five child-switches. This means, that there are 156 switches in total. Each of the 125 switches of the lowest level is connected to seven hosts, i.e. the total number of hosts is 875. The second topology is a random tree of varying size generated with algorithm 8. Instead of connecting all the hosts to the lowest layer in the tree, they are connected to random switches

all over the tree. The only requirement is, that no switch is linked to more than 24 entities, i.e. 24-port switches. The largest topology used contains 300 switches and 4202 hosts.

The experiments are based on a real data set of daily stock closing prices by Yahoo! Finance [60] as well as synthetically generated data sets. The synthetic data sets are generated over a 6-dimensional event space using values from range $[0, 4096]$ for each dimension. In order to generate the data sets, both a uniform distribution and a zipfian distribution with 5 hotspots are used. In particular, the following four configurations are used in the experiments:

- Uniform - synthetic data set where advertisements, subscriptions and events are uniformly distributed over the event space
- Zipfian - synthetic data set where advertisements, subscriptions and events follow the same zipfian distribution
- Mixed - synthetic data set where advertisements and subscriptions are uniformly distributed while events follow a zipfian distribution
- Real Data - uniformly distributed subscriptions with real events stemming from Yahoo! Finance stock data

Algorithm 8 Generating a random tree

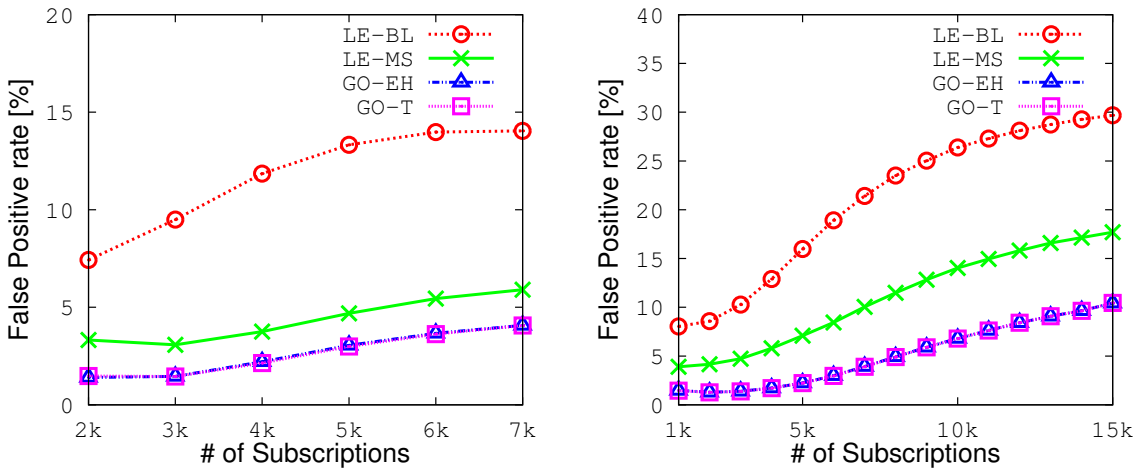
```
1:  procedure CreateRandomTree(#Switches, #hosts)          ▷ mp is a merge point
2:      Switches ← create #Switches switches
3:      root ← pick s ∈ Switches uniform random
4:      Connected ← {root}
5:      Unconnected ← Switches \ {root}
6:      while Unconnected ≠ ∅ do
7:          switch ← pick s ∈ Unconnected uniform random
8:          partner ← pick s ∈ Connected uniform random s.t. s has at least one unused port
9:          Connect(switch, partner)
10:         Unconnected ← Unconnected \ switch
11:         Connected ← Connected ⋃ switch
12:     end while
13:     for 1 → #hosts do
14:         host ← create host
15:         attachmentPoint ← pick s ∈ S uniform random s.t. s has at least one unused port
16:         Connect(host, attachmentPoint)
17:     end for
18: end procedure
```

6.2 Network False Positives

The main design criterion of the proposed algorithm is to minimize the increase in network false positives introduced through merging. This section answers the question, to which degree the

proposed algorithms achieve this goal under various conditions. It especially investigates the effect of the number of subscribers and the merge factor on the network false positive rate. The baseline for measuring network false positives is the original system without merging. This is equivalent to setting the flow limit of infinity. We are only interested in how many new false positives are introduced due to merging and therefore assume that there are no false positives in the case of infinite flow limits. Whenever an event is forwarded over a link although it shouldn't, it counts as a network false positive. The network false positive rate is the number of network false positives divided by the total number of forwardings of events over any link.

This first set of experiments was conducted with a fixed set of advertisements and increasing number of subscriptions. The subscriptions are distributed uniformly random over the hosts. The flow limit is set to 400 flows per switch in case of the regular tree topology and 600 flows per switch for the random tree topology. The event history based approach uses a sampling factor of 1, i.e. every packet is collected. Figure 6.1 depicts the relationship between number of subscriptions and false positive rate for both topologies using the uniform distribution model. The false positive ratio increases with increasing number of subscription in all cases. Generally, more subscriptions lead to a higher number of flows on each switch. Hence, more merges are necessary and the overall expressiveness of flows decreases. The local base line approach is clearly outperformed by the improved local minimum space approach, showing that minimizing space and port expansion alone comes with a great benefit. Nevertheless, the global approaches offer a further improvement, although not by up to 8 percentage points. It's notable, that both global approaches achieve almost identical results. This is due to the fact that the traffic based approach makes the assumption that publishers publish events uniformly over their advertised space. In this specific case, this assumption is perfectly true; thereby the event based approach cannot gain any additional advantage by knowing the actual event distribution.

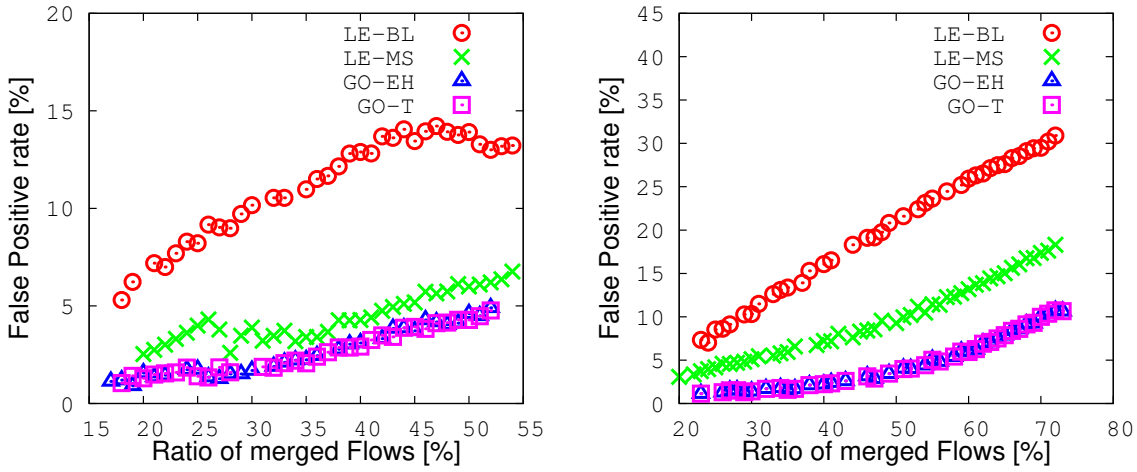


(a) Regular Tree

(b) Random Tree

Figure 6.1: Number of subscribers to false positive rate with uniform distribution

Figure 6.2 shows the result of the same experiments from a different angle. Instead of depicting the number of subscriptions on the x-axis, it shows by how much the number of flows has been reduced. This is depicted in terms of the merge ratio, defined as the average flow reduction on each switch. For example, if the system would deploy 2000 flows on a switch in absence of flow limits, but reduces the number of deployed 500 flows, the merge ratio would be $\frac{2000-500}{2000} = 75\%$. The interpretation is similar to the one for figure 6.1. Increasing number of merges lead to more coarse grained flows and therefore more network false positives. However, it's remarkable that even with a flow reduction as high as 70%, the false positive rate is only at 10% in case of the global algorithms.



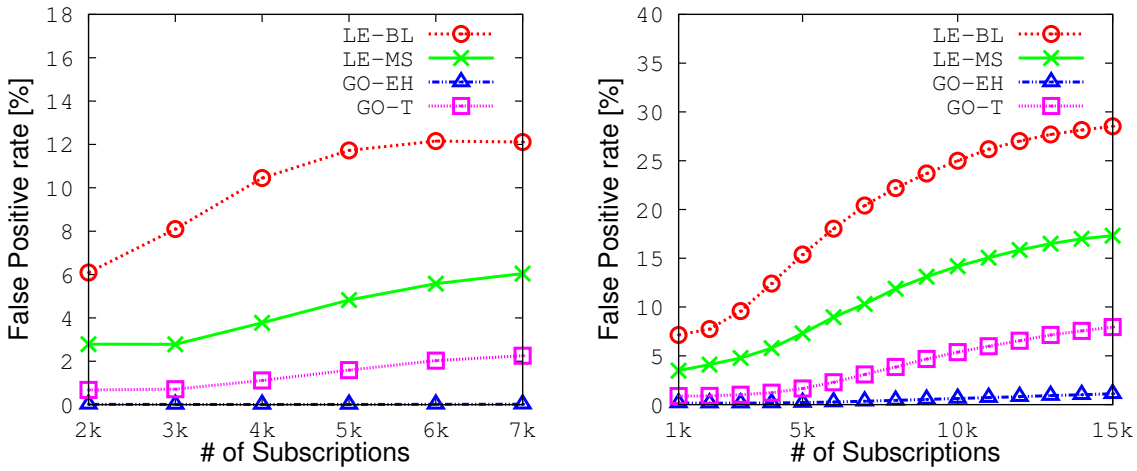
(a) Regular Tree

(b) Random Tree

Figure 6.2: Merge ratio to false positive rate with uniform distribution

The next set of experiments is similar to first set, with the difference that this time the mixed distribution model is used in order to generate advertisements, subscriptions and events. Similar to the figures seen before, figure 6.3 shows the relationship between subscriptions and false positives, while figure 6.4 shows the relationship between merge ratio and false positives. The most interesting characteristic of the mixed distribution model is that there is a big mismatch between the distribution of the subscriptions and the distribution of events. While subscriptions and advertisements cover the event space mostly uniformly, events are concentrated around a few hotspots. This makes merging a lot more interesting because the potential for very good and for very bad merges exists. In fact, the event history based global optimizer is able to enforce the flow limit while causing almost no false positives at all. Due to its knowledge of the actual event distribution it can primarily merge flows that forward events in low traffic regions of the event space. Thereby, it can outperform the traffic based optimizer. The traffic based optimizer also tries to differentiate high and low traffic subspaces. However, it only sees the total traffic produced by a publisher. So when a publisher advertises only in high or low traffic regions, the estimations should be quite good. When the advertisements of a publisher on the other hand span both high and low traffic areas it loses some precision because it averages the traffic over the whole advertised area. Nevertheless, the traffic based

global optimizer is still significantly better than the local approaches. The difference between the best approach, i.e. the event based global optimizer and the baseline approach is up to 30 percentage points.



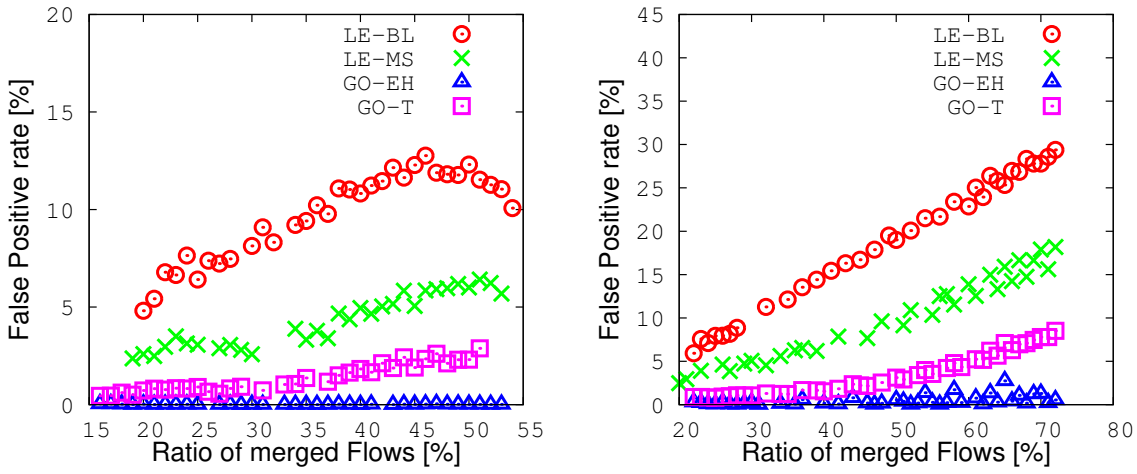
(a) Regular Tree

(b) Random Tree

Figure 6.3: Number of subscribers to false positive rate with mixed distribution

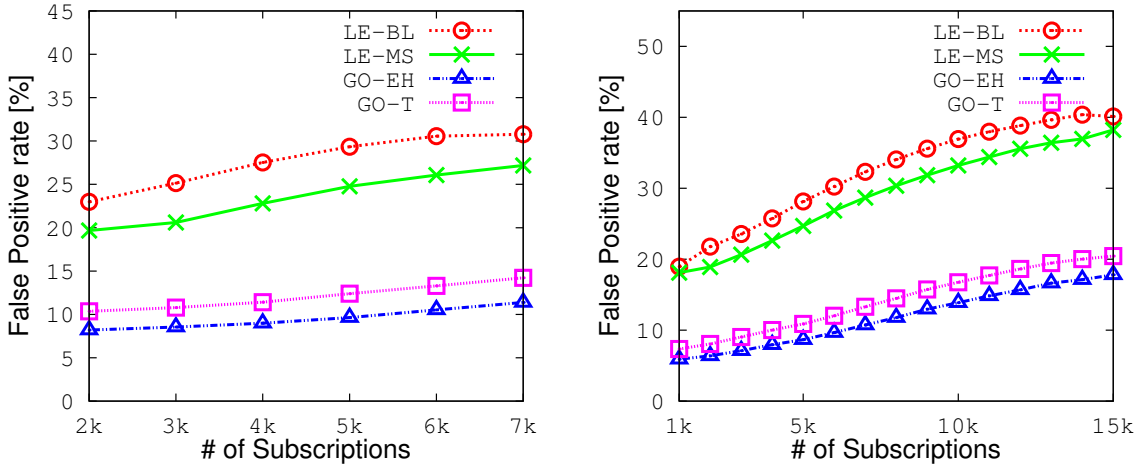
The next set of experiments again use the same conditions apart from the workload distribution. This time, the pure zipfian distribution model is used, i.e. advertisements, subscriptions and events follow the same zipfian distribution. Figure 6.5 plots the number of subscribers against the false positive rate. Again, the event based optimizer provides the lowest false positive rate while the baseline approach produces the worst one. All algorithms have higher false positive rates than in the previous distributions, because in this model, the advertisements, subscriptions and events are all crammed into small parts of the event space. This makes it hard to make good merge decisions. Within the small subspaces where most advertisements and events are situated, the events are relatively uniformly distributed over the advertisements. Therefore, the event history based optimizer also cannot offer much improvement over the traffic based approach. Figure 6.6 shows the relationship between the ratio of merged flows and false positive rate. The graph is in accordance to the results of figure 6.5 and the interpretation follows the aforementioned arguments.

Figure 6.7 depicts the evaluation of the complete system in a dynamic setting. Chapter 5 proposes to use a local enforcement algorithm in combination with periodically executing the global optimizer in order to deal with constantly changing subscribers and publishers. To evaluate this scenario, subscriptions are continuously added to the system at a fixed rate while enforcing flow limits with the local minimum space enforcer. Every 3000 subscriptions the global optimizer is executed. The figures illustrate how the false positive rate pans out. Figure 6.7a shows the traffic based optimizer in combination with the minimum space local enforcer. Figure 6.7b shows the history based optimizer again with the minimum space local enforcer. Both figures also plot the false positive rate when using only the MS local enforcer. For these experiments, a random tree topology with 200 switches and the mixed distribution model is



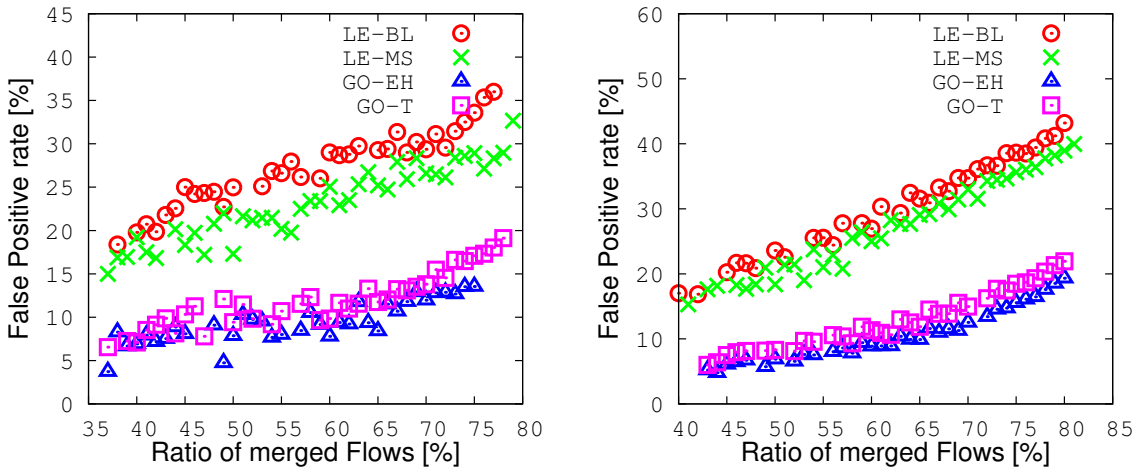
(a) Regular Tree (b) Random Tree

Figure 6.4: Merge ratio to false positive rate with mixed distribution



(a) Regular Tree (b) Random Tree

Figure 6.5: Number of subscribers to false positive rate with pure zipfian distribution



(a) Regular Tree

(b) Random Tree

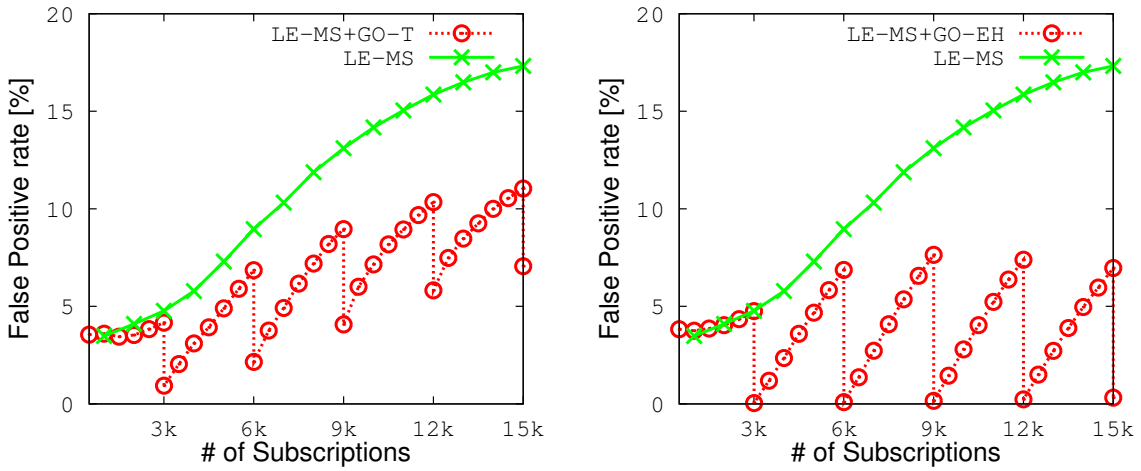
Figure 6.6: Merge ratio to false positive rate with pure zipfian distribution

used. In both cases, the false positive rate increases while new subscriptions are introduced and merged with the local algorithm. Whenever the global optimizer is applied, the false positive rate drastically decreases only to slowly increase again. However, the false positive rate never reaches the level of the local only approach again after the first optimization. Even executing the global optimizer as seldom as every 3000 subscriptions yields a significant improvement over using only the local enforcement algorithm - no matter which cost function is used for the global optimizer. However, the history based approach still outperforms the traffic based approach, especially in the presence of many subscriptions. The history based approach is able to reduce the false positive rate to almost zero every time, while the false positive rate after applying the traffic based optimizer increases with increasing subscriptions. This is the same behavior we saw in the earlier experiments pictured in figure 6.3 which were conducted with similar conditions.

6.3 Execution Time

This section is dedicated to the influence of various factors on the execution time of the algorithms, in particular of the global optimization algorithms.

Figure 6.8 plots the execution time of both variations of the global optimization algorithm against the number of subscribers. The experiment is conducted in the regular tree topology. With increasing subscriptions, the execution time for both algorithm increases roughly at the same rate. This comes as no surprise, since both algorithms need to calculate the cost for every merge point. More subscriptions result in a higher number of flows on each switch and thereby more possible merge points that need to be considered. The difference between the two variations is how they compute the cost of each merge point. The traffic based approach



(a) Traffic based GO + Local Enforcer

(b) Event history based GO + Local Enforcer

Figure 6.7: Local Enforcer with and without periodic global optimization

is faster in this experiment than the event history based one. However, the execution time of the history based approach also depends on how many events are part of the history, i.e. on the sampling rate and the rate at which events are published. This is discussed in more detail in the sampling section. For this experiment, a sampling factor of 1 is used. Note that the history based approach also comes with the overhead of collecting the event history, which is not depicted here as it is done separately from running the optimizer.

Figure 6.9 shows the relationship between the size of the topology, especially the number of switches and execution time. To achieve this, the optimization algorithm is executed in increasingly large random tree topologies with up to 300 switches. While the number of switches increases, the number of publishers, subscriptions and the flow limits remains the same. As expected, the execution time increases with the topology size, since a) the algorithm has to compute the best flow set for every switch and b) when calculating the flow set, the algorithms have to look at various up and down stream switches whose number also increases with topology size. However, after the network reaches a certain size (around 100 switches), the increase of execution time significantly decreases. This is due to the fact in larger networks the constant number of subscriptions is more spread out which leads to a lower average number of merge points per switch. The traffic based approach again outperforms the history based one in these experiments.

Lastly, the average execution times of the local enforcement algorithms have been evaluated. Since they are only executed locally for a single switch, their execution time is independent from the number of switches in the network. In average, the base line approach takes around 0.2 milliseconds. The minimum space approach has a higher execution time since it has to evaluate the possible merge points with a simple cost function. Its average execution time amounts to 0.5 milliseconds. Note that a single new subscription can trigger the local enforcement algorithm on many switches. However, the execution of the enforcer can be done

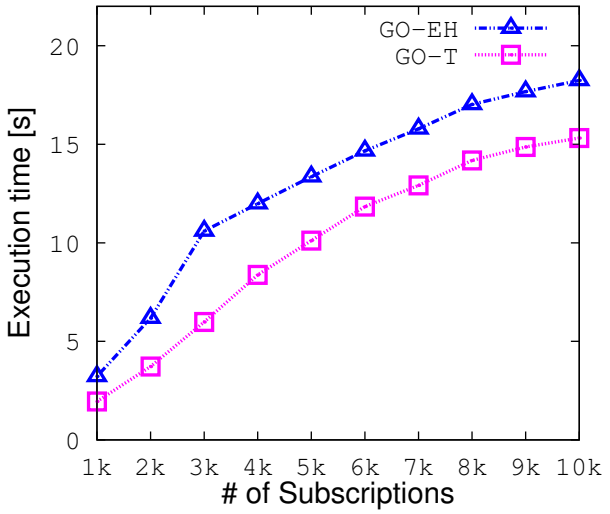


Figure 6.8: Influence of subscription quantity on global optimization execution time

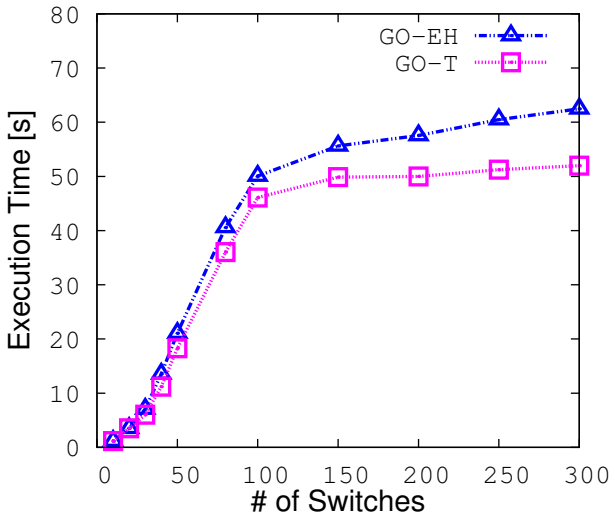


Figure 6.9: Influence of the topology size on global optimization execution time

in parallel for each switch.

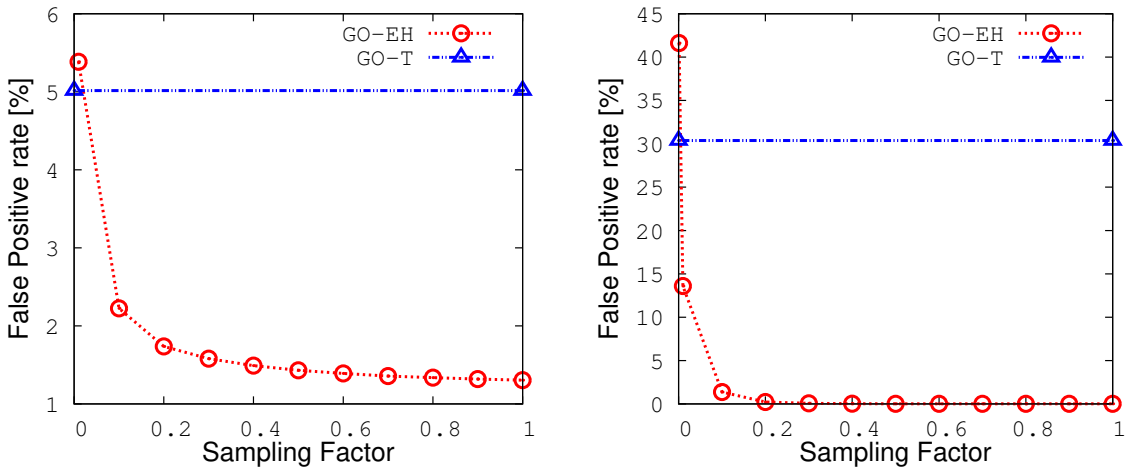
6.4 Sampling

This section investigates the effect of the sampling factor, i.e. which percentage of events that are considered in the event history. The system can freely choose and adjust this parameter. When choosing the sampling factor, there is a general tradeoff between accuracy and overhead. The higher the sampling factors should be better to approximate the event distribution, but higher sampling factor also means that more events have to be collected and processed. This section focuses on the overhead when processing events during the execution of the global optimizer.

Figure 6.10a depicts the network false positive rate when using the event history based global optimizer at different sampling factors ranging from 1% to 100%. The experiment was conducted in a random tree topology with 200 switches and more than 4000 hosts using the mixed distribution model. As expected, the false positive rate continuously decreases with increasing sampling factor. The gradient of the decrease also decreases quickly and continuously. Increasing the the sampling factor from 1% to 10% reduces the false positive rate by ca. 3 percentage points. Increasing the rate from 10% to 20% only reduces the false positive rate by about 0.5 percentage points. Setting the sampling factor higher than 50% yields negligible improvements. The difference in false positives between sampling at 50% and 100% is less than 0.15 percentage points. Furthermore, the figure shows the false positive rate of the traffic based approach. This approach doesn't use the event history and is therefore unaffected by the sampling factor. The history based approach outperforms the traffic based one when using a sampling factor of approximately 2.5% or higher. All in all, this shows that the history based approach can make good estimations about the event distribution with a sampling factor as low as 20%-30% and considerably outperforms the traffic based approach.

Figure 6.10b shows a similar experiment using the real stock data and uniformly distributed subscriptions. The events in this data set are even more concentrated around few hotspots than in case of the zipfian distribution. Therefore, the event based global optimizer can even better exploit low traffic areas when merging. This is reflected in the figure. With sampling factor 0.2 or better, the false positive ratio is almost zero, proofing the effectiveness of the history based approach in a realistic scenario. This experiment also includes sampling at the rate of 0.1%. However, sampling at this rate leads to a high false positive rate, since the algorithm can easily draw wrong conclusions about the event distribution with such a small sample.

Figure 6.11 looks at the relationship between sampling factor and network false positives from a different angle. Instead of looking at the false positive rate in the whole network, it only considers the events that reach the subscribers. Furthermore, this experiment is executed on the real SDN testbed with the mixed distribution model. The figure shows the average false positive rate at the subscribers for different sampling factors ranging from 0.1% to 100%. The false positive rate at subscribers follows the same trend as the network false positive



(a) Mixed distribution model

(b) Stock data set

Figure 6.10: Influence of the sampling factor on network false positives

rate before. With increasing sampling factor the false positive rate quickly decreases in the beginning and then only improves slightly.

Finally, figure 6.12 depicts the relationship between sampling factor and the execution time of the event history based global optimization algorithm. Unsurprisingly, the execution time increases almost proportionally with increasing sampling factor. This is due to the fact that the algorithm has to match every event in the history many times against different dz-expressions in order to check if it would cause false positives or not. While keeping previous figures in mind, this clearly demonstrates the tradeoff between execution time and bandwidth efficiency when choosing the sampling factor. In this experiment sampling factors ranging from 1% to 100% and the stock data event distribution where used.

6.5 Adapting to Changing Event Distributions

This section evaluates, how the event history based optimizer adapts to changing event distributions. In this model, the events are generated using a zipfian distribution with 5 hotspots. In this particular experiment, the hotspots are randomly shifted by up to 10% every 10,000 events. The advertisements and subscriptions remain the same during the whole experiment. The event based global optimizer is run periodically. Figure 6.13 depicts the course of the experiment. The false positive rate initially amounts to ca. 0.5%. Due to the shifting of the hotspot locations, earlier made merge choices are not optimal anymore and the false positive rate increases. The farther the hotspots move away from their original position, the higher the false positive rate climbs. Till, at some point, the optimizer is executed again and the false positive rate drops back to 0.5%. Then, the process begins anew. This experiment was conducted in a random tree topology containing 200 switches.

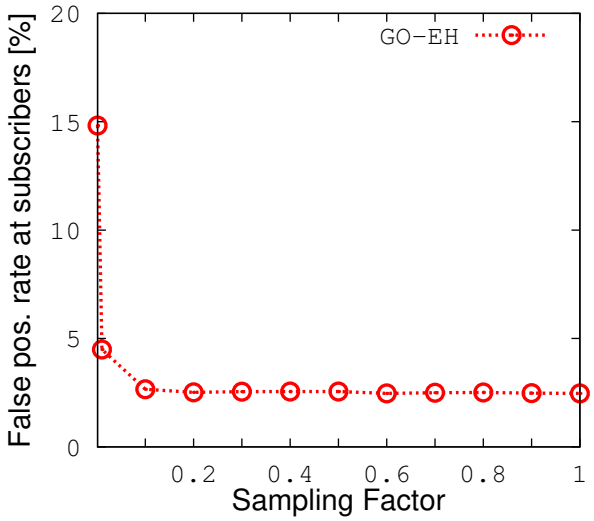


Figure 6.11: False positive rate at subscribers in real testbed

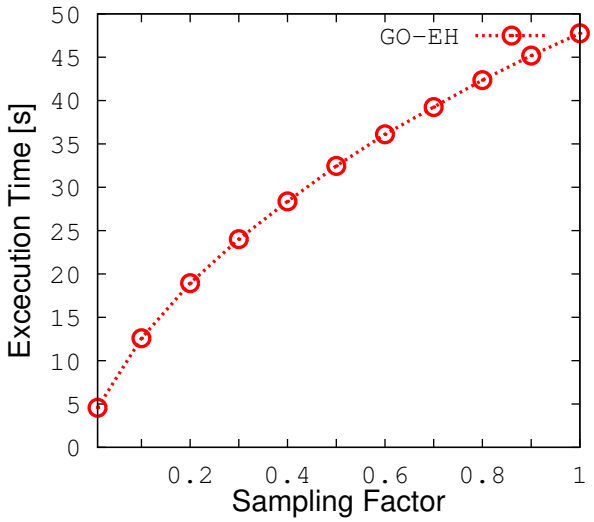


Figure 6.12: Influence of the sampling factor on execution time

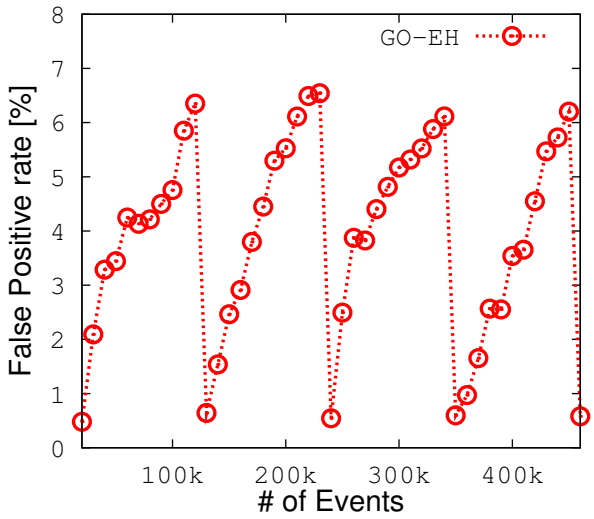


Figure 6.13: Adapting to a changing mixed distribution

6.6 Flow Limit Compliant Deployment

Section 5.5 introduced a mechanism to deploy the flow changes caused by the global optimizer (and to a lesser extend the local enforcement algorithms) without introducing false negatives. This has been tested in on the real test bed as well as the emulated Mininet environment. Therefore, events were published at a constant rate while the flow changes were deployed. None of the experiments introduced false negatives, proving the correctness of the proposed method for flow limit conform deployment of flows.

Chapter 7

Conclusion and Future Work

In order to address the limited TCAM space in current OpenFlow-enabled switches, this work offers mechanisms to constrain to number of flow entries to a specific limit in the context of SDN-based pub/sub systems. By selectively merging flow entries, this can be achieved without introducing any false negatives in the system. The proposed algorithms have been integrated into PLEROMA, an existing content-based pub/sub system. By combining a continuously acting local enforcement algorithm and a periodically executed global optimization algorithm, the system is able to adhere to the flow limits while keeping the increase in network false positives at a minimum - even in highly dynamic systems. In fact, in many situation, the global optimizer is able to reduce the number of flows by more than 70% without increasing the false positive rate significantly. The effectiveness of the proposed algorithms has been demonstrated consistently under various workloads using synthetic data sets as well as data derived from the real world. The evaluations were conducted on multiple toplogies using a real SDN testbed as well as popular network emulation tools.

In total this thesis proposed two local algorithms that both enforce the flow limit at all times. In addition to that, a global optimization algorithm has been introduced that can tries to find the best possible configuration of flows that a) adheres to the flow limit and b) minimizes the increase in false positives. The Optimizer comes in two flavors. A traffic based approach that relies on statistics provided by OpenFlow and an event history based approach. The event history based approach was shown to be more bandwidth-efficient at the expense of additional overhead due to the collection of an event history. Furthermore, an algorithm for applying the modifications to the flow tables suggested by the optimizer without violating flow constraints has been introduced. Finally, this work also proposed a component that enables the system to autonomously determine its current false positive rate. However, there is still room for improvement.

While it was shown that the proposed algorithms can adapt to changing event distributions by periodically executing the global optimizer and thereby employing a sliding window approach, this is a purely reactive approach. In the meantime between a significant change of the event distribution and the next optimization round the false positives will be increased. Future work could try to build predictive statistical models of the event distribution based on the collected event histories and traffic data. This would allow the optimizer to proactively adapt the flow configuration before the event distribution changes and therefore keeping the false positive rate constantly low.

Furthermore, this thesis proposes two local enforcement algorithms, the base line approach as well as the minimum space heuristic which has shown to be a major improvement over the base line approach. However,the main focus of this work was the global optimization problem. The design of the local enforcement algorithms is quite constraint due to the hard time requirements. Nevertheless, future work could investigate more sophisticated algorithms for continuously enforcing flow limits on a local basis. These algorithms could for example also take the event distribution into account.

The evaluation sections showed demonstrated the tradeoff between overhead and accuracy when choosing the sampling factor. However, choosing a sampling factor is not integrated into the system. The current implementation uses the same statically set sampling factor for all publishers. Future work could look into dynamically adapting the sampling rate, individually for ever publisher, for example based on the amount of traffic they generate. Furthermore, the false positive rate gathered with the proposed self-evaluation component could come at play here. Another related design parameter that hasn't been discussed at all is the size of the sliding widow employed by the global optimizer.

Finally, the current solution assumes a single dissemination tree. However, PLEROMA normally uses multiple dissemination trees, for example in order to better spread out the traffic over the whole network. Future work could adapt the the current implementation to support multiple trees. Possible approaches are to either define and enforce flow limits individually for each tree (e.g. a switch that is part of multiple dissemination trees has a separate flow limit constraint for flows of each tree) or to define and enforce a global flow limit across all trees.

Bibliography

[1] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 332–383, 2001.

[3] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Springer Science & Business Media, 2006.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, no. 3, pp. 332–383, Aug. 2001.

[5] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh, "The padres publish/subscribe system." *Principles and Applications of Distributed Event-Based Systems*, vol. 164, p. 205, 2010.

[6] B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," *Proceedings of the I997 Australian UNLX Users Group (A UUG'I997)*, pp. 243–255, 1997.

[7] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content based routing with elvin4," in *Proceedings AUUG2k*, 2000.

[8] G. Mühl, "Large-scale content-based publish-subscribe systems," Ph.D. dissertation, Technische Universität, 2002.

[9] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel, "Pleroma: A sdn-based high performance publish/subscribe middleware," in *Proceedings of the 15th International Middleware Conference*. ACM, 2014, pp. 217–228.

[10] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Dürr, T. Kohler, and K. Rothermel, "High performance publish/subscribe middleware in software-defined networks," *IEEE/ACM Transactions on Networking*, 2016.

[11] B. Koldehofe, F. Dürr, M. A. Tariq, and K. 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*, ser. MW4NG '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2405178.2405181

[12] S. Bhowmik, M. A. Tariq, B. Koldehofe, A. Kutzleb, and K. 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*. ACM, 2015, pp. 92–103.

[13] K. Zhang and H.-A. Jacobsen, "Sdn-like: The next generation of pub/sub," *arXiv preprint arXiv:1308.0056*, 2013.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[15] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

[16] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in tcams," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008, pp. 111–115.

[17] S. Bhowmik, M. A. Tariq, L. Hegazy, and K. Rothermel, "Hybrid content-based routing using network and application layer filtering," in *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, 2016, pp. 221–231.

[18] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 13–24.

[19] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in openflow networks," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 177–188.

[20] S. Bhowmik, M. A. Tariq, J. Grunert, and K. Rothermel, "Bandwidth-efficient content-based routing on software-defined networks," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 137–144.

[21] M. Franklin and S. Zdonik, ""data in your face": push technology in perspective," in *ACM SIGMOD Record*, vol. 27, no. 2. ACM, 1998, pp. 516–519.

[22] J. Bates, J. Bacon, K. Moody, and M. Spiteri, "Using events for the scalable federation of heterogeneous components," in *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*. ACM, 1998, pp. 58–65.

[23] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, "A peer-to-peer approach to content-based publish/subscribe," in *Proceedings of the 2nd international workshop on Distributed event-based systems*. ACM, 2003, pp. 1–8.

[24] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. pearson education, 2005.

[25] G. Mühl and L. Fiege, "Supporting covering and merging in content-based publish/subscribe systems: Beyond name/value pairs," *IEEE Distributed Systems Online (DSOnline)*, vol. 2, no. 7, pp. 224–238, 2001.

[26] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba event service," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, 1997.

[27] M. Altınel and M. J. Franklin, "Efficient filtering of xml documents for selective dissemination of information," in *Proc. of the 26th Int'l Conference on Very Large Data Bases (VLDB), Cairo, Egypt*, 2000.

[28] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[29] M. A. Tariq, G. G. Koch, B. Koldehofe, I. Khan, and K. Rothermel, "Dynamic publish/subscribe to meet subscriber-defined delay and bandwidth constraints," in *European Conference on Parallel Processing*. Springer, 2010, pp. 458–470.

[30] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. 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*. ACM, 2012, pp. 213–224.

[31] A. K. Y. Cheung and H.-A. Jacobsen, "Load balancing content-based publish/subscribe systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 9, 2010.

[32] O. Papaemmanouil and U. Cetintemel, "Semcast: Semantic multicast for content-based data dissemination," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 242–253.

[33] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman, "Exploiting ip multicast in content-based publish-subscribe systems," in *IFIP/ACM International Conference on Distributed systems platforms*. Springer-Verlag New York, Inc., 2000, pp. 185–207.

[34] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (bgp-4)," Internet Requests for Comments, RFC Editor, RFC 4271, January 2006, http://www.rfc-editor.org/rfc/rfc4271.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4271.txt

[35] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[36] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[37] "Project floodlight - open source software for building software-defined networks," http://www.projectfloodlight.org/floodlight, accessed: 2017-04-19.

[38] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[39] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of sdn/openflow controllers," in *Proceedings of the 9th central & eastern european software engineering conference in russia*. ACM, 2013, p. 1.

[40] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[41] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using tcam," in *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*. IEEE, 2004, pp. 174–183.

[42] A. J. McAuley and P. Francis, "Fast routing table lookup using cams," in *INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE*. IEEE, 1993, pp. 1382–1391.

[43] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "Lipsin: line speed publish/subscribe inter-networking," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 195–206, 2009.

[44] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[45] IBM TJ Watson Research Center, "Gryphon: Publish/subscribe over public networks," http://www.research.ibm.com/distributedmessaging/gryphon.html, accessed: 2017-04-19.

[46] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM, 1999, pp. 53–61.

[47] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE, 1999, pp. 262–272.

[48] H. Parzyjegla, D. Graff, A. Schröter, J. Richling, and G. Mühl, "Design and implementation of the rebeca publish/subscribe middleware," in *From active data management to event-based systems and more*. Springer, 2010, pp. 124–140.

[49] G. Mühl, "Generic constraints for content-based publish/subscribe," in *International Conference on Cooperative Information Systems*. Springer, 2001, pp. 211–225.

[50] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 175–180.

[51] ——, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research*. ACM, 2016, p. 6.

[52] H. Zhu, H. Fan, X. Luo, and Y. Jin, "Intelligent timeout master: Dynamic timeout for sdn-based data centers," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 734–737.

[53] K. Kannan and S. Banerjee, "Flowmaster: Early eviction of dead flow on sdn switches," in *International Conference on Distributed Computing and Networking*. Springer, 2014, pp. 484–498.

[54] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 545–549.

[55] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "Scalable rule management for data centers." in *NSDI*, vol. 13, 2013, pp. 157–170.

[56] H. Samet, "Applications of spatial data structures," 1990.

[57] R. Ahuja, R. Illingworth, H. Kanakia, and B. Shah, "System and method for locating a route in a route table using hashing and compressed radix tree searching," Aug. 31 1999, uS Patent 5,946,679.

[58] J.-I. Aoe, K. Morimoto, and T. Sato, "An efficient implementation of trie structures," *Software: Practice and Experience*, vol. 22, no. 9, pp. 695–721, 1992.

[59] R. L. S. de Oliveira, A. A. Shinoda, C. M. Schweitzer, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*. IEEE, 2014, pp. 1–6.

[60] Middleware Systems Research Group, University of Toronto, "Yahoo! finance stock quote data set," http://www.msrg.org/datasets/acDataSet, accessed: 2017-04-19.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature