PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware

Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, Kurt Rothermel
University of Stuttgart
{first name.last name}@ipvs.uni-stuttgart.de

ABSTRACT

With the increasing popularity of Software-defined networks (SDN), TCAM memory of switches can be directly accessed by a publish/subscribe middleware to perform filtering operations at low latency. This way two important requirements for a publish/subscribe middleware can be fulfilled: namely bandwidth efficiency and line-rate performance in forwarding messages between producers and consumers. Nevertheless, it is challenging to sustain line-rate performance in the presence of dynamic changes in the interest of producers and consumers. In this paper, we propose and evaluate the PLEROMA middleware to realize publish/subscribe at line-rate and bandwidth efficiently in SDN. PLEROMA offers methods to efficiently reconfigure a deployed topology in the presence of dynamic subscriptions and advertisements. Furthermore, PLEROMA ensures interoperability and independent reconfiguration of multiple controlled SDN networks.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Distributed networks; C.2.4 [**Distributed Systems**]: Distributed applications; D.2.11 [**Software Architectures**]: Data abstraction

Keywords

Content-based Routing, Publish/Subscribe, Software-defined Networking, Network Virtualization

1. INTRODUCTION

Publish/Subscribe is a universal paradigm to mediate information (events) in a bandwidth efficient manner between multiple consumers (subscribers)and multiple producers (publishers) of information. Each subscriber expresses in a subscription its interest in events and in turn receives all events matching the expressed interest. Therefore, publish/subscribe can yield significant saving in terms of bandwidth usage especially if the interest in information is highly diverse.

Middleware'14 December 08 - 12 2014, Bordeaux, France
Copyright 2014 ACM 978-1-4503-2785-5/14/12...$15.00
http://dx.doi.org/10.1145/2663165.2663338

In publish/subscribe systems, (e.g. [2, 19, 8, 25]), saving bandwidth is commonly achieved by installing filters inside the network. Filtering is performed by specific components called *brokers* which are placed inside the network and mediate events between subscribers and publishers. Broker-based filtering in a publish/subscribe system imposes a significant delay by extending the end-to-end path length with a detour to the brokers and a processing delay for matching events against installed filters.

Recent progress towards the software-defined configuration of computer networks, however, poses a high potential to reduce this delay and apply publish/subscribe in delay sensitive application fields to quickly react to events, e.g., financial applications, traffic control, online gaming, manufacturing, or the smart grid. Standards like OpenFlow [3][1] specify the interface to directly install and modify flows on switches and install dedicated communication paths between the hosts connected to the network. To match the received packets efficiently against the installed flows, switches are equipped with a specific hardware called TCAM memory. Interestingly, the matching operations performed by TCAM memory are equivalent in expressiveness to many filtering operations needed in content-based publish/subscribe systems [13] and can significantly outperform software implementations. Therefore, in a software-defined network, it is possible to install a network topology which is not only bandwidth efficient, but also yields line-rate performance in forwarding events between producers and consumers.

While a deployed network topology offers line-rate performance in forwarding events at the data-plane, the topology needs also to be dynamically updated with ongoing subscriptions and advertisements. In software-defined networks this is the task of the controller, a logically centralized component which is in charge of installing/modifying flows in switches and can access the switches via a dedicated control network. The design of the control algorithm is therefore crucial for the performance of a publish/subscribe middleware in the presence of dynamic subscriptions and unsubscriptions.

In this paper we focus on three main requirements towards achieving scalability for SDN-configured publish/subscribe:

1. Publish/subscribe should in the presence of subscriptions and advertisements offer a low latency until sub-

[1]The OpenFlow standard is defined by the OpenNetwork Foundation, and many switch vendors such as CISCO, HP, and NEC currently implement the OpenFlow as part of their products

scribers can react to published events. For example, financial trading, traffic monitoring, or online gaming are not only known to be highly latency sensitive applications, but also highly dynamic with respect to the interests of publishers and subscribers [10, 15]. The subscriptions and advertisements often depend on the context on which subscriptions and publications are performed. In order to analyse the trend of stocks and quotes, the threshold for receiving events is updated in the time-scale ranging from just a few seconds to several hours for a single subscription [10]. Traffic monitoring and online gaming require location-dependent updates of run-time parameters such as the location of objects, often at larger frequency than one update per minute per subscriber [15].

2. Publish/subscribe should offer interoperability between multiple independently manageable network domains. Independently managed network domains naturally arise in many business systems, for instance to avoid interference of manufacturing processes and enforce security policies in accessing events [22, 24]. In addition, a partitioning of a large network can increase the scalability by performing multiple concurrent advertisements and subscriptions.

3. The control algorithm must be efficient in the number of flows installed in a switch. Note, that the cost for TCAM is critical in the design of switches. Therefore, vendors offer only a limited set of flows which is currently in the order of 40,000–180,000 flow entries per switch [5].

The contributions of this paper are a detailed performance evaluation of SDN-configured publish/subscribe systems and methods for their scalable reconfiguration addressing the aforementioned requirements. In Section 2, we first give an overview of the PLEROMA middleware. Section 3–5 present mechanisms that i) achieve efficient reconfiguration of the network topology, ii) ensure interoperability of multiple independently controlled SDN networks, and iii) limit the number of flows to be installed inside an SDN switch. Finally, in Section 6 we present detailed performance evaluation of PLEROMA on a real SDN testbed [12] and Mininet [16] environment with respect to event forwarding performance (throughput and delay), reconfiguration delays, control overhead in managing multiple controllers and bandwidth efficiency. We conclude with a comparison to state of the art publish/subscribe systems and future work.

2. THE PLEROMA MIDDLEWARE

Before introducing the methods for managing dynamic subscriptions and unsubscriptions we present the main components of the PLEROMA middleware. The middleware manages one or multiple interconnected network partitions (cf. Figure 1). The network partition is composed of a set of SDN-configurable switches. Two neighboring network partitions are interconnected to each other by a pair of border switches. All switches of a partition are controlled by a single controller which knows the entire network topology of a partition including outgoing links to neighboring partitions.

Each controller runs the PLEROMA middleware which configures the network topology to comprise short and bandwidth efficient paths between publishers and subscribers.

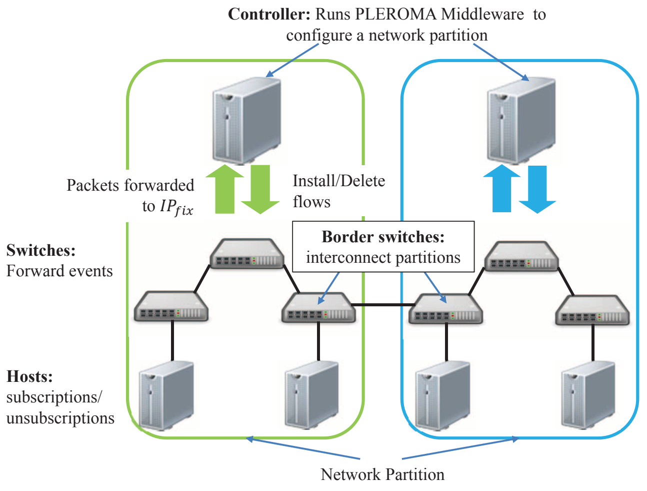

Controller: Runs PLEROMA Middleware to configure a network partition
Packets forwarded to IP_{fix}
Install/Delete flows
Switches: Forward events
Border switches: interconnect partitions
Hosts: subscriptions/ unsubscriptions
Network Partition

Figure 1: Two interconnected network partitions: Each controler autonously configures its network partition using the PLEROMA middleware.

The network topology is changed by installing or removing flows on the SDN-switches. In this paper, we follow the widely accepted OpenFlow standard to perform such updates. Consequently, each flow will describe any content which can be matched against the header fields, such as VLAN tags, a MAC address, or IP address [20]. A flow further defines an outgoing port of a switch to which a packet with a matching header field is forwarded. Note that not all of these fields are equally well suited to describe a network topology for publish/subscribe. For example, configuring MAC addresses can easily conflict with other services operated at the link layer. Therefore, in this paper we assume that PLEROMA will only install flows with respect to the fields corresponding to IP-Multicast addresses.

An update of the network topology is initiated whenever the controller receives an advertisement or a subscription request. Advertisements, publications, and subscriptions are performed by hosts connected to the SDN-switches of the network topology. A publisher is required to send an advertisement before sending any publications, while a subscriber is required to perform a subscription before receiving any messages. Since hosts are not directly connected and aware of the SDN-control network, each publisher/subscriber is required to send an advertisement/subscription to a specific IP address, IP_{fix}, which has been reserved specifically for the purpose of performing advertisements and subscriptions. No switch will install a flow with respect to IP_{fix} and therefore, all packets carrying this address will be directly forwarded from a switch over the control network to the controller.

To ensure high expressiveness and establish paths with low-bandwidth usage between publishers and subscribers, we follow the content-based subscription model, i.e., an event is composed of a set of attribute value pairs. The event space Ω, i.e., the set of all possible events that can be disseminated by the publishers, can be interpreted by a multi-dimensional space of which each dimension refers to the values of a specific attribute. An event is simply represented as a point in this space. Building on the principle of spatial indexing [13], we can divide the event space into regular subspaces that serve as enclosing approximations for events, advertise-

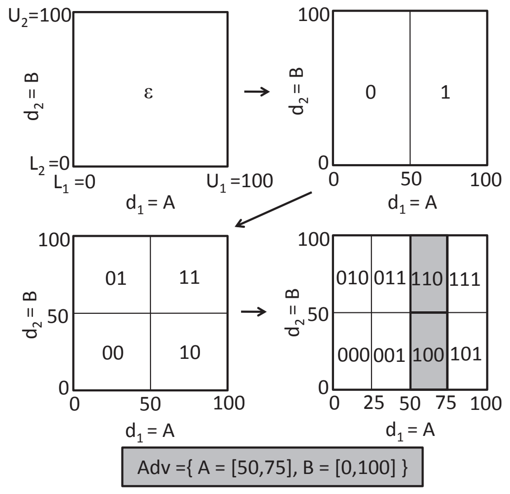


Figure 2: This figure illustrates for two attributes A and B the decomposition of the event-space in subspaces each associated with a distinct dz [14].

ments, and subscriptions. Any subspace can be identified by a binary string named dz-expression (in short dz). In particular, dz-expression fulfill following characteristics. 1) The shorter the dz the larger is the corresponding subspace in Ω. 2) A subspace represented by dz_i is *covered* by the subspace represented by dz_j, iff dz_j is a prefix of dz_i. In this case, we write $dz_j \succ dz_i$. 3) Two subspaces dz_i and dz_j are overlapping if either $dz_i \succ dz_j$ or $dz_j \succ dz_i$ holds and the overlap $dz_i \cap dz_j$ is identified by the longest of two dz. 4) For overlapping non identical subspaces dz_i and dz_j, the non overlapping part, say $dz_i - dz_j$, may need to be identified by multiple subspaces. For instance, the non overlapped part of $dz_i = 0$ w.r.t. $dz_j = 000$ contains subspaces 001, 010 and 011.

The advertisement/subscription can be composed of several dz, denoted as DZ. For instance, to approximate the advertisement in Figure 2 two dz are required, i.e., $DZ = \{110, 100\}$. The containment and overlap relationships between a pair of DZ can be defined w.r.t. set of dz-expressions represented by them.

Using the above relationships, an event e disseminated by a publisher p will comprise in its packet header fields a dz of maximum length. In order to deliver e to a subscriber s, the controller must have installed on each switch along the path (between p and s) a flow comprising a covering dz in a field. The subspace relationship of dz-expressions allows subscribers with overlapping subscriptions to share a common subpath(s) and therefore, bandwidth.

Incoming advertisements and subscriptions are processed by the controller in a sequence to avoid inconsistent updates to the network topology. In order to ensure connectivity between the distinct network partitions, a controller will need to react in addition to internal advertisement and subscription requests, also to external advertisements and subscriptions received via the border gateways (switches). The rate at which a controller receives external and internal requests is critical for the time until a subscription is available. Furthermore, with increasing number of advertisements and subscriptions the number of flows to be installed within each partition can grow.

The PLEROMA middleware addresses these issues along three components: The *Topology Reconfigurator* reacts upon dynamic internal advertisements/subscriptions. It computes changes in the network topology of a single partition and installs flows correspondingly. The *border-gate configuration component* of the PLEROMA middleware acts upon external advertisements/subscriptions. It treats border gateway switches as additional subscriber and publisher and ensures that events published in one particular network partition will be also forwarded to subscribers in other network partitions. Partitioning the network already helps to improve scalability since subscriptions (of which many may only be of relevance to a single partition) can be processed concurrently by multiple controllers. However, the number of flows can be further reduced by the *dimension-selection component* which determines for each network partition the most relevant attributes and limits the length of any installed dz.

3. TOPOLOGY RECONFIGURATION

In this section, we describe how reconfigurations of the network topology are performed. To this end, we need to maintain a dissemination structure which allows for bandwidth and latency efficient forwarding of packets. Moreover, changes to the network topology must be efficiently installed, to handle many subscription and advertisement requests.

3.1 Overview

For the design of a dissemination structure we consider as constraints latency efficiency, bandwidth usage, and cost efficiency to update the network topology. Clearly, the lowest latency is achieved if a controller establishes a shortest path for each publisher/subscriber pair. However, this severely limits the reuse in forwarding an event on common paths. Moreover, each new subscription or advertisement would trigger updates of the network topology to add paths between all relevant publishers and subscribers and therefore, impose a very high reconfiguration cost.

A common alternative—often taken by traditional broker-based systems [2, 8]—is to embed the paths between publishers and subscribers by means of filters in a single spanning tree. The spanning tree reflects low latency paths between any pair of publisher and subscriber. Since all paths between publishers and subscribers are embedded in the same tree, the number of times an event needs to be forwarded is significantly reduced. The reconfiguration cost is also limited to the edges in the spanning tree and is significantly reduced wherever subscriptions and advertisements overlap. Nevertheless, such a topology imposes limits on the capacity of forwarding events. While links in the core of the spanning tree are heavily utilized other links remain even idle. Moreover, with an increasing number of subscribers and publishers the path length may increase (because of the limited choice of edges in the spanning tree) imposing larger delays [27, 17].

The approach followed by the PLEROMA middleware is to maintain multiple independently configurable spanning trees. The intention is to i) better balance the load on the links of the network and this way offer a higher forwarding capacity in sending events, ii) reduce the length of the path by limiting the number of publishers and subscribers (i.e., organizing edges of the spanning tree only w.r.t. relevant publishers and subscribers) and this way reduce both the latency as well as the reconfiguration cost of a path. To this end, PLEROMA accomplishes the following: Depending on dynamic advertisements of the publishers, PLEROMA updates the set of spanning trees and determines to which spanning trees a publisher should forward its events (cf. Section 3.2). Furthermore, depending on dynamic subscriptions, PLEROMA installs flows with respect to a spanning tree to ensure published events will reach each subscriber on an embedded path (cf. Section 3.3).

3.2 Spanning trees

The controller maintains a set of spanning trees (in short *trees*), denoted by T. Each tree $t \in T$ is built to disseminate events covered by a set of subspaces and is overlapping with advertisements of some publishers. It logically interconnects all hosts and switches of a partition. A tree t is identified by the set of dz-expressions, $DZ(t)$, that span the overlap with advertisements of the publishers. We will ensure for all trees t and t' in T that $DZ(t) \cap DZ(t') = \emptyset$. Therefore, an event will be disseminated at most in a single tree. Let P_t denotes the set of publishers whose advertisements overlap with $DZ(t)$ of a tree t. Likewise, T_p denotes the set of trees with which the advertisement of a publisher p overlaps. Furthermore, $DZ^t(p)$ defines the part of the advertisement of a publisher p in P_t which overlaps with $DZ(t)$.

On the arrival of an advertisement $DZ(p)$ from a publisher p, the controller checks for each dz_i in $DZ(p)$ the trees in T to which the publisher can *join*, i.e., each tree $t \in T$ whose subspace set $DZ(t)$ overlaps with the subspace of dz_i. Moreover, the controller creates a separate tree t_n for the (fine granular) subspaces of dz_i which are not covered by the DZ of existing trees (cf. Algorithm 1, lines 2-15). In more detail, the controller checks the containment relation between dz_i and the dz comprised in DZ of existing trees, and performs one of the three actions: (1) If $dz_i \in DZ(p)$ is *covered by* DZ of one or more existing trees, then the publisher p joins the corresponding trees (for publishing events matching dz_i). For instance, a publisher p with a new advertisement $DZ(p) = \{11\}$ joins an existing tree with $DZ = \{1\}$. (2) If dz_i in $DZ(p)$ *covers* the DZ of one or more existing trees, then p joins the corresponding trees (for publishing events matching the covered subspaces of its dz_i) and the uncovered subspaces are added to the DZ of new tree t_n. For instance, a publisher p with a new advertisement $DZ(p) = \{0\}$ joins an existing tree with DZ containing 00 and uncovered part of $DZ(p)$, i.e., 01, is added to the $DZ(t_n)$ of a new tree t_n. (3) If no containment relation exists, then dz_i is added to the $DZ(t_n)$ of newly created tree t_n. The procedure *createTree* is used to build the tree t_n as a shortest path tree rooted at the publisher p[2](cf. Algorithm 1, line 14). With the controller having a complete view on the switch network, the building of a tree (e.g., t_n) is reduced to a simple graph problem. The procedure *adjustFlowMultSubs* identifies existing subscriptions in the system that match $DZ^{t_n}(p)$ and establishes routing paths along the edges of t_n between the subscribers (of those subscriptions) and the publisher p by adding flows to the switch network. The addition of flows to the switch network is detailed in Section 3.3.

The proposed strategy might result in the creation of large number of trees depending on the advertisements of the publishers. To avoid this problem, the controller merges existing trees if their number increases above a certain threshold. Merging can be easily performed by mapping DZ of trees (to be merged) to a smaller set of coarser subspaces (i.e., with shorter $|dz|$). For instance, two trees t_1 and t_2 with $DZ(t_1) = \{0000, 0010\}$ and $DZ(t_2) = \{0001, 0011\}$ respectively, can be merged in a single tree with $DZ = \{00\}$. Section 5 details further methods to avoid the problem associated with the number of trees.

Finally, it is worth noting that the proposed tree creation strategy can be further enhanced by taking into account the event traffic and available subscriptions in the systems. However, we opted for the simple version to ensure low reconfiguration overhead in terms of creation/merging of trees and thus low reconfiguration delay in the presence of dynamically arriving (un)subscriptions and (un)advertisements.

3.3 Maintenance of flow tables

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

3.3.1 Subscription requests

Subscription requests are handled similar to the advertisements, except for the fact that a newly arrived subscription cannot trigger the creation of new trees (cf. Algorithm 1, lines 16-25). More precisely, upon the reception of a subscription $DZ(s)$ from a subscriber s, the controller searches for a set of trees T_s such that the subspaces represented by the DZ of trees in T_s overlap with the subspaces of $DZ(s)$. If no overlap exists (i.e., T_s is empty), then the subscription is simply stored at the controller. The stored subscriptions are again checked for overlap, whenever DZ of an existing tree is changed or a new tree is created as a result of an arrival of a new advertisement (cf. Algorithm 1, lines 9, 15).

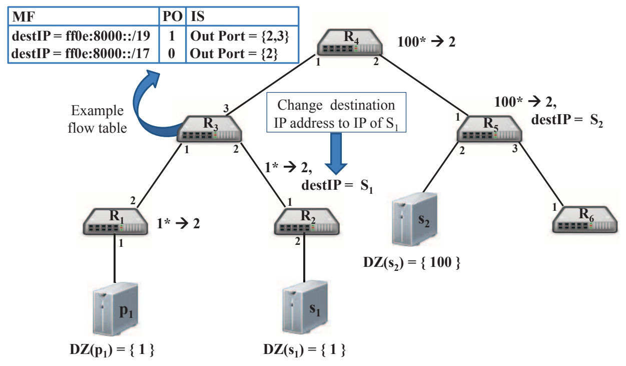
Nevertheless, if T_s is not empty (i.e, overlap exists), then the controller adds subscriber s to each t in T_s by establishing paths between the subscriber s and all publishers with overlapping advertisements in t. Each path between a publisher p and a subscriber s on a tree t only forwards the events matching the subspaces overlapped between $DZ^t(s)$ and $DZ^t(p)$. This way false positives are avoided.

As a first step, the controller calculates the *route* between the subscriber s and each relevant publisher p (in P_t) on the tree t. A route consists of a sequence of physical switches (denoted as R) on which flows need to be established along with the out ports (denoted as oP) through which a matching event should be forwarded so that connectivity is achieved between the publisher p and the subscriber s, i.e., $\langle p, s, t\rangle = \{(R_i, oP_i), \ldots (R_j, oP_j)\}$. Once the route is calculated, the controller establishes the path by inserting (or modifying) flows on the switches along the route between the publisher p and the subscriber s. The flows ensure only the events matching the overlapped subspaces (i.e., $DZ^t(s) \cap DZ^t(p)$) are forwarded on the path. The process of establishing paths along the switch network is detailed in the next section.

3.3.2 Flow installation

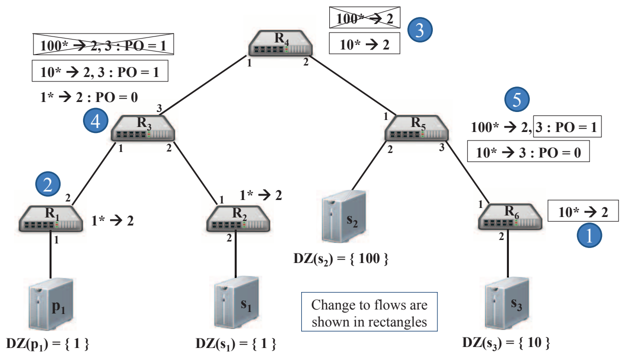
The installation of flows on the switches requires to specify the *match field (MF)*, *instruction set(IS)* and priority order

[2]We opted for shortest path tree in our implementation, however, other tree creation algorithms such as minimum spanning tree etc., can also be employed without any modification to the propose approach.



MF	PO	IS
destIP = ff0e:8000::/19	1	Out Port = {2,3}
destIP = ff0e:8000::/17	0	Out Port = {2}

Figure 3: Forwarding in the switch network



~~$100^* \rightarrow 2$~~
3
$10^* \rightarrow 2$
R_4
~~$100^* \rightarrow 2, 3 : PO = 1$~~
$10^* \rightarrow 2, 3 : PO = 1$
$1^* \rightarrow 2 : PO = 0$
4
R_3
5
R_5
$100^* \rightarrow 2, 3 : PO = 1$
$10^* \rightarrow 3 : PO = 0$
2
R_1
$1^* \rightarrow 2$
R_2
$1^* \rightarrow 2$
s_2
R_6
$10^* \rightarrow 2$
1
$DZ(s_2) = \{ 100 \}$
p_1
s_1
Change to flows are shown in rectangles
s_3
$DZ(p_1) = \{ 1 \}$
$DZ(s_1) = \{ 1 \}$
$DZ(s_3) = \{ 10 \}$

Figure 4: Flow maintenance on the arrival of s_3

(PO) of a flow [20]. The matching field defines the header information against which packets are matched. Recall that PLEROMA uses for interoperability with other services the IP-multicast range to embed dz-expressions. For instance, subspaces with $dz = 101101$ and $dz = 101$ are converted to IPv6 multicast addresses `ff0e:b400:*` and `ff0e:a000:*` respectively. The symbol $*$ is used to represent standard wildcard/masking operations, which are supported by hardware switches for matching IP addresses using Class-less Interdomain Routing (CIDR). Therefore, an event $dz = 101101$ can be matched against a flow with $dz = 101$ in hardware switch during forwarding, i.e., `ff0e:a000::/19` $\succ$ `ff0e:b400::/22`.

Furthermore, in the instruction set the outgoing ports are specified, ensuring that a matching packet (i.e., an event) can be forwarded to multiple destinations in the spanning tree. Other than forwarding on out ports, another action which can be specified in IS is to change the destination IP field of the matching packets (i.e., events) on the terminal switches (from dz) to the IP of the hosts on which subscribers are running, as depicted by switches R_2 and R_5 in Figure 3. Finally the priority order, needs to be defined to decide on the order in which flows will be applied to a packet. For example, an incoming event ($dz = 1001$) on switch R_3 matches multiple flows with $dz = 1$ and $dz = 100$. However, the switch only follows the instructions of the first match. Therefore, to ensure proper forwarding the flow installation gives higher priority to the flows with longer dz. In Figure 3, priority order on R_3 ensures that all packets matching flow with $dz = 100$ are forwarded to both switches (R_2 and R_4), however, packets matching flow with $dz = 1$ but not with $dz = 100$ are only forwarded to R_2.

To describe the maintenance of flows in the presence of dynamic (un)subscriptions, we first define the containment relation between flows w.r.t. a single switch. A flow fl_1 covers (or contains) another flow fl_2, denoted by $fl_1 \succ fl_2$, iff the following two conditions hold: (i) the dz associated with the destination IP address in the match field of fl_2 is covered by the dz of fl_1, and (ii) the out ports to which a packet matching fl_2 is forwarded are subset of those specified in the IS of fl_1. Likewise, a partial containment relation ($\gtrapprox$) can be defined between flows of a switch (or flows to be installed on a switch). A flow fl_1 partially covers (or contains) another flow fl_2, denoted by $fl_1 \gtrapprox fl_2$, if dz associated with the match field of fl_1 covers dz of fl_2, but not all the out ports used for forwarding packets matching fl_2 are listed in the IS of fl_1.

The procedure *flowAddition* is used by the controller to set up flows on the switches along the route $\langle p, s, t\rangle$ between the publisher p and the subscriber s (cf. Algorithm 1, lines 31 - 51). The dz used for creating the match field of the new flows (to be added in the switch network) is determined from the overlap between $DZ^t(s)$ and $DZ^t(p)$, as mentioned earlier.

In more detail, the controller iteratively checks the existing entries in the flow tables of each switch R_i along the route $\langle p, s, t\rangle$ and determines whether to add a new flow fl_n or to modify (or delete) existing flows. The following cases drive the process of flow addition and modification at a particular switch R_i. The cases are explained w.r.t. the changes to the flow tables of the switches in Figure 4 on the arrival of new subscriber s_3 with subscription $DZ(s_3) = \{10\}$. (1) If the flows are not currently installed on a switch, then the new flow fl_n is simply added to the flow table of that switch, e.g., a new flow with $dz = 10$ is added to R_6 in Figure 4. (2) If an existing flow fl_c already covers the new flow fl_n to be installed on the switch (i.e., $fl_c \succ fl_n$), then no action is performed, e.g., no new flow is added to the switch R_1 in Figure 4. (3) If an existing flow fl_c is covered by the new flow fl_n, then the new flow fl_n is added and fl_c is deleted from the flow table as it is no longer needed, e.g., in Figure 4 existing flows associated with $dz = 100$ are replaced by new flows with $dz = 10$ on R_3 and R_4. (4) If the new flow fl_n is partially covered by an existing flow fl_c (i.e., $fl_c \gtrapprox fl_n$), then fl_n should be added with high priority and should include the out ports in the IS of fl_c, as depicted by R_3 in Figure 4. (5) Finally, if the existing flow fl_c is partially covered by the new flow fl_n, then besides adding fl_n to the flow table, the existing flow fl_c should be updated to include out ports used by fl_n and to hold higher priority than fl_n, e.g., in Figure 4 an additional out port (i.e., $oP = 3$) and a higher priority order is assigned to an existing flow on R_5 with $dz = 100$.

3.3.3 Unsubscriptions

Until now we have focused on the maintenance of flow tables w.r.t. advertisements and subscriptions in the system. Now, we will briefly discuss the handling of unsubscriptions. On the arrival of an unsubscription, the subscriber s is removed from all the trees T_s associated with the corresponding subscription. This is accomplished by removing previously established paths between s and all publishers with overlapping advertisements (i.e., $DZ^t(s) \cap DZ^t(p) \neq \emptyset$) on each tree t in T_s. To remove a path on a tree t, the flows are either deleted or downgraded depending upon other sub-

Algorithm 1 Publish/Subscribe maintenance at a single controller

```
1:  upon event Receive(ADV, $p$, $DZ(p)$) do
2:  for all $dz_i \in DZ(p)$ do
3:    dzNewTree = $\emptyset$
4:    $T_p = \{t \in T \wedge \exists dz_j \in DZ(t) : dz_i \succ dz_j \vee dz_j \succ dz_i\}$ // Set of trees with overlapping subspaces with $dz_i$
5:    if $T_p \neq \emptyset$ then
6:      for all $t \in T_p$ do
7:        $DZ^t(p) = DZ(t) \cap dz_i$ // Overlapped subspaces
8:        joinTree(p, $DZ^t(p)$, $t$)
9:        addFlowMultSub(p, $DZ^t(p)$, $t$)
10:     uncoveredSubspace = $dz_i - \bigcup_{t \in treeSet} DZ^t(p)$
11:     dzNewTree = uncoveredSubspace
12:   else
13:     dzNewTree = $dz_i$
14:   $t_n$ = createTree(p, dzNewTree)
15:   addFlowMultSub(p, dzNewTree, $t_n$)

16: upon event Receive(SUB, $s$, $DZ(s)$) do
17: for all $dz_i \in DZ(s)$ do
18:   $T_s = \{t \in T \wedge \exists dz_j \in DZ(t) : dz_i \succ dz_j \vee dz_j \succ dz_i\}$
19:   if $T_s \neq \emptyset$ then
20:     for all $t \in T_s$ do
21:       $DZ^t(s) = dz_i \cap DZ(t)$ // Overlapped subspaces
22:       pubSet = $\{p \in P_t \wedge \exists dz_j \in DZ^t(p) : dz_i \succ dz_j \vee dz_j \succ dz_i\}$ // Publishers with overlapping $DZ^t(p)$
23:       for all $p \in$ pubSet do
24:         overlapWithPub = $dz_i \cap DZ^t(p)$
25:         flowAddition(overlapWithPub, $\langle p, s, t \rangle$, $t$)

26: procedure addFlowMultSub( $p$, $DZ$, $t$) do
27: for all $s \in S$ do
28:   if $DZ \cap DZ^t(s) \neq \emptyset$ then
29:     overlapWithPub = $DZ \cap DZ^t(s)$
30:     flowAddition(overlapWithPub, $\langle p, s, t \rangle$, $t$)

31: procedure flowAddition( $dz$, $\langle p, s, t \rangle$, $t$) do
32:  destIP = $(binary(\texttt{ff0e:b400}) \& (dz \ll 112 - |dz|)) \backslash 16 + |dz|$
33:  for all $r_i \in \langle p, s, t \rangle$ do
34:    Flow $fl_n = MF \cup IS \cup PO$
35:    $fl_n.MF$ = destIP
36:    $fl_n.PO$ = default value
37:    $fl_n.IS.oP = \{r_i.oP_i\}$
38:    curFlow = getCurrentFlowsFromSwitch($r_i.R_i$)
39:    if $r_i$ is last entry in $\langle p, s, t \rangle$ then
40:      $fl_n.IS$.set-destIP = $s$.IP
41:    if curFlow $\neq \emptyset \wedge \neg(\exists fl_c \in$ curFlow : $fl_c \succ fl_n)$ then
       // Cases 3 - 4: None of the curFlow fully covers $fl_n$
42:      for all $fl_c \in$ curFlow : $fl_n \succ fl_c$ do // Case 3
43:        deleteFlowFromSwitch($fl_c$, $r_i.R_i$)
44:      for all $fl_c \in$ curFlow : $fl_c \gtrapprox fl_n$ do // Case 4
45:        $fl_n.IS.oP = fl_n.IS.oP \cup fl_c.IS.oP$
46:        increasePriority($fl_n.PO$)
47:      for all $fl_c \in$ curFlow : $fl_n \gtrapprox fl_c$ do // Case 5
48:        $fl_c.IS.oP = fl_c.IS.oP \cup fl_n.IS.oP$
49:        increasePriority($fl_c.PO$)
50:        modifyFlowOnSwitch($fl_c$, $r_i.R_i$)
51:    addFlowOnSwitch($fl_n$, $r_i.R_i$)
```

scribers reachable (w.r.t. their relevant publishers) via a particular switch or in other words depending upon the already established paths passing through a particular switch. For example, on the arrival of an unsubscription from s_3 in Figure 4, the flow with $dz = 10$ is deleted from the flow table of R_6 as no other subscriber is reachable w.r.t. p_1 via R_6. However, the flows installed on switches R_3, R_4 and R_5 have to be downgraded from $dz = 10$ to use $dz = 100$ (in their match fields) because the path from p_1 to subscriber s_2 with $DZ(s_2) = \{100\}$ passes through these switches.

4. INTEROPERABILITY OF PARTITIONS

So far the mechanisms allow only for the independent configuration of a single partition. In this section, we detail how PLEROMA ensures also connectivity between subscribers and publishers of distinct partitions. Our mechanisms preserve the decentralized control in which each controller performs independent updates to its own switch network (i.e., own partition) and therefore, ensures scalability in performing reconfiguration. The design goal of PLEROMA is to minimize the coordination overhead between partitions and exchange as little information as possible between neighboring partitions. To this end the controller needs to perform two main tasks. 1) The controller has to discover adjoined switch networks and establish communication with the corresponding controllers (cf. Section 4.1). 2) The controller has to install inter-partition paths that provide connectivity between publishers and subscribers in different partitions (cf. Section 4.2).

4.1 Communication across partitions

In order to set up routing paths between publishers and subscribers residing in different switch networks, additional knowledge is required such as i) interconnectivity between network partitions, ii) identity of controllers assigned to different switch networks (i.e., partitions), and iii) local subscriptions and advertisements managed by each switch network. For instance, in Figure 5 an inter-switch routing path between an incoming subscriber s_1 (at c_3) and an existing publisher p_1 can only be established if the controller c_3 knows that i) a publisher p_1 with $DZ(p_1)$ is residing in partition N_{c_1}, and ii) partition N_{c_1} is only reachable via N_{c_2}.

In our approach, a controller (e.g., c_1 in Figure 5) only maintains knowledge about the directly reachable neighboring switch networks (e.g., N_{c_2}). Information required to set up inter-network paths (such as advertisements or subscriptions) is only shared with the neighboring switch networks (e.g., N_{c_2}), which in turn may choose to share the information with their respective neighbors (e.g., N_{c_3}) and in this way the routing paths spanning multiple switch networks can be established. In realizing a decentralized approach, two important issues have to be resolved: First, the controller c, which is only aware of its own N_c, needs to discover reachable switch networks. Second, the controller c needs mechanisms to contact and share information with the controllers of the discovered switch networks without knowing their actual identities.

Discovery of adjoining switch networks: A controller c can discover each of its adjoining switch networks N_{c_n} by identifying border switch/es connecting N_c and N_{c_n}. Here, we briefly describe an OSI layer 3 (i.e., network layer) mechanism to automatically detect the border switches of adjoining switch networks in the same connectivity domain. More precisely, a controller (such as Floodlight [7]) discovers the topology associated with it by sending Link Layer Discovery Protocol (LLDP) packets to all switches in its own local switch network. On receiving LLDP packet directly from the controller, a switch R_i forwards the packet over all its ports to other connected switches. However, in case the LLDP packet is received from another switch it is handed over to the controller which accordingly creates a connection between the two switches (i.e., target switch of the LLDP packet and the switch forwarding the packet to the controller) in its discovery table. In general, LLDP packets generated by one controller is considered invalid by

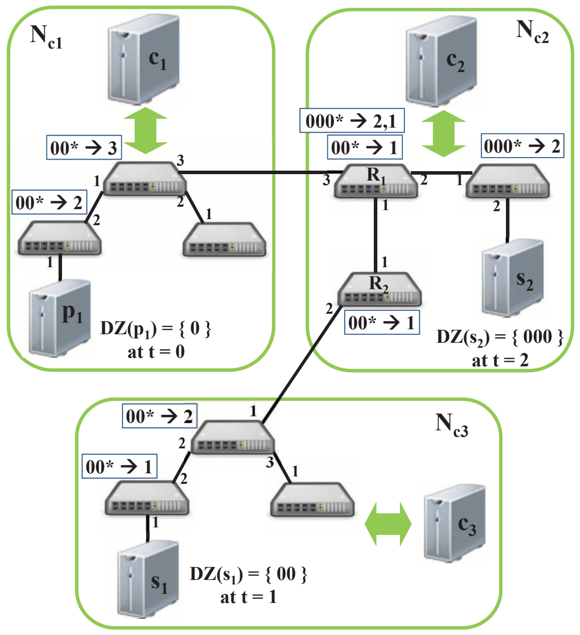


Figure 5: Publish/Subscribe across multiple independently controlled switch networks.

the other controllers. However, we extend this mechanism by identifying the LLDP packets received from other controllers and noting down the switch-port tuples at which they were received. Clearly, according to this mechanism, a switch can receive a packet from another controller only if it is directly connected to a switch of the adjoining network.

Communication between controllers: For a controller c to forward a packet to a controller c_n of a neighboring partition, c looks up the switch-port tuple (R, oP) for the partition N_{c_n}. It instructs the border switch R to forward the packet via port oP. The destination IP of the packet is set to IP_{fix}. Therefore, the border switch of N_{c_n} will forward the packet to c_n. In Figure 5, the switches R_1 and R_2 connected to the network N_{c_2} serve as border gateways (or switches) with their ports 3 and 2 linked to adjoining networks N_{c_1} and N_{c_3} respectively. The controller c_2 can communicate with c_1 by sending message through switch-port tuple $(R_1, 3)$ with destination IP set to IP_{fix}. It is worth mentioning that the proposed strategy ensures interoperability between heterogeneous switch networks. A neighboring switch network may be organized (and optimized) in a different manner and may be managed by multiple controllers (e.g., for load balancing).

4.2 Inter-partition routing paths

In this section, we detail how to establish inter-partition paths building on the mechanisms in Section 4.1. In general, to set up paths, advertisements are forwarded across all partitions of the SDN network. The subscription requests subsequently follow the reverse path previously taken by overlapping advertisements.

More precisely, a controller c on receiving an advertisement request Adv_e from a host (connected to a switch in N_c) forwards the request to the controllers of its adjoining networks using the border switches. The external request Adv_e (i.e., request from an adjoining network) is perceived by a (remote) controller c_r as arriving from the virtual host connected to its border switch. Requests from virtual hosts (i.e., external requests) are treated in the same way as those from the local hosts. Moreover, the external advertisement request Adv_e is forwarded by c_r to all other adjoining switch networks, except the one from which the (external) request is received, to ensure advertisement is known to all partitions of the SDN network. For example, in Figure 5 on receiving an external advertisement request ($DZ(p_1) = \{0\}$ at time $= 0$) from N_{c_1}, the controller c_2 i) identifies it as arriving from a virtual host connected to switch R_1 at port 3, ii) creates/merges trees associated with $dz = 0$ in N_{c_2}, and iii) finally forwards the external request to the adjoining network N_{c_3}.

It is important to mention that forwarding each advertisement to all partitions of the SDN network increases the inter-partition control traffic and induces extra load on controllers (in terms of processing and memory resources). Therefore, we employed covering-based forwarding approach, where an advertisement request is only shared with the controller of a neighbouring switch network if it is not covered by previously forwarded requests.

Likewise, on receiving a subscription request matching an external advertisement, a controller c establishes the path in its own N_{c_3} by modifying flow tables of switches along the route between the subscriber and the virtual host of the external advertisement. Moreover, the controller c forwards the subscription to the controller of the adjoining network from where the external advertisement arrives. Similar to advertisements, a subscription request is only forwarded to an adjoining network if it is not covered by the previously forwarded subscriptions (to save inter-switch network control traffic). For example, in Figure 5 the controller c_3 forwards an incoming subscriptions ($DZ(s_1) = \{00\}$ at time $= 1$) to c_2 for setting up paths in N_{c_2}. The request is in turn forwarded by controller c_2 to c_1. However, subsequent subscription request from s_2 (i.e., $DZ(s_2) = \{000\}$ at time $= 2$) is not forwarded by c_2 to c_1 because it is already covered by the subscription of s_1.

5. DIMENSION SELECTION

In PLEROMA, the length of dz-expressions required to accurately represent the subspaces mapped by subscriptions (advertisements or events) increases linearly with the number of attributes (or dimensions) in the system. Recall, in practice the length of dz is limited by the range of IPv6 (or IPv4) multicast address reserved for the publish/subscribe. Similarly, for an event space with many attributes, the number of dz (i.e., subspaces) for an accurate subscription (or advertisement) representation may be very high and may produce flow tables with large number of entries.

PLEROMA addresses the above limitations by performing spatial indexing only on a small subset of dimensions, denoted as Ω_D. The dimensions in the set Ω_D are selected according to their ability to avoid dissemination of unnecessary messages during in-network filtering. More precisely, the ability of a dimension d to reduce false positives mainly depends on two factors, i.e., selectivity of subscriptions and distribution of events along that dimension. It is worth mentioning, that both the factors play an equally important role during the selection of dimensions. For instance, selection of a dimension solely based on high selectivity of subscriptions may not be beneficial if the event traffic along the dimension is always consumed by the same subscriptions. Therefore, PLEROMA utilizes variability (or in other words variance) in the set of subscriptions matched by the events (according to current event traffic) as the main criteria for the selection of dimensions to perform in-network filtering of events.

In more detail, let E^t be the set of last ψ events published in the system before a given time t. Let S_e^d denote the set of subscriptions matched by an event $e \in E^t$ along a dimension d. The number of subscriptions matched by the events in E^t along all dimensions can be represented by a matrix $W \in \mathbb{R}^{|\Omega| \times |E^t|}$, such that $w_{i,j} = |S_j^i|$. Given the matrix W, the dimensions with high variability in the set of subscriptions matched by events in E^t can be easily selected by calculating the variance along each row (of W). However, distribution of events and subscriptions may be correlated along different dimensions, e.g., subscriptions may consume similar event traffic along multiple dimensions. Selection of highly correlated dimensions is not desirable because correlated dimensions introduce redundancy (i.e., additional overhead in terms of the length of dz-expressions and number of flows) without much benefit towards reduction of unnecessary event traffic in the system.

Without going into much mathematical details, we highlight the main steps required to select the dimensions (in Ω_D) that can perform in-network filtering with low redundancy. Initially, we calculate the covariance matrix, i.e., $\mathbb{C} = \overline{W} \times \overline{W}^T$, where $\overline{W}$ is a centred matrix obtained by subtracting the mean of the matrix W from its columns. The covariance matrix captures the correlations between all possible pairs of dimensions w.r.t. event traffic consumed by the subscriptions along each dimension. Afterwards, we spectrally analyse the covariance matrix by means of eigendecomposition, i.e., $\mathbb{C} = Q\Lambda Q^T$, where $\Lambda = \{\lambda_i, \ldots, \lambda_{|\Omega|}\}$ is diagonal matrix of eigenvalues and $Q = \{q_i \ldots, q_{|\Omega|}\}$ is the matrix whose columns are orthogonal eigenvectors of $\mathbb{C}$. The eigendecomposition transforms the original dimensions(in Ω) into an orthogonal basis (represented by Q) such that the eigenvectors (of Q) represent the dimensions along which the variance in W is maximized (i.e., variance in the event traffic matched by subscriptions is maximized). However, dimensions in the orthogonal basis (i.e., eigenvectors) do not directly correspond to the original dimensions. Therefore, we need a method that can utilize the variability information represented by eigenvectors to select the original dimensions. We employed the method proposed by [18] for this purpose. In particular, an eigenvector q with largest eigenvalue represents the dimension (in the orthogonal basis) along which variance is maximized and thus this eigenvector q is used to rank the original dimensions. In more detail, a higher absolute value of i^{th} coefficient of q (i.e., q_i) indicates that the dimension d_i is more important to be used for filtering. Thus the dimensions (in the original space) that correspond to the first k coefficients with higher magnitude are selected for filtering. The number of dimensions (i.e., k) to use for filtering can be determined by analysing the magnitude of coefficients in q. More precisely, given the eigenvector q (i.e., with the highest eigenvalue) sorted according to the magnitude of its coefficients, k can be selected such that $\frac{\sum_i^k q_i}{\sum_i^{|\Omega|} q}$ is above an administrator defined threshold.

Once the dimensions in Ω_D are selected, the controller i) generates new DZ for existing subscriptions and advertisements, ii) installs flows between publishers and subscribers in the switch network w.r.t. to newly created DZ, and iii) finally notifies publishers about the selected dimensions to ensure future events are published with correct dz. In practice, subscriptions/advertisements as well as event traffic change dynamically and therefore, over time the previously selected dimensions may become suboptimal. In order to adapt to the changes, a controller periodically collects information about the events disseminated (in the recent time window) by the publishers and repeats the dimension selection process.

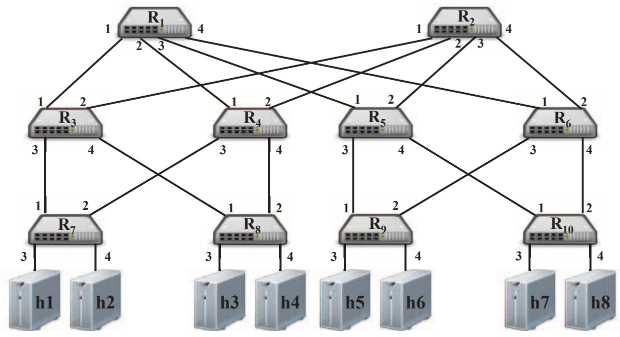


Figure 6: Testbed topology.

6. PERFORMANCE EVALUATIONS

This section is dedicated to an analysis of the design and implementation of the proposed PLEROMA middleware. A series of experiments are conducted to understand the effects of the design on performance metrics such as (i) end-to-end delay for event dissemination, (ii) throughput w.r.t. number of published events, (iii) bandwidth efficiency in terms of false positives w.r.t. length of dz and number of flows, (v) reconfiguration delay in the presence of dynamics, and (iv) control overhead in managing multiple control partitions.

6.1 Experimental setup

The PLEROMA middleware has been evaluated under two test environments. The majority of the experiments has been conducted on a SDN testbed [12] consisting of commodity PC hardware and virtualization technologies as used in datacenters. Figure 7 depicts the hierarchical fat-tree topology of the testbed which consists of a cluster of hosts (running on commodity rack PCs). Some of these hosts act as OpenFlow switches (switch hosts) with four physical ports by executing a production-grade software switch (Open vSwitch [21]) attached to the 4-port NIC, i.e., switches $R1$ to $R10$. The other hosts act as end systems (end hosts) by executing virtual machines on two physical machines, i.e., h_1 to h_8. The end hosts implement the functionality to publish and subscribe events. The SDN controller is hosted on a front end machine which manages the virtual machines and network topology. Besides the testbed, experiments associated with multiple controllers have been conducted on a prominent tool for emulating software-defined networks, namely, Mininet [16]. Mininet uses the concept of OS-level lightweight virtualization for network emulation that allows users to experiment with various topologies and application traffic. We use Mininet to experiment with 20 switches on two different topologies, a fat-tree topology (as describe above) and a ring topology. In the ring topology, each switch

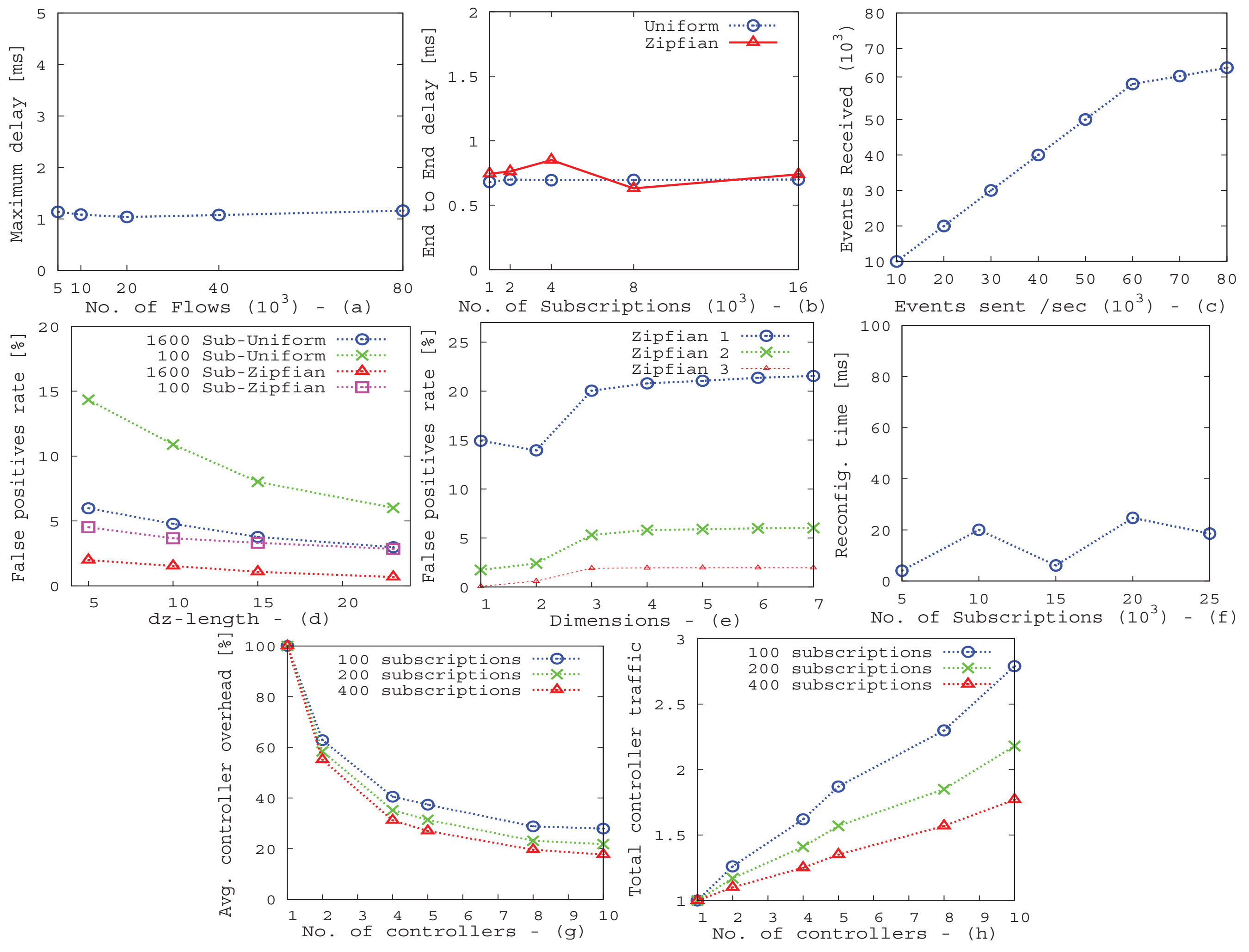


Figure 7: Performance evaluations

has an end host connected to it which may play the role of a publisher or a subscriber.

The content-based schema contains up to 10 attributes, where the domain of each attribute vary in the range $[0, 1023]$. Experiments are performed on two different models for the distributions of subscriptions and events. The uniform model generates random subscriptions and events independent from each other. The interest popularity model chooses 7 hotspot regions around which subscriptions/events are generated using the widely used zipfian distribution.

6.2 End-to-end delay

The first set of experiments study the delay characteristics of the aforementioned SDN testbed (with fat-tree toplogy) w.r.t. various metrics. More specifically, we first analyse the effect of flow table entries on the end-to-end delay between a publisher and a subscriber connected via the longest path in the topology. During the experiment, the flow table size of each switch along the longest path is varied between 5000 to $80,000$ entries and $10,000$ UDP packets containing random events (i.e., events matching different entries in the flow table) are sent from the publisher at a constant rate. The size of each packet is up to 64 bytes depending upon the length of dz. Figure 7(a) shows that the average delay calculated at the subscriber remains almost constant for different flow table sizes. This clearly indicates that the number of entries in the flow table has very little impact on the processing delay at the switches.

We next analyse the end-to-end delay to deliver an event from a publisher to all interested subscribers w.r.t. the number of subscriptions in the system. For the experiment, up to $16,000$ subscriptions are generated using the two above mentioned distributions (i.e., uniform and zipfian) and divided among different end hosts. Furthermore, end-to-end delay measurements are averaged across $10,000$ events published in the system at a constant rate. Figure 7(b) indicates that the number of subscriptions does not significantly impact end-to-end delay. It is worth noting that for uniform distribution the generated subscription set is randomly divided among all end hosts (i.e., subscribers). As a result, the possibility of every end host receiving at least some events is extremely high, resulting in a near constant end-to-end delay in the system. However, in case of zipfian distribution, each end host is assigned a hotspot and subscribes for subspaces corresponding to its respective hotspot only. With events also following a zipfian distribution, it may so happen that one or more end hosts do not receive any events. As a result, the average end-to-end delay in the system may vary (albeit slightly) with different subscription workloads, as indicated in Figure 7(b).

We would like to stress that using virtual switches does not invalidate our results, but rather gives very conservative performance bounds. For instance, recently performed micro-benchmarks to evaluate the forwarding delay using a NEC hardware switch validate our results and indicate additional gain that is expected from using hardware switch [5].

6.3 Throughput

In this experiment, we evaluate the ability of the switch network and the end hosts to handle high incoming event rates. Subscriptions are generated following zipfian distribution and are divided among 4 end hosts. A single publisher is used to send events at varying rates. According to Figure 7(c), beyond a certain event rate, not all the events are received by the end hosts. However, results indicate that the switch network is able to successfully forward every event to the end hosts. The drop in received events is due to the processing limitations at the end hosts. In another set of experiments, we determine that by using faster machines the throughput can be increased upto $170,000$ events per second. However, in either case the bottleneck is on the side of end hosts.

6.4 Bandwidth efficiency w.r.t. false positives

False positives are those events which are delivered to a subscriber without it being interested in receiving them. We define false positive rate (FPR) as a percentage of the number of unnecessary events received to the total number of events received. Clearly, false positives are undesirable and the aim of any publish/subscribe system is to keep them to a minimum. We observe that longer the dz, the lesser are the false positives. This follows from the fact that as the length of the dz increases, the granularity of the subspaces (assigned to advertisements, subscriptions and events) also increases and hence the false positives delivered to a subscriber decrease. Figure 7(d) shows the variation of false positive rate with the length of dz for different number of subscriptions for both uniform as well as zipfian distribution. As seen in the figure, with increase in the length of the dz the false positives decrease for both distributions. The variation of false positives is also noticeable with number of subscriptions. This is justifiable as a large number of subscriptions divided randomly among end hosts almost represents the near-ideal case. As we only have a limited number of bits, say L_{dz}, for the representation of dz in an IP multicast address, subscriptions and events which differ in dz only after the L_{dz} cannot be differentiated. Thus, for less number of subscriptions, an event e might fit into the filtering criteria of a subspace – which does not actually contain (or cover) the event e – due to dz truncation and is counted as a false positive. But for large number of subscriptions, the same event e might have been contained in (or covered by) the subspace subscribed by another subscription and hence is no longer counted as a false positive.

It is quite clear from the above discussion that the L_{dz} constraint in the dz representation of subscriptions and events severely impacts the occurrence of false positives. If the number of dimensions in the event space is high, the dz constituting subscription subspaces can be very long and difficult to be accommodated in limited number of bits (cf. Section 5). For this reason we introduced the concept of dimension selection in this paper. To portray the effectiveness of this concept, we conducted a set of experiments using an event space with 7 dimensions. The subscriptions were generated using zipfian distribution and divided equally among the end hosts. Events for the experiments are also generated primarily using zipfian distribution. To model varying selectivity (across different dimensions of event space), we impose restrictions on the degree of variance of event values along certain dimensions. Depending on the restrictions, three types of zipfian workloads are generated and evaluated. Figure 7(e) presents the behavior of FPR on dimension selection for the generated zipfian workloads. The figure clearly indicates that reduction of dimensions proves to be an effective way for decreasing false positives.

6.5 Reconfiguration delay

This experiment measures the average reconfiguration time required by a controller on the arrival of a new subscription, after a particular number of subscriptions has already been deployed in the system. Figure 7(f) shows that the reconfiguration delay varies significantly with the initial number of subscriptions installed in the system and it is difficult to draw a relationship between the two. Results indicate that this delay depends on the number of flows modified/added on arrival of a subscription which itself is influenced by various factors such as position of the subscriber in the network, existing subscription workload etc. However, it is quite clear from the results that even with $25,000$ already deployed subscriptions in the system, the controller is capable of processing around 54 subscriptions/second. This implies that the proposed system scales fairly well in the presence of dynamically arriving subscription and unsubscription requests.

6.6 Control overhead

A single controller processes one subscription request at a time (to ensure consistency of flow tables in the switch network) and this behavior can be further improved, as discussed earlier, by introducing multiple controllers capable of processing simultaneous requests in disjoint network partitions. In the distributed setting, there are mainly two types of control messages received by a controller. These are messages generated by end hosts (i.e., internal advertisements and subscriptions) and those generated by controllers (i.e., external advertisements and subscriptions) to share control information with others (i.e., controllers). Even though this implies that the generated (control) traffic increases, analysis shows that the average overhead at each controller may reduce significantly. Earlier, every end host request would be sent to the only controller available in the system. But now, end host requests are primarily sent only to the local controller which then takes a decision on sharing it with other controllers. So, our next set of evaluations study the behavior of average controller overhead and total control traffic with increased partitioning. We use uniform distribution and the subscriptions are randomly distributed between the end hosts of the network. Consequently, the number of requests received per controller for different number of controller instances leading to different network configurations has been measured.

Figure 7(g) presents normalized graphs depicting average controller overhead when the number of controller instances (network partitions) is gradually increased from 1 to 10. For each subscription workload, the average controller overhead reduces with increasing number of controllers. The normalization has been done to compare the benefits of partitioning when different number of subscriptions is used. It is visible from the figure that if the number of subscriptions is significantly increased, the benefit of partitioning increases as well. This follows from the fact that with more number of subscriptions, the probability of a match at the local controller also increases. This means that gradually lesser number of subscriptions needs to be shared with remote controllers because of covering-based routing. For the same reason, in Figure 7(h), which depicts behavior of total control traffic with increased partitioning, the comparative increase in control traffic for 400 subscriptions is less than 200 subscriptions which in turn is less than 100 subscriptions.

7. RELATED WORK

In the past decade, many middleware implementations for content-based publish/subscribe have been proposed with scalability as the main design criteria [2, 8, 19, 25]. Subscription summarization – using techniques such as subscription covering [2] and subscription merging [19] – has been identified as a promising method to achieve scalability. Using subscription summaries, events are filtered from disseminating to the parts of the broker network that do not host interested subscribers. Similarly, forwarding of new subscriptions is only restricted to the brokers which previously do not receive subsuming (or covering) subscription summaries. Clearly, subscription summarization is beneficial to reduce unnecessary message overhead in the broker network. However, the time until a new subscriber starts receiving events may be arbitrarily high, because the subscription request has to be forwarded and processed by a series of brokers. Moreover, maintenance of subscription summaries is very expensive in the presence of dynamic changes to the topology of the borker network [4]. In literature several solutions have been proposed to reduce the overhead associated with the maintenance of subscription summaries and efficient handling of new subscriptions [1, 10, 4]. Jayaram et al. [10] introduce the concept of parametric subscription to efficiently handle the subscriptions which are continuously changing w.r.t. certain parameters such as location. Kyra [1] partitions the publish/subscribe broker network into smaller routing networks such that event matching and subscription maintenance overhead is balanced between the networks. However, the partitions are frequently recalculated to cope with the changes in the workload (e.g., event traffic) as well as dynamics of the system (e.g., failure or arrival of a broker).

A common drawback of most of the existing publish/subscribe systems is their dependence on the application layer mechanisms to optimize publish/subscribe operations. For instance, event routing on a broker network that is organized oblivious to the underlying physical network (in short underlay), may result in higher bandwidth utilization (irrespective of the use of subscription summarization and event filtering mechanisms) and higher end-to-end delays, since multiple logical links in the broker network may share the same physical links [27]. Only a few systems explicitly take into account the properties of the underlying network and its topology to organize publish/subscribe broker network [9, 17, 27, 6]. Although, such systems bear significant cost, it is still hard to accurately infer advanced underlay properties such as the current link utilization based on observations on end systems (such as brokers). In the past, IP multicast has been proposed to distribute events between the clusters (or groups) of subscribers and publishers. Clearly, IP multicast overcomes many drawbacks of application layer by routing events on the network layer [23, 26]. However, IP multicast is very expensive in the presence of frequently changing subscriptions and event traffic, mainly because clusters have to be recalculated to ensure minimal false positives. The recent advent of new networking technologies, such as SDN and NetFPGA, has raised some research efforts towards realizing publish/subscribe middleware that can support event filtering and routing within the network. LIPSIN [11] uses bloom filters in data packets to enable efficient multicast of events on the network layer. However, the expressiveness of LIPSIN is limited to the topic-based publish/subscribe. Zhang et al. [28] address impact of SDN on future design of publish/subscribe middleware and describe realization of logically centralized publish/subscribe controller in a distributed manner. Koldehofe et al. [14] present reference architecture to realize content-based publish/subscribe using OpenFlow specifications. Nevertheless, to the best of our knowledge we are first ones to thoroughly evaluate the performance of SDN-enabled content-based publish/subscribe middleware.

8. CONCLUSION

In this paper we have proposed the PLEROMA middleware leveraging line-rate performance for content-based publish/subscribe in Software-definable computer networks. In particular, we have proposed methods that preserve the performance characteristics of PLEROMA in the presence of dynamic subscriptions and publications. Our evaluations show that PLEROMA i) imposes very low latency in mediating events between publisher and subscriber, ii) achieves low latency reconfiguration, iii) reduces the control overhead in managing multiple partitions, and iv) allows for limiting the number of flows and preserve at the same time bandwidth efficiency. Our future research will be on further optimizing the proposed methods to efficiently configure SDN-based publish/subscribe systems so that predictable performance guarantees can also be provided. In this direction, beyond the presented algorithms for forwarding events and performing reconfigurations, new mechanisms need to be introduced in order to detect and react to overload situation in the presence of a dynamic workload.

Acknowledgments

We are very grateful to Frank Dürr and Thomas Kohler for providing us with deep insight in the performance of hardware switches [5]. Moreover, we would like to thank all students contributing to the implementation and testing of PLEROMA. Special thanks to Mohamed El-Shamouty for supporting us day and night in the evaluations.

9. REFERENCES

[1] F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *Proceedings of the 23th IEEE international conference on computer communications, joint conference of the*

IEEE computer and communications societies (INFOCOM). IEEE, 2004.

[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*, 19(3):332–383, 2001.

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

[4] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *Proceedings of the ACM symposium on applied computing (SAC)*, 2004.

[5] F. Dürr and T. Kohler. Comparing the Forwarding Latency of OpenFlow Hardware and Software Switches. Tecnical report 2014/04, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2014.

[6] C. Esposito, D. Cotroneo, and A. Gokhale. Reliable publish/subscribe middleware for time-sensitive Internet-scale applications. In *Proceedings of the 3rd ACM international conference on distributed event-based systems (DEBS)*, 2009.

[7] Project Floodlight: open source software for building software defined networks. `http://www.projectfloodlight.org/`.

[8] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*, pages 164–205. 2010.

[9] M. A. Jaeger, H. Parzyjegla, G. Muehl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *Proceedings of the ACM symposium on applied computing (SAC)*, 2007.

[10] K. R. Jayaram, C. Jayalath, and P. Eugster. Parametric subscriptions for content-based publish/subscribe networks. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 128–147, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: line speed publish/subscribe inter-networking. *ACM SIGCOMM Computer Communication Review*, 39(4):195–206, 2009.

[12] B. Koldehofe, F. Dürr, and M. A. Tariq. Event-based Systems Meet Software-defined Networking. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 649–671. John Wiley & Sons, Ltd., June 2013.

[13] 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 MW4NG Workshop of the 13th International Middleware Conference 2012*, pages 3:1–3:6, 2012.

[14] 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*, MW4NG '12, pages 3:1–3:6, New York, NY, USA, 2012. ACM.

[15] B. Koldehofe, B. Ottenwälder, K. Rothermel, and U. Ramachandran. Moving Range Queries in Distributed Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS).*, pages 201–212, Berlin, July 2012. ACM.

[16] B. Lantz, B. Heller, and N. McKeown. A network on a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[17] A. Majumder, N. Shrivastava, R. Rastogi, and A. Srinivasan. Scalable content-based routing in pub/sub systems. In *Proceedings of the 28th IEEE international conference on computer communications, joint conference of the IEEE computer and communications societies (INFOCOM)*, 2009.

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

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

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

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

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

[23] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[24] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel. Distributed heterogeneous event processing: Enhancing scalability and interoperability of cep in an industrial context. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 150–159, New York, NY, USA, 2010. ACM.

[25] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, and K. Rothermel. Meeting subscriber-defined QoS constraints in publish/subscribe systems. *Concurrency and Computation: Practice and Experience*, 23(11):2140–2153, 2011.

[26] 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 (DEBS)*, pages 213–224, 2012.

[27] M. A. Tariq, B. Koldehofe, and K. Rothermel. Efficient content-based routing with network topology inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2013.

[28] K. Zhang and H.-A. Jacobsen. Sdn-like: The next generation of pub/sub. *CoRR*, 2013.