Addressing TCAM Limitations of Software-Defined Networks for Content-Based Routing

Sukanya Bhowmik, Muhammad Adnan Tariq, Alexander Balogh, Kurt Rothermel
University of Stuttgart, Germany
{firstname.lastname}@ipvs.uni-stuttgart.de

ABSTRACT

In recent years, content-based publish/subscribe middleware has harnessed the power of Software-Defined Networking (SDN) to leverage performance gains in terms of throughput rates, end-to-end latency, etc. To this end, content filters are directly installed on the Ternary Content Addressable Memory (TCAM) of switches. Such a middleware assumes unlimited TCAM space to deploy content filters. However, in reality, TCAM is a scarce resource and the number of flow table entries available for publish/subscribe traffic is severely limited. While such a limitation poses severe problems for the deployment of publish/subscribe middleware in practice, it is yet to be addressed in literature.

So, in this paper, we design a filter aggregation algorithm that merges content filters on individual switches to respect TCAM constraints while ensuring minimal increase in unnecessary network traffic. Our algorithm uses the knowledge of advertisements, subscriptions, and a global view of the network state to perform bandwidth-efficient aggregation decisions on necessary switches. We provide different flavors of this algorithm with varying degrees of accuracy and complexity and thoroughly evaluate their performances under realistic workload. Our evaluation results show that our designed aggregation algorithm successfully meets TCAM constraints on switches while also reducing unnecessary traffic introduced in the network due to aggregation as compared to a baseline approach by up to 99.9%.

CCS CONCEPTS

•Networks → Network services; •Computer systems organization → Distributed architectures;

KEYWORDS

SDN, Publish/Subscribe, Content-based Routing, TCAM Limitations, Filter Aggregation

ACM Reference format:
Sukanya Bhowmik, Muhammad Adnan Tariq, Alexander Balogh, Kurt Rothermel. 2017. Addressing TCAM Limitations of Software-Defined Networks for Content-Based Routing. In *Proceedings of Distributed and Event-Based Systems, Barcelona, Spain, June 19-23, 2017 (DEBS '17),* 12 pages.
DOI: http://dx.doi.org/10.1145/3093742.3093924

DEBS '17, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/3093742.3093924

1 INTRODUCTION

Content-based publish/subscribe (pub/sub) is a widely adopted communication paradigm designed to enable loosely coupled producers (publishers) and consumers (subscribers) of information to interact in a bandwidth-efficient manner. More specifically, subscribers express specific interests (subscriptions) that determine content filters which are installed on content-based routers along the paths between publishers and subscribers. These content filters ensure the forwarding of only relevant content to each interested subscriber, thus ensuring bandwidth efficiency in the pub/sub system. Over the past decade, content-based pub/sub has primarily been realized as an overlay network of software brokers. However, such broker-based routing and content filtering in software results in performance (e.g., end-to-end latency and throughput) that is far behind the performance of network layer implementations of communication protocols.

To mitigate the aforementioned problems and achieve line-rate performance in content-based pub/sub systems, the power of software-defined networking (SDN) has been harnessed in recent times. SDN has enabled the realization of a content-based pub/sub system [6, 29] that performs routing and filtering of content directly on the network layer, improving performance manifold. SDN makes this possible by offering standards like Openflow [14] that can be used to allow software to flexibly define the network. More specifically, SDN enables the extraction of all control logic from hardware switches and its hosting on a logically centralized server called controller, thus establishing a clear separation of the control plane and the data/forwarding plane. The logically centralized controller (control plane) has a global view of the entire network and can flexibly configure it in an optimal manner. Such capabilities are easily exploited by content-based pub/sub which uses the global network view at the controller to establish paths between publishers and subscribers. To this end, content filters are translated into forwarding rules or flows and installed on the Ternary Content Addressable Memory (TCAM) of switches along each path, thus enabling direct filtering of published content on hardware switches.

Clearly, the efficient mapping of content filters to forwarding rules or flows installed on TCAM is key to the expressiveness of an SDN-based pub/sub middleware. More specifically, a content filter is mapped to the match field (IP address, VLAN tag, etc.) of a forwarding rule which enables header-based matching of packets and subsequent forwarding based on the installed rule to interested subscribers. So, the importance of these forwarding rules (content filters) on TCAM is paramount as they directly impact the amount of unnecessary traffic in the network. As a result, the objective is to make these content filters as expressive as possible in order to ensure the forwarding of relevant traffic only. However, this

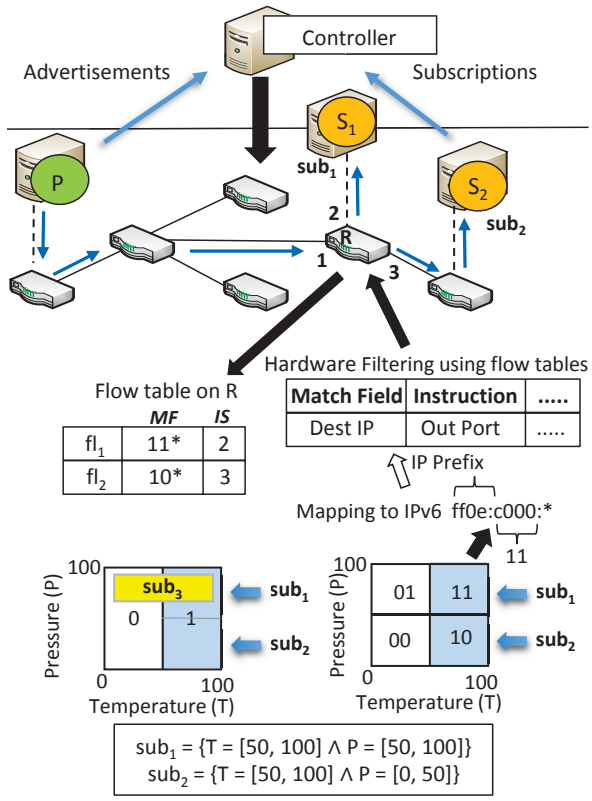
objective faces a serious challenge due to an inherent hardware limitation.

TCAM is an expensive, power-hungry resource and as a result the number of flow table entries (forwarding rules) available for content filtering is limited in hardware switches. Most switch vendors design Openflow-enabled switches that typically support up to a couple of thousands of flow entries per switch [16, 21], and such a hardware limitation has already been the subject of much research in the past [5, 16, 19, 21]. The main reason behind the design of TCAM with such limited space is the inherent trade-off between table size and other factors such as power and cost. In fact, studies show that compared to conventional RAM, TCAM consumes almost 100 times more power [28] and has almost 100 times more cost [3]. As a result, applications should only rely on a limited number of flow entries for their design. Moreover, considering traffic from various applications being routed over the switches of a network, the number of flow entries reserved for content-based routing is merely a fraction of the entire capacity of TCAM on a switch. To address such a limitation, Bhowmik et al., in [5], propose the installing of a subset of filters on hardware and the remaining on software by designing a hybrid pub/sub middleware where filtering of content occurs both in the application layer and the network layer. However, this implies higher end-to-end latency and reduced throughput rates for the traffic that is filtered in software.

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

2 SDN-BASED PUB/SUB MIDDLEWARE

In this section, we provide a brief overview of a content-based pub/sub middleware using SDN, i.e., PLEROMA [6, 29], already existing in literature that provides an insight on how existing systems realize SDN-based pub/sub.



sub1 = {T = [50, 100] ∧ P = [50, 100]}
sub2 = {T = [50, 100] ∧ P = [0, 50]}

Figure 1: SDN-based Pub/Sub Middleware

The content-based pub/sub middleware designed on software-defined networks, called PLEROMA, consists of mainly two participants—the publisher and the subscriber. A publisher, the producer of content, specifies the content it intends to publish by sending advertisements to the SDN controller. Similarly, a subscriber, the consumer of content, specifies content it is interested in receiving by sending a subscription to the controller. With the information of these advertisements and subscriptions, the logically centralized controller installs content filters on the path between each publisher and its interested subscriber. The widely adopted Openflow standard is used to deploy content filters as match field of flows in the TCAM of Openflow-enabled switches. For example, in Figure 1, the publisher P and the subscriber S_1 send an advertisement and a subscription respectively to the controller which installs content filters along the path between them. Now, when a publisher publishes content (events), header-based matching of the event packets against the installed flow entries is performed and packets are forwarded as dictated by the flow on account of a match at line-rate.

As impressed upon in the previous section, the effectiveness of a content-based pub/sub middleware largely depends on the efficient mechanism used to represent content expressively. For this purpose, a content-based subscription model where published events are represented as attribute-value pairs and advertisements and subscriptions (i.e., content filters) are represented as conjunction of filters on these attributes is used. Moreover, the content filters

need to be efficiently mapped to a match field of the flow table entries. Similarly, the events need to be mapped to the corresponding header field of the published packets. This is performed with the help of a two-step process.

The first step involves mapping values and ranges of values (content filters) along attributes to binary strings in order to represent them in flow table entries/packet headers. Various techniques may be employed to perform this conversion of content to binary form, which includes the use of bloom filters [8], spatial indexing [29], etc. Since, in literature (e.g. PLEROMA), spatial indexing has been primarily used for this purpose, in this paper too, we specifically look at spatial indexing and its effects on TCAM space. However, please note that the developed concepts in this paper also apply to other flavors of SDN-based pub/sub using different mapping techniques. In spatial indexing, the event-space (denoted by Ω) is represented geometrically as a ω-dimensional space where each dimension represents a content attribute. Recursive binary decomposition of Ω generates regular subspaces that serve as enclosing approximations for advertisements, subscriptions and events which are represented by binary strings known as *dz*s. To illustrate spatial indexing, let us take the example of Figure 1 where a subscriber S_1 has a subscription sub_1 : {T = [50, 100] $\wedge$ P = [50, 100]}. The recursive decomposition of Ω to the closest possible approximation of sub_1 yields the dz {11}.

The *dz*s have characteristic properties based on the subspaces they represent. For example, (i) the shorter a dz, the larger is the corresponding subspace it represents. This is depicted in Figure 1 where the dz {1} clearly represents a subspace which is larger than and contains the subspace {11}. So, more expressive a content filter is, more fine granular is the recursive binary decomposition and longer are the resulting *dz*s. The previous example also illustrates another property of spatial indexing that (ii) the dz of a subspace has a prefix equivalent to the dz of the subspace containing it. This property ensures that an event (i.e., a point in Ω), represented by a longer dz is considered a match for all subspaces containing it simply through a prefix match. Finally, (iii) spatial indexing may generate a set of *dz*s for the same subscription resulting in multiple flows for the same subscription. This can be seen in Figure 1 where the subscription sub_3 (represented by the yellow subspace) will need to be represented by at least two *dz*s, {01} and {11}, in the 2-bit representation of the subscription. In fact, further division of Ω will yield longer and more precise representation consisting of even more *dz*s for sub_3. Where on one hand multiple *dz*s for the same subscription adds to the expressiveness of filters, on the other hand they occupy more space in the expensive TCAM.

After binary representation of content, the second step is to map *dz*s representing content filters to the designated match field in flow entries and *dz*s representing events to the corresponding field in packet headers. To this end, a range of IP multicast addresses (e.g. IPv6) is chosen as destination IP addresses in the match field of flows as well as in the packet headers. Mapped *dz*s are appended to a fixed prefix in the destination IP address, e.g., ff0e (representing the IPv6 multicast address range available to pub/sub traffic). So, a subscription (i.e., content filter) is represented by an IPv6 multicast address which is used by the flow entries in the flow tables of switches for event matching and forwarding. The prefix matching of events against installed flows is enabled in IP addresses with the help of Class-less Interdomain Routing (CIDR) style masking supported by Openflow-enabled switches where masks are represented by the 'don't care' symbol (*). The entire process of converting content to match field of flows is illustrated in Figure 1, where, for example, the subscription sub_1 from S_1 is first spatially indexed into a dz {11}. Next, the dz is mapped to an IPv6 address which is used as a destination IP match field on all switches along the path between P and S_1. Similarly, the subscription sub_2 (i.e., {10}) from S_2 is converted to an IPv6 address and installed on all switches between P and S_2. The flow table of the switch R depicts the flow entry fields, i.e., match field (MF) and instruction set (IS), relevant to this middleware. For example, in fl_1 the IPv6 multicast address representing the dz {11*} constitutes the match field and the outgoing port 2 constitutes the instruction set which dictates the forwarding of an event packet through the specified port on account of a match. So, an event where the destination IP address field in the packet header represents the dz {110000101000} will be forwarded by R through outgoing port 2 as this event will match fl_1.

Just as there exists containment relations between *dz*s, similarly flows on a switch are related. In the context of this pub/sub middleware, flow relations may be defined as follows [29]. A flow fl_i covers (or contains) another flow fl_j, denoted by $fl_i \succ fl_j$, if the following two conditions hold: (i) the dz associated with the destination IP address in the match field of fl_j is covered by the dz of fl_i, and (ii) the out ports to which a packet matching fl_j is forwarded are subset of those specified in the IS of fl_i. Likewise, a partial containment relation ($\succsim$) can be defined between flows of a switch. A flow fl_i partially covers (or contains) another flow fl_j, denoted by $fl_i \succsim fl_j$, if dz associated with the match field of fl_i covers dz of fl_j, but not all the out ports used for forwarding packets matching fl_j are listed in the IS of fl_i. Otherwise, two flows are disjoint or unrelated. For example, in figure 1, the two flows fl_1 and fl_2 on R are unrelated as the *dz*s associated with their IP addresses in the match fields are unrelated. While installing a flow fl_n on a switch for a new advertisement or subscription, fl_n is installed only if there is no existing flow fl_e on that switch that covers fl_n. If $fl_e \succ fl_n$, then fl_n is redundant as the traffic for it is already forwarded by fl_e. In the reverse case, i.e., if $fl_n \succ fl_e$, fl_e is removed and fl_n installed for the same reason. A new flow is only installed if it is unrelated or has a partial containment relation with existing flows. This ensures that no redundant flows exist on any switch as TCAM is a scarce resource.

3 IMPACT OF TCAM LIMITATIONS

From the above discussion on spatial indexing of content and its subsequent conversion to flow entries, it is quite evident that more expressive representation of subscriptions demands the installation of multiple flows on a switch. In fact, as mentioned before, even a single subscription may yield multiple *dz*s which results in multiple flow entries. Also, typically, applications using content-based pub/sub may have up to millions of subscribers which might require deployment of millions of filters. With such high demand of TCAM resources, it is very natural to run out of TCAM space in such applications. A limited number of available flow table entries implies two paths of action—ignoring any subsequent subscription

filters once TCAM capacity is reached, or aggregating flows to reduce occupied TCAM space. The former will lead to false negatives (i.e., events that are not forwarded to interested subscribers), which is of course not acceptable in the context of this pub/sub middleware, and the latter may result in false positives (i.e., events that are forwarded to uninterested subscribers) which means unnecessary bandwidth usage. In this paper, we employ the latter, i.e., aggregate or merge filters, while also striving for bandwidth efficiency.

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

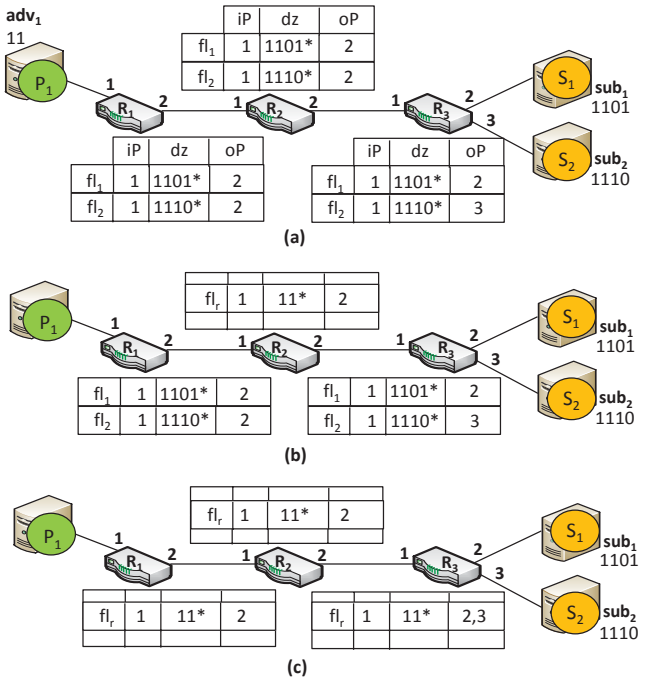


Figure 2: Importance of upstream switch filters

4 FILTER AGGREGATION PROBLEM

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

More formally, let $\mathbb{R}$ be the set of all switches in the network and F_{R_i} be the set of all flows that should be deployed for a given set of advertisements and subscriptions on switch R_i where $fl \in F_{R_i}$. Let cap_{R_i} be the maximum TCAM capacity of switch R_i available for pub/sub traffic. For each switch R_i, we need to determine a set of flows SF_{R_i} belonging to and aggregated from F_{R_i} that is within the given TCAM capacity. Let C_{R_i} be the aggregation cost, in terms of unnecessary traffic forwarded due to aggregation of filters, of R_i. So, our objective is to determine the set SF_{R_i} subject to $|SF_{R_i}| \leq cap_{R_i}$ for each $R_i \in \mathbb{R}$ such that $\sum_{i=1}^{|\mathbb{R}|} C_{R_i}$, i.e., overall unnecessary traffic in the network due to aggregation of filters, is minimum.

Problem Analysis : The defined problem specifies minimizing the aggregation cost on each individual switch such that the overall amount of unnecessary traffic in the network can be kept to a minimum. The aggregation cost of a switch is nothing but the sum of the aggregation cost of all the flow merges made on that switch to meet the TCAM capacity of that switch. As a result, while deciding on the filters to be aggregated on a switch, the aggregation cost of each possible merge should be calculated. However, just looking at the flow information local to a switch for a possible local merge is not the optimal way to determine its cost as our investigation into the defined problem shows that existing filters on other switches in the network have a significant role to play on the aggregation cost of a merge. In fact, the main challenge in the filter aggregation problem is the determination of the aggregation cost of each possible merge on a switch which depends largely on the already installed filters on switches in the upstream paths of the aggregated filter.

To understand the importance of filters on switches in the upstream paths of a filter being considered for a merge, we look at an example depicted in Figure 2. Figure 2(a) shows a system with a publisher and two subscribers and their respective advertisement and subscriptions which result in the deployment of the depicted flows on the three switches. Each flow is depicted by the flow name, incoming port (iP), dz constituting the destination IP address, and the outgoing ports (oP) in IS. Let us assume that R_2 can only accommodate a single flow. As a result, fl_1 and fl_2 on R_2 are merged, in the manner explained in Section 3, to compose fl_r as depicted in Figure 2(b). fl_1 and fl_2 had the same incoming and outgoing ports

and as a result only the filter subspace in fl_r gets expanded. Now, R_2 will forward all traffic matching {11*} instead of just {1101*} and {1110*}. However, if we look upstream, we see that R_1 will already filter out any traffic that does not lie within the subspaces {1101} and {1110} in Ω. As a result, the merge at R_2 does not impact the false positives in the system as R_2 does not receive any additional false positives from its upstream path and acts as only a forwarder in Figure 2(b). However, the scenario is different in Figure 2(c) where there is a need to merge the two flows on R_1. In this case, not only does R_1 forward all traffic matching {11*}, but also these false positives get forwarded by R_2 due to the aggregation of its filter. At R_3 too, owing to a merge, these false positives from previous switches do not get filtered out. In fact, as the resultant flow combines the outgoing ports of the two original flows fl_1 and fl_2 on R_3, false positives are now forwarded along both downstream links of R_3. Port 2 forwards false positives lying within the subspace {11 - 1101} and port 3 forwards those lying within {11 - 1110}. If the upstream filters at R_1 or R_2 were precise, then fl_r on R_3 would only forward false positives lying within {1110} through port 2 and {1101} through port 3 as the remaining would be filtered out upstream. This example clearly indicates that even if filter expansion occurs, false positives forwarded by a merged flow depends on the filter aggregation on upstream switches. Also, even if no filter aggregation occurs on the upstream path, an aggregation involving merging of two or more flows whose outgoing ports are not subsets of each other will result in traffic meant to be forwarded by one port being forwarded by the remaining ones as well. This will always result in false positives along all involved outgoing ports (cf. R_3 in Figure 2(c)).

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

5 FILTER AGGREGATION ALGORITHM

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

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

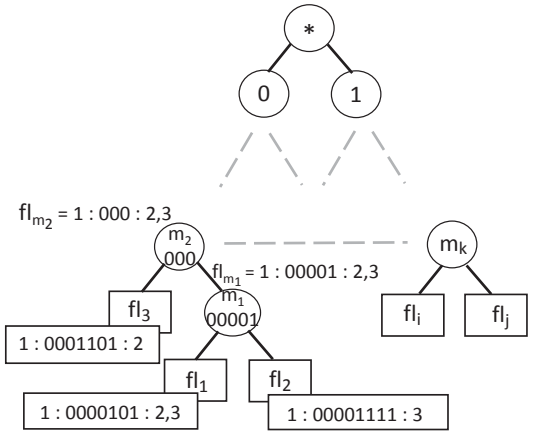


Figure 3: Merge Point Tree for Incoming Port 1

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

5.1 Selecting Flow Merges on a Switch

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

So, a merge point tree is a prefix tree where every non-leaf node is a merge point and every leaf node is a flow. In the tree, a merge point signifies the minimum filter expansion required to cover two

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

Once all merge points are determined for the switch being processed, the aggregation cost for each of them is calculated to determine the final set of merges on the switch. We explain the cost calculation at each merge point in details in the following subsection 5.2. Having calculated the aggregation cost (C) for every possible merge point across all merge point trees of a switch, we also determine the benefit (denoted by B) of each merge. The benefit is, simply, the number of flows reduced on the switch due to the merge (i.e., B_m = number of flows covered by merge point m - 1). Next, we calculate the cost per benefit of each merge point m, i.e., C_m/B_m and sort the merge points on a switch in the increasing order of cost per benefit. For a switch, say R_i, we start selecting merge points with minimum cost per benefit for the final set of selected flows (SF_{R_i}) on R_i. We start with all original unaggregated flows in SF_{R_i}. Once a merge point m gets selected, (i) all original flows covered by m are removed from SF_{R_i}, (ii) the flow fl_m at m is added to SF_{R_i}, (iii) the next step depends on one of the following

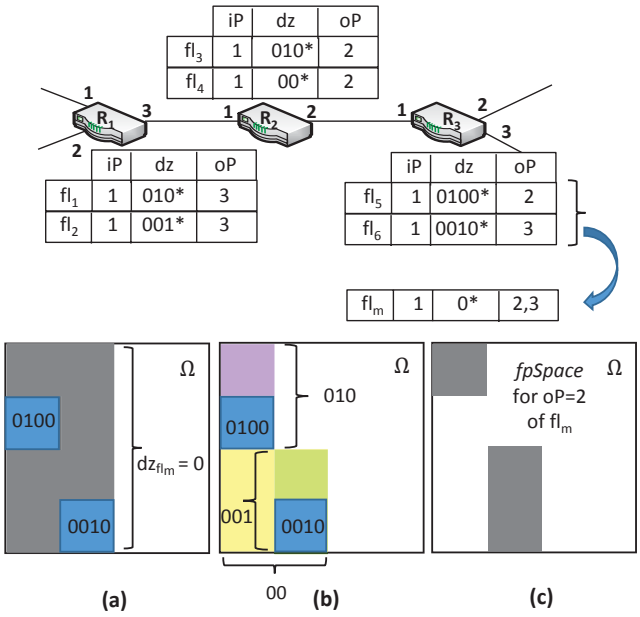


Figure 4: Cost Calculation

three scenarios—(a) if a merge point m_j is covered by the selected merge point m (i.e., $m \succ m_j$), then m_j is removed from the set of merge points as it is now redundant, (b) if a merge point m_j covers the selected merge point m (i.e., $m_j \succ m$), then m_j cannot be removed but its cost and benefit get reduced as some of its cost and benefit has already been considered when selecting m, and (c) if m has no relation with any other merge point then no action is taken. After each selection, the set of merge points is re-sorted and the next merge point with least cost per benefit is selected until $|SF_{R_i}| \leq cap_{R_i}$, i.e., the number of flows on R_i is within the TCAM capacity of the switch.

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

5.2 Aggregation Cost at a Merge Point

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

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

publisher intended to be forwarded along *iPaths* and the upstream switch filters which influence the filtering of this published traffic.

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

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

$$C = \sum_{p \in iPaths} \sum_{op \in oP} fp_{op}^{p} * dLinks_{op} \quad (1)$$

In fact, the two flavors of cost calculation proposed in this paper—load-based method and pattern-based method—differ only in the manner of determining the false positive value, i.e, fp in Equation 1 as discussed later in this section. The load-based method estimates resultant network false positives due to the merge by using the knowledge of incoming traffic load, whereas the pattern-based method collects and inspects published event packets to accurately determine network false positives introduced due to the merge by not only using the knowledge of traffic load but also traffic pattern in Ω.

In the remaining part of this section, we introduce the details of (i) determining the incoming traffic at a merge point and (ii) determining false positives from this incoming traffic along the downstream paths of the merge point. As explained above, these two steps together determine the aggregation cost, C, of a merge point. We take an example from Figure 4 to explain the steps of cost calculation.

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

As mentioned earlier, the amount of incoming traffic along a path depends on the filters installed on the upstream paths of the merge point. So, for a path $p \in iPaths$, first, we determine the set of all upstream filters, i.e., $uFilters$. This is, effectively, the set of filters on all upstream switches on the current path that forward events to fl_m. As explained in Section 4, these upstream filters/flows are of utmost importance in determining the aggregation cost. In fact, the most fine-grained filters among this set dictate the incoming traffic for fl_m as these filters already filter out the bulk of unnecessary traffic. So, it is imperative to only identify the set of most fine-grained filters, i.e., $mfgFilters \in uFilters$, as the traffic forwarded by them is the only traffic that reaches fl_m. We denote the set of dzs representing the filter subspaces of $mfgFilters$ as $mfgDzs$. Let us look at an example from Figure 4 where the aggregation cost of m needs to be calculated on switch R_3 and $F_m = \{fl_5, fl_6\}$. The set of relevant flows on upstream switches, i.e., $uFilters$, of a path p that connects a publisher *pub* to fl_m, consists of fl_1, fl_2, fl_3, and fl_4 as depicted in the figure. Figure 4(b) illustrates the event subspace representation of each of the filter dzs of $uFilters$ and Figure 4(a) depicts that of fl_m. From the figure, it is quite clear that $\{dz_{fl_4} = 00\} > \{dz_{fl_2} = 001\}$. This means that fl_2 already filters out events lying in the subspace { 00 - 001} depicted by the yellow subspace in Figure 4(a). As a result, all events lying within the yellow subspace do not reach the incoming port of fl_m and so this subspace cannot be considered as a false positive subspace in the aggregation cost of fl_m. So, in this example, $mfgFilters = \{fl_1, fl_2, fl_3\}$ and the effective subspaces they represent, i.e., the most fine-grained dzs, $mfgDzs = \{001, 010\}$. Also, only those subspaces in $mfgDzs$ are considered to contribute to the incoming traffic that lie within the advertised subspace of the current publisher as the remaining subspace cannot be accounted for any incoming traffic of fl_m due to the absence of published events lying within them. So, all traffic lying within $mfgDzs$ can now be considered as the incoming traffic at a merge point m for a path p.

5.2.2 False Positives on Downstream Paths. Having identified the subspaces that forward traffic to the incoming port of fl_m along a specific path, we proceed to calculate the false positives lying within these subspaces that will be forwarded by the current merged filter fl_m along its downstream paths. However, before we do so, please recall that if fl_m aggregates original flows which have different outgoing ports, then the false positives for one outgoing port may be different from those of the others. As a result, we calculate the amount of false positives that may be forwarded by each outgoing port of fl_m separately and compute the sum of the individual costs of each outgoing port $\in oP$ of fl_m to obtain the total aggregation cost of fl_m for the specific path. So, for each outgoing port, $op \in oP$, we identify the original flows, i.e. $F_{op} \in F_m$, that should forward traffic through the current outgoing port *op*. With the information of the dzs of the flows that are originally supposed to forward events through the current port, it is easy to determine the false positive subspace ($fpSpace$) for *op*. So, the effective false

positive subspace ($fpSpace$) for each port of an aggregated filter can be computed by subtracting the dzs of all flows belonging to F_{op} from the subspace representing $mfgDzs$. Events lying in $fpSpace$ are the only unnecessary events that will be forwarded by the current outgoing port of the aggregated flow fl_m. For example in Figure 4(c), the gray area is the effective $fpSpace$ for outgoing port 2 of fl_m. All events, published by the publisher pub, that lie in this subspace account for the aggregation cost of port 2 for the specific path p.

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

Load-based Method (FA-PB): The load-based method uses the traffic load of the publisher along the current path in consideration and the value of $fpSpace$ to estimate the false positives of the outgoing port. More specifically, it collects statistics related to the total number of events ($p_{traffic}$) published by the current publisher in the advertised subspaces ($adSpace$) and estimates the false positives within $fpSpace$. To calculate this estimated number of false positives, we quantify (q) subspaces as a fraction of the entire event space Ω. So, in Figure 4, while calculating false positives forwarded by port 2, the q value for the subspaces representing $mfgDzs$ is 1/4, that of the subspace representing the dz of F_{op}, i.e.,{0100}, is 1/16, and, therefore, that of $fpSpace$ = 3/16. So, using the quantified values for the subspaces, the estimated false positives within $fpSpace$ are $(fpSpace/adSpace)^* p_{traffic}$.

As mentioned earlier, the aggregation costs of all outgoing ports of fl_m are summed up to calculate the aggregation cost of a path and then the aggregation costs of all paths are summed up to calculate the total aggregation cost of a merge point. So, we formally define the aggregation cost of a merge point using load-based method by extending Equation 1 as follows:

$$C = \sum_{p \in iPaths} \sum_{op \in oP} (fpSpace^p_{op}/adSpace^p) * p^p_{traffic} * dLinks_{op} \tag{2}$$

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

More specifically, in this method, we collect published events from all publishers. For a given path p, the exact number of forwarded false positives ($pb\_fp^p_{op}$) can be determined for each $op \in oP$ of fl_m depending on the calculated $fpSpace$ and the events published on that path by investigating the content of each event and determining whether it lies within the $fpSpace$ in question. So, we define the aggregation cost of a merge point using pattern-based method by again extending Equation 1 as follows:

$$C = \sum_{p \in iPaths} \sum_{op \in oP} pb\_fp^p_{op} * dLinks_{op} \tag{3}$$

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

5.3 Resolving Dependencies Between Switches

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

5.4 Handling Dynamics

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

So, when a subscription/advertisement arrives at the controller, the usual flow installation is performed for each dz representing it. While installing a flow for a particular dz, say dz_{sub}, on a specific switch, say R, the controller discovers that the capacity of that

switch is already full. As a result, to accommodate the new flow, an aggregation of at least 2 flows must be performed on R and the local aggregation approach is employed for this purpose. The main idea behind the local aggregation approach is to simply merge two flows without conflicting relations (cf. Section 5.1) on a switch with exceeded capacity such that only the knowledge of the state local to a switch is required for aggregation. We, again, use the merge point trees local to the switch for this purpose. As only a single flow needs to be reduced, only the merge points connected to the leaf nodes (flows) in the lowest level of the tree are considered for aggregation. Please recall that filter expansion of involved flows is the least in the lowest level and increases as we go up the tree towards the root. So, the local aggregation approach selects any one of these merge points merging flows at the lowest tree level whenever a switch exceeds its capacity on advent of a subscription or advertisement. Such an approach portrays an aggregation technique with least overhead.

6 PERFORMANCE EVALUATIONS

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

Experimental Setup: We perform our performance evaluations mainly under two test environments— the very prominent tool, Mininet [22], for emulating a variety of networks and an SDN-testbed consisting of a Whitebox Openflow-enabled EdgeCore switch and commodity PC hardware. To show the applicability of our algorithms in a real SDN environment, we create a hierarchical fat-tree topology consisting of 10 switches and 8 end-hosts on the SDN-testbed where the switches are created by partitioning the hardware Whitebox switch running the network operating system PicOS (version 2.6) [1, 2]. The 8 end-hosts reside on commodity rack PCs and perform the role of publishers and subscribers. The SDN controller is hosted on a 3.10 GHz machine with 40 cores. However, to fully explore the various aspects of our system, we also require very large and flexible topologies. We use Mininet for this purpose which enables us to experiment with various topologies and published traffic. In the emulated Mininet environment, the SDN controller connected to the emulated network is hosted on a 3.2 GHz machine with 4 cores. To show the impact of severe TCAM limitations on the performance of the system and how the designed aggregation ensures bandwidth efficiency despite severe constraints, we constrain the TCAM capacity (i.e., *cap*) of each switch to up to 600 flows. We experiment with up to 300 switches and 4402 end-hosts on different topologies. In fact, to capture the false positives along every link of the network and gather the overall network false positives, we also implement our own analyzer.

We use both synthetic and real-world data for our experiments. To generate synthetic data, we use a content-based schema that uses up to 5 attributes, where the domain of each attribute varies between the range [0,1023]. Our evaluations include up to 15,000 subscriptions and up to 100,000 events. We primarily use two models, predominantly used in literature [13, 26], for the distribution of subscriptions and events. The uniform model generates subscriptions and events independent of each other uniformly in the event space, whereas, the interest popularity model selects up to 8 hotspot regions around which it generates workload using the widely used zipfian distribution. We also use real-world workload in the form of stock quotes procured from Yahoo! Finance containing a stock's daily closing prices [12] to show the performance of our system in a realistic environment.

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

Figure 5(a) and Figure 5(b) show the false positive rate when each of the aggregation algorithms are applied to a network having a regular tree topology for different workload distributions. Figure 5(a) depicts a scenario where workload is generated using uniform distribution whereas Figure 5(b) shows the behavior of the algorithms when zipfian distribution is used. In both scenarios, the local aggregation approach is heavily outperformed by the other two as a result of performing aggregation based on local switch state as compared to the two flavors of the filter aggregation algorithm which consider a holistic view of the network for filter aggregation for both distributions. The amount of false positives in the network on using LA for aggregation clearly shows the importance of having a more refined algorithm for aggregation. In case of uniform distribution in Figure 5(a), we see that the performances of FA-LB and FA-PB are almost equivalent. The main difference between FA-LB and FA-PB is that FA-PB analyses each of the event packets to determine the amount of false positives along each path whereas FA-LB determines the amount of traffic on each path and then estimates the amount of false positives while considering the traffic to be distributed uniformly over the advertised subspace for that path. As a result, the two methods behave very similarly for uniform distribution as the estimate of false positives is almost identical to the actual false positives in the network. However, for the same reason, the advantage of FA-PB over FA-LB is very apparent in Figure 5(b) where FA-PB clearly outperforms FA-LB as the decision-making process in FA-PB considers the exact nature of published events that follow a zipfian distribution. In fact, when using FA-PB, the false positive rate is reduced by up to 99.9% as compared to LA and is almost non-existent even on aggregating a large number of subscription filters in the system, highlighting the effectiveness of the filter aggregation algorithm proposed in this paper. This is mainly because, with zipfian distribution, FA-PB can take efficient decisions to merge flows which do not experience

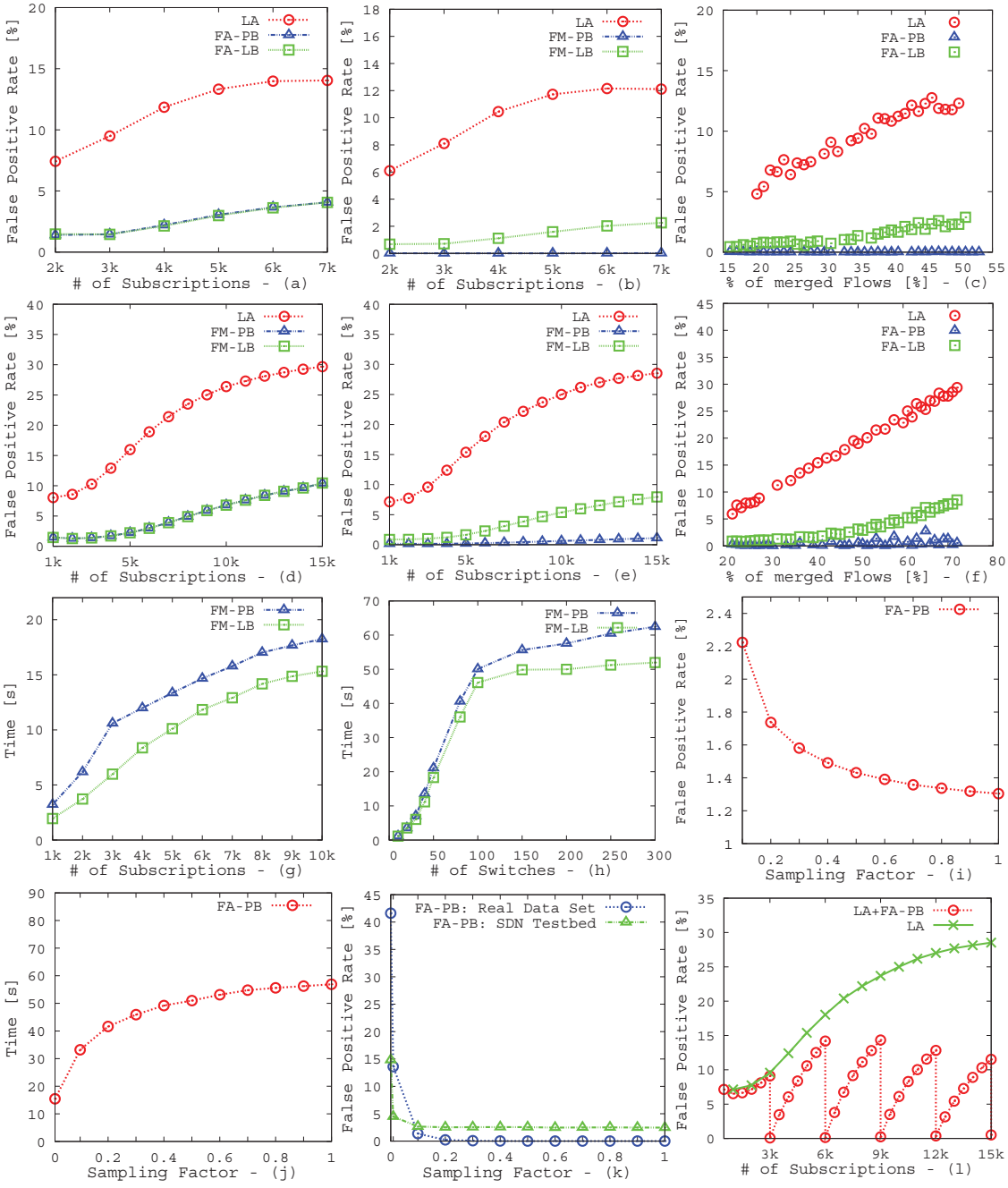


Figure 5: Performance Evaluations

too much traffic and therefore less false positives while preserving flows relevant to traffic hotspots. In fact, we depict the false positive rate vs total percentage of merged flows in the network in Figure 5(c) for zipfian distribution. This graph shows the impact of flow aggregation on false positives. Of course, more the number of merged flows (i.e., aggregation) in the network, more is the false positive rate. The plots show that even when a large percentage of flows are aggregated, it results in very low false positives for FA-PB. In fact, even when over 50% of flows are merged, the false positives in the network are negligible implying that the TCAM constraint does not adversely impact the system if FA-PB is used for aggregation. The performance of FA-PB is closely followed by FA-LB which is followed by LA.

To show the effectiveness of the proposed algorithms irrespective of the type of topology, we also conducted experiments on a random topology as depicted in Figure 5(d) and Figure 5(e). Here

too, we see the same behavior of the algorithms as in the tree topology. As before, FA-PB and FA-LB perform similarly in the case of uniform distribution as depicted in Figure 5(d). In the case of zipfian distribution, again, on performing aggregation using FA-PB, the false positives in the network are almost negligible despite aggregating a large number of subscription filters as depicted in Figure 5(e). We, also, show a graph to depict false positive rate vs total percentage of merged flows in the network in Figure 5(f) for the random topology when zipfian distribution is used. As can be seen in the figures, the comparison of performance in terms of false positive rate of the various algorithms is similar to that for the other topology which implies that the behavior of the algorithms is not specific to a type of topology.

Comparing Runtime Overhead: The effectiveness of the algorithms w.r.t. bandwidth efficiency is clear from the above discussion. However, where FA-PB outperforms the others in bandwidth efficiency, the others come with lower overhead. The higher overhead in FA-PB is not only due to the fact that published events need to be collected from the publisher but also due to a higher runtime overhead than the others. We confirm the same in our next set of experiments depicted in Figure 5(g) where we compare the two flavors of filter aggregation algorithm. We measure the runtime overhead with increasing number of subscriptions. Again, note, FA-PB has a sampling factor of 1 in this set of experiments. As depicted in the figure, FA-PB has a higher runtime overhead than FA-LB consistently as it additionally considers event traffic patterns for cost calculation. Our evaluations also show that the average runtime overhead for LA is merely 200 microseconds on a switch . So, we see that there is a trade-off between accuracy and overhead as the improvement in one adversely affects the other.

We also evaluate the runtime overhead of the two flavors of filter aggregation algorithm with increasing topology size. In this experiment we keep the number of publishers and subscribers fixed and expand the topology in terms of number of switches. Of course, more the number of switches more will be the overhead for both FA-PB and FA-LB as the cost calculation has to be done over more switches with each calculation considering longer paths (more upstream filters). Such a behavior is visible in Figure 5(h) where the overhead for both FA-PB and FA-LB increases with increasing number of switches. Again, FA-PB has higher overhead than FA-LB due to the aforementioned reasons.

Impact of Sampling Factor: To reduce the overhead of collecting published events and cost calculation of FA-PB, we introduced the sampling factor (i.e., sfr) in Section 5. In the next set of experiments, we show the behavior of our system when subjected to various sampling factors. Figure 5(i) plots the false positive rate with increasing value of sfr for zipfian distribution. As expected, more the value of sfr, fewer are the false positives as FA-PB is more accurate in its cost calculation when it considers more past events. However, higher the sampling factor of FA-PB, higher is the overhead as depicted in Figure 5(j) where we plot the runtime overhead for increasing values of sfr.

To ensure that our aggregation algorithm is effective in realistic scenarios, we conducted experiments to show its behavior on real-world stock data. Figure 5(k) plots the false positive rate with increasing sampling factor for the real-world data set. The plot clearly shows that, even for a sampling factor of just 0.4, the network false positives due to aggregation are almost non-existent. These evaluation results further highlight the applicability and efficiency of the algorithm presented in this paper.

For our next set of experiments, we measure the false positives at the subscribers when the aggregation algorithm is deployed on a real SDN testbed described in the experimental setup of this section. So, Figure 5(k) also plots the false positive rate when FA-PB aggregates flows for increasing sampling factors for workload generated using zipfian distribution. The graph shows that on the real SDN testbed, the algorithm behaves as expected and the false positive rate decreases rapidly with increasing sampling factor.

Dynamic Behavior: In our final set of experiments, we evaluate the performance in terms of false positive rate of our system in a dynamic environment. We progressively introduce subscriptions in the system and apply our aggregation algorithm for handling dynamics. So, in general, the local aggregation approach is applied whenever switches exceed their capacity on introduction of a new subscription as explained under handling dynamics in Section 5. Additionally, FA-PB is employed after every 3000 subscriptions as depicted in Figure 5(l). The figure shows that the false positive rate gradually increases with more and more subscriptions when LA is used till FA-PB is performed which makes the false positives in the system almost negligible. Again, the false positive rate keeps increasing on using LA till the next application of FA-PB. We, also, plot the behavior of the system if only LA is employed. This clearly shows the amount of false positives reduced in the system at every step due to the intermittent application of FA-PB.

7 RELATED WORK

In the past couple of decades, a significant amount of research has been dedicated to broker-based middleware implementations of content-based pub/sub [11, 18, 25] that focus on achieving scalability. In this context, techniques for subscription summarization that include subscription covering [11] and subscription merging [25] are widely employed to realize scalable systems. Subscription summaries not only help in filtering out of events from the parts of the broker network without interested users but also ensure forwarding of new subscriptions only to brokers which previously do not receive subsuming (or covering) subscription summaries. So, these systems, primarily, use subscription summarization to reduce unnecessary message overhead in the broker network. Also, much effort [10, 15, 18] has been devoted to reduce the overhead of maintenance of these subscription summaries but from the context of efficiently handling dynamically changing subscription requests in the broker network. Implemented in the application layer, these systems do not need to address the problems of limited hardware space in the network layer to accommodate the subscription filters and subscription summarization is primarily performed to achieve bandwidth efficiency in broker networks.

However, in recent times, network layer implementations of pub/sub middleware [6, 7, 17, 29] has gained considerable popularity as they exploit the capabilities of SDN to achieve forwarding of events at line-rate. For example, in PLEROMA [6, 29], content filters are directly installed in TCAM of hardware switches enabling line-rate performance. However, existing SDN-based implementations do not consider the constraints on TCAM space (no. of

flow table entries) while deploying content filters on them in their design [4, 7, 29]. Besides the methods used to map content filters to flows on switches in PLEROMA, other methods to do the same could also be employed. For example, P4 [9], the protocol-independent programming abstraction provides complete flexibility to implement new protocols and headers such that packets can be processed independent of the hardware target. However, the flexible mapping of content filters to flows using P4 will still face the problems of limited TCAM space as no matter how the mapping is performed, all expressive content filters must be installed on TCAM to reap the benefits of line-rate forwarding. Bhowmik et al. [5] propose a hybrid pub/sub middleware which may be used to offload some of the content filters to the application layer resulting in filtering of events in both software and hardware. However, such a middleware loses some of the advantages of a pure network layer implementation.

In general, the problem of limited flow table entries in TCAM of SDN-compliant switches is well-known and much researched [19, 21, 30]. For example, Katta et al. [21] in CacheFlow use rule dependencies to cache the more popular flows on the limited TCAM while the remaining traffic is left to rely on software. As in the aforementioned hybrid pub/sub, here, too, the performance of the traffic forwarded by the software switch will suffer. Significant amount of work in literature deals with optimizing rule placement in a software-defined network [19, 20, 27]. For example, OneBigSwitch [19] uses endpoint policy and routing policy to aggregate sets of rules in order to take decisions on distributing them over network switches. However, it is incapable of handling scenarios where the rule sets are larger than the aggregate table size. Also, considerable amount of work has been done in the lines of compacting the representation of flow rules for the purposes of reducing TCAM space [23, 24]. Although, the aforementioned systems target efficient network provisioning and compressing rules on a switch, the proposed solutions are not applicable to our problem in the context of content-based routing. Also, systems, such as SmartTime [30], use an adaptive timeout technique to pro-actively evict flow rules while ensuring that there is minimum TCAM misses. Since, we consider a system that does not allow false negatives and filters can only be removed on account of an unsubscription or unadvertisement, a timeout-based heuristic is not ideal for our problem.

8 CONCLUSION

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

REFERENCES

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

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

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

[4] S. Bhowmik, M. A. Tariq, J. Grunert, and K. Rothermel. Bandwidth-efficient content-based routing on software-defined networks. In *Proc. of the 10th ACM Int. Conf. on Distributed Event-Based Systems*, DEBS 2016.

[5] S. Bhowmik, M. A. Tariq, L. Hegazy, and K. Rothermel. Hybrid content-based routing using network and application layer filtering. In *Proc. of 36th IEEE Int. Conf. on Distributed Computing Systems*, ICDCS '16.

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

[7] 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 *Proc. of the 9th ACM Int. Conf. on Distributed Event-Based Systems, DEBS*, 2015.

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

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*

[10] F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *Proc. of 23rd IEEE INFOCOM*, 2004.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 2001.

[12] A. K. Y. Cheung and H. Jacobsen. Green resource allocation algorithms for publish/subscribe systems. In *Int. Conf. on Distributed Computing Systems*, 2011.

[13] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In *Proc. of the Int. Conf. on Distributed Event-based Systems*, 2007.

[14] O. M. E. Committee. *Software-defined Networking: The New Norm for Networks.* Open Networking Foundation, 2012.

[15] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *Proc. of ACM Symp. on Applied Computing (SAC)*, 2004.

[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proc. of the ACM SIGCOMM 2011 Conference.*

[17] A. Hakiri and A. S. Gokhale. Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking. In *Proc. of the 10th ACM Int. Conf. on Distributed Event-Based Systems DEBS, 2016.*

[18] K. R. Jayaram, C. Jayalath, and P. Eugster. Parametric subscriptions for content-based publish/subscribe networks. In *Proc. of 11th Int. Conf. on Middleware*, 2010.

[19] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proc. of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13.

[20] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *Proc. of the IEEE INFOCOM 2013.*

[21] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. of the Symposium on SDN Research*, SOSR '16.

[22] B. Lantz, B. Heller, and N. McKeown. A network on a laptop: Rapid prototyping for software-defined networks. In *Proc. of 9th ACM Workshop on Hot Topics in Networks*, 2010.

[23] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: a systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM IEEE/ACM Transactions on Networking*, 2010.

[24] C. R. Meiners, A. X. Liu, and E. Torng. Bit Weaving: A non-prefix approach to compressing packet classifiers in tcams. *IEEE/ACM IEEE/ACM Transactions on Networking*, 2012.

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

[26] V. Muthusamy and H.-A. Jacobsen. Infrastructure-free content-based publish/subscribe. *IEEE/ACM IEEE/ACM Transactions on Networking*, 2014.

[27] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti. Optimizing rules placement in openflow networks: Trading routing for better efficiency. In *Proc. of the 3rd Workshop on HotSDN '14.*

[28] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *Proc. of the 11th IEEE ICNP Conference*, 2003.

[29] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel. PLEROMA: A SDN-based high performance publish/subscribe middleware. In *Proc. of 15th Int. Middleware Conference*, 2014.

[30] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya. Effective switch memory management in openflow networks. In *Proc. of the 8th ACM Int. Conf. on Distributed Event-Based Systems*, DEBS '14.