High Performance Publish/Subscribe Middleware in Software-defined Networks

Sukanya Bhowmik†, Muhammad Adnan Tariq†, Boris Koldehofe∓, Frank Dürr†, Thomas Kohler†, and Kurt Rothermel†

†University of Stuttgart, {first name.last name}@ipvs.uni-stuttgart.de
∓University of Darmstadt, {first name.last name}@kom.tu-darmstadt.de

***Abstract*—With the increasing popularity of Software-defined Networking (SDN), Ternary Content-Addressable Memory (TCAM) of switches can be directly accessed by a publish/subscribe middleware to perform filtering operations at low latency. In this way, three important requirements for a publish/subscribe middleware can be fulfilled: namely, bandwidth efficiency, line-rate performance, and low latency in forwarding messages between producers and consumers. Nevertheless, it is challenging to sustain line-rate performance in the presence of dynamically changing interests of producers and consumers. In this article, we realize a scalable, SDN-based publish/subscribe middleware, called PLEROMA, that performs efficient forwarding at line-rate. Moreover, PLEROMA offers methods to efficiently reconfigure a deployed topology in the presence of dynamic subscriptions and advertisements. We evaluate the performance of both the data plane and the control plane of PLEROMA to support our claim. Furthermore, we evaluate and benchmark the performances of SDN-compliant hardware and software switches in the context of our middleware.**

***Index Terms*—Content-based Routing, Publish/Subscribe, Software-defined Networking, Consistency, Middleware**

I. Introduction

Content-based routing as provided by publish/subscribe (pub/sub) systems has evolved as a key paradigm for interactions between loosely coupled application components (content publishers and subscribers). The basic idea of content-based routing is to utilize the diversity of information exchanged between application components to increase the efficiency of forwarding. Using content-based forwarding rules (also called content filters) installed on content-based routers (also termed brokers), bandwidth-efficiency is increased by only forwarding content to the subset of subscribers who are actually interested in the published content.

Many middleware implementations for content-based pub/sub have been developed over the last decade (e.g., [17], [19], [10]). These approaches have proven to efficiently support content-based routing between a large number of distributed application components. However, implemented in an overlay network of software brokers, their performance is still far behind the performance of communication protocols implemented on the network layer w.r.t. throughput, end-to-end latency, and bandwidth efficiency. This is because these middleware implementations are unable to exploit the performance benefits of standard multilayer switches or hardware routers capable of forwarding packets at line-rate and achieving data rates of 10 Gbps and more using dedicated hardware such as Ternary Content Addressable Memory (TCAM). Moreover, routing on overlay networks may not be bandwidth efficient due to the dissemination of the same packet multiple times over the same physical link being shared by multiple logical links. This is in contrast to routing on the network layer.

Therefore, it would be highly attractive to implement content-based routing directly on the network layer. However, even till the recent past, changes to existing standard network protocols and hardware seemed to be unrealistic and most research refrained from network layer implementations. This, however, has changed with the advent of software-defined networking (SDN), which provides the possibility to go beyond the limitations of traditional network architectures by allowing software to flexibly configure the network. With the help of standards like OpenFlow [9], the lower-level network functionalities are abstracted and presented as network services. In doing so, SDN establishes a clear distinction between the control plane and the data (forwarding) plane by extracting all control logic from the forwarding devices and hosting it on a logically centralized component, the *controller*. A controller has an integrated view of the entire network. It has the ability to collect and process information (e.g., network statistics, application-specific requests) from the data plane and perform network updates accordingly by modifying the state of network devices (i.e., switches).

SDN technology can be exploited by existing content-based publish/subscribe middleware to enhance performance on the data plane w.r.t. throughput, end-to-end latency, and bandwidth efficiency in highly demanding application fields such as financial trading, traffic control, online gaming, and smart grid. This is because the expressive filtering of events, which was previously done at the application layer, can now be performed on the TCAM memory of switches (in the date plane) at line-rate [34]. Moreover, since the logically centralized controller has a global view of the underlying topology, it is possible to install a network topology for forwarding information between producers and consumers in a bandwidth efficient manner.

In this article, we mainly focus on fulfilling two requirements, one at the data plane and the other at the control plane, towards achieving a scalable SDN-based publish/subscribe middleware. First, while a deployed network topology offers line-rate performance in forwarding events at the data plane, efficient mapping of content to expressive filters capable of

being installed in the TCAM of switches can prove to be extremely challenging and an important requirement. Content representation should be expressive enough to keep the amount of unnecessary traffic in the network to a minimum, even in the presence of hardware limitations, e.g., limited number of flow table entries. 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 thousands to hundreds of thousands flow entries per switch [12]. Second, in the presence of high dynamics, the logically centralized control plane needs to engage in very frequent network topology updates with changing interests of publishers and subscribers. 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 [19], [23]. 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 [19]. Traffic monitoring and online gaming require location-dependent updates of runtime parameters such as the location of objects, often at higher frequency than one update per minute per subscriber [23]. Providing a scalable control plane with high responsiveness to such topology change requests in a dynamically changing environment is, therefore, crucial to the middleware and constitutes the second requirement.

This article is an extended version of previous publications on the PLEROMA middleware [34], [5], [22] and provides a detailed insight into various aspects of it. In particular, the contributions of this article are the design, implementation, and detailed evaluation of an SDN-based publish/subscribe middleware offering line-rate forwarding of events and methods for its scalable reconfiguration addressing the aforementioned requirements. To the best of our knowledge, we are the first to provide a middleware implementation using SDN [25].

The remainder of this article is structured as follows. In Section II, we first introduce the architecture of the PLEROMA middleware. Section III presents mechanisms that i) provide a content representation capable of being mapped to hardware switches, ii) achieve reconfiguration of the network topology, and iii) limit the number of flows to be installed inside an SDN switch. In Section IV, we focus on increasing responsiveness of the control plane to data plane requests by introducing scaling mechanisms and provide methods to reduce the number of flow updates on switches. Finally, in Section V, we present detailed performance evaluation of the data plane and the control plane of the PLEROMA middleware. We also evaluate and benchmark the performances of real hardware switches and virtual switches implemented in software in the context of our middleware. We conclude with a comparison to related state of the art systems and a summary of our work.

II. The PLEROMA Middleware

A content-based publish/subscribe consists of mainly two types of participants—publishers and subscribers—, which are connected to switches in a software-defined network. Publishers specify the information they intend to publish by sending

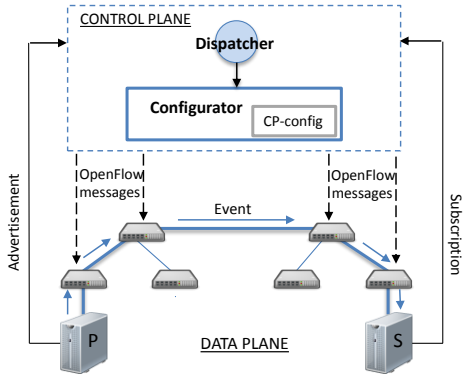

Fig. 1: SDN-based Pub/Sub Middleware

advertisements to the control plane. Likewise, subscribers specify information they are interested in receiving by sending subscriptions. The logically centralized controller collects all control requests ((un)advertisement/(un)subscription) from participants based on which it installs paths on the data plane between each publisher and all its interested subscribers. In doing so, it configures the network's data plane by proactively installing suitable flow table entries—representing content-based filters—on SDN-configurable switches by utilizing the widely accepted OpenFlow standard [9] (cf. Figure 1). We specifically use IP-Multicast addresses in flow table entries to represent filters in PLEROMA. In this paper, we use the term *flow* to represent a flow table entry on an SDN-configurable switch. A flow further defines an outgoing port of a switch to which a packet with a matching header field (packet-header-based filtering) is forwarded. Note, our content representation mechanisms are generic and other fields, e.g., MAC addresses or VLAN tags [31], can also be used for the same purpose.

Figure 1 illustrates the architecture of the PLEROMA middleware, which establishes line-rate content-based routing. The control plane consists of a two-tiered architecture; a *dispatcher* collects control requests from publishers and subscribers in a software-defined network, and a component, known as *configurator*, processes these requests and performs network updates accordingly. SDN allows the *dispatcher* and the *configurator*, constituting the logically centralized controller, to acquire a global view of the entire network and configure it as needed. The *dispatcher* serves as the entry point to the control plane. It collects all data plane control requests and forwards them to the *configurator*.

On receiving control requests forwarded by the *dispatcher*, a *configurator* configures the network topology to establish short and bandwidth efficient paths between publishers and subscribers. So, a *configurator* needs to read the current state of the network, decide on updates, and then make changes to the network state for each control request. The network state is represented by network configuration that consists of (i) all switches constituting the network, (ii) all links connecting the switches to account for a dissemination structure, and (iii) all pub/sub flows deployed on each switch. In general, the network configuration is maintained both at the data plane and the control plane of a software-defined network. On the one hand, the network configuration at the data plane (denoted as DP-config) is maintained implicitly as a result of pub/sub flows

deployed on the TCAM of hardware switches. On the other hand, a control plane network configuration (denoted as CP-config) is maintained by the logically centralized control plane which serves as a reflection of DP-config. The *configurator* needs to maintain the network state CP-config so that it does not need to query the switches in the data plane and read their states for processing every control request. As mentioned before, installing paths between publishers and subscribers involves reading the existing flows of each switch (along the path), taking decisions on flow changes and writing these changes to the switch. Since the controller assumes CP-config to be identical to DP-config, it uses CP-config to read existing flows and decide on flow changes. On taking a decision, the controller sends the new flow changes to the hardware switch, resulting in a change in DP-config. Meanwhile, the controller also performs these flow changes in the CP-config to ensure that it remains consistent with DP-config.

The above description of the PLEROMA middleware directly leads us to the problems to be solved on the data plane and the control plane. In particular, the challenges on the data plane include the (i) mapping of expressive content-filters to flow entries in hardware switches, (ii) design of a bandwidth and latency efficient dissemination structure for packet forwarding, and (iii) preserving bandwidth efficiency in the presence of hardware limitations w.r.t. number of bits available for filtering. These challenges are addressed in Section III where all mechanisms employed to enable in-network content-based filtering are explained in details. Moreover, the challenges on the control plane include (i) design of a consistent control plane which is highly responsive to dynamically changing subscriber interests, and (ii) addressing limitations of SDN-compliant switches w.r.t. the rate at which flow updates are performed. These challenges are addressed in Section IV where a mechanism to scale the control plane in a consistent manner is discussed in details.

III. In-network Content-based Filtering

In this section, we detail the methods employed to address the aforementioned challenges related to the data plane. We, first, provide a mechanism to linearize content such that advertisements, subscriptions and events can be represented as match fields in flows of switches (or header field of an event packet). This is followed by an algorithm that details the processing of both advertisements and subscription requests at the control plane such that necessary paths are deployed between publishers and relevant subscribers by installing the aforementioned linearized content filters represented by switch flows along these paths. Moreover, we also discuss ways to preserve bandwidth efficiency in the presence of hardware limitations while expressing content filters.

A. Content Representation

To ensure high expressiveness and establish paths with low-bandwidth usage between publishers and subscribers, we follow the content-based subscription model, i.e., an event is composed of a set of attribute value pairs. To realize the aforementioned packet-header-based filtering of events

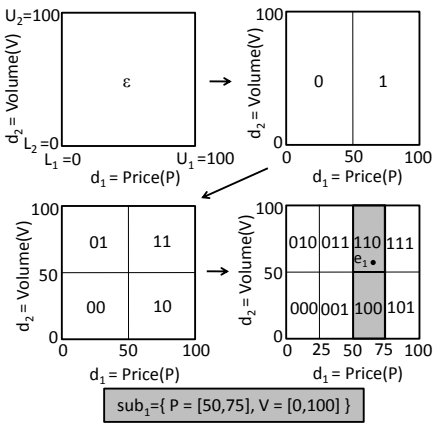

Fig. 2: Spatial Indexing [22]

at the data plane, we need an efficient mapping between content attributes and flow identifiers (i.e., one or more header fields that uniquely identify flow entries in the flow tables of switches). There are two steps to this mapping process.

The first step yields a binary representation of content following the principle of spatial indexing [22]. 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 and a subscription or advertisement as a subspace in Ω. Building on the principle of spatial indexing, we can divide the event space into regular subspaces that serve as enclosing approximations for events, advertisements, and subscriptions. In fact, since events are points in Ω, they are represented by subspaces of finest possible granularity. Any subspace can be identified by a binary string named dz-expression (in short dz). In particular, dz-expressions fulfill the following characteristics. 1) The shorter the dz, the larger is the corresponding subspace in Ω. Again, since events are points in Ω, they are represented by dzs of maximum length. 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 the 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.

We illustrate spatial indexing with an example in Figure 2 where we consider a stock quote dissemination system implemented by the pub/sub paradigm. In this example, we consider two attributes (or dimensions) stock price (P) and stock volume (V) of a stock quote dissemination system. An advertisement/subscription can be composed of several dzs, denoted as DZ. For instance, to approximate the subscription *sub*$_1$ in Figure 2, two dzs 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. For the sake of simplicity, here, we consider only two dimensions. However, multiple dimensions can be indexed

which can even include string attributes such as company name in the stock quote example. The string attributes can be linearized by hashing and indexed in a similar manner [30].

The second step involves the mapping of the generated binary strings (dzs) to flow identifiers. Using the above relations, an event e disseminated by a publisher p will comprise in its packet header field(s) a dz that represents its attribute values. In order to deliver e to a subscriber s with a subscription sub which expresses an interest in e, the *configurator* must have installed on each switch along the path (between p and s) a flow whose match field(s) matches the header field(s) of this event. With respect to spatial indexing, an event will match a subscription filter if it lies within the subspace representing the subscription in Ω, i.e., the dz representing sub covers ($\succ$) the dz representing e. So, for a match to occur between e and sub, we utilize the characteristics of dzs such that the match field(s) in flows representing the filters for sub covers header field(s) of event packet representing e. To this end, we use a range of IPv6 multicast addresses reserved for pub/sub traffic, as the flow identifiers. So, a subscription/advertisement 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 covering relation between subspaces is accommodated in IP addresses with the help of Class-less Interdomain Routing (CIDR) style masking supported by hardware switches where the 'don't care' symbol (*) is used to represent masking operations. An event is also represented as an IPv6 multicast address and forms part of the header of the event packet. This enables header-based matching and subsequent forwarding of the event packet as dictated by a flow on account of a match. So, continuing the stock quote example from Figure 2, the dz representing the subspace $\{110\}$ is converted to the IP address `ff0e:c000:*` (`ff0e:c000::/19`). Now, if the event e_1={P=65, V=55} in the figure, which lies within (matches) *sub*$_1$, is represented by the dz 110010, then it is converted to an IP address `ff0e:c800::` and header-based matching of this event packet takes place with the installed flows for *sub*$_1$.

B. Topology Reconfiguration

An efficient approach to topology reconfiguration is central to pub/sub using SDN. To this end, we need to maintain a dissemination structure which considers as constraints latency efficiency, bandwidth usage, and cost efficiency to update the network topology. Clearly, the lowest latency is achieved if a *configurator* establishes a shortest path for each publisher/subscriber pair. However, this severely limits the reuse in forwarding an event on common paths, i.e., the possibility to share common subpath(s) and, therefore, bandwidth between a publisher and subscribers with overlapping subscriptions. Moreover, each new subscription or advertisement would trigger updates of the network topology to add paths between all relevant publishers and subscribers and, therefore, impose a very high reconfiguration cost.

A common alternative—often taken by traditional broker-based systems [17]—is to embed the paths between publishers and subscribers by means of filters in a single spanning tree. The spanning tree reflects low latency paths between any pair of publisher and subscriber. Since all paths between publishers and subscribers are embedded in the same tree, the number of times an event needs to be forwarded is significantly reduced. The reconfiguration cost is also limited to the edges in the spanning tree and is significantly reduced wherever subscriptions and advertisements overlap.

As a result, the PLEROMA middleware maintains a spanning tree (comprising switches), denoted by t, at the *configurator*, to account for an acyclic dissemination structure on which paths are embedded between publishers and subscribers by installing appropriate flows (filters) on switches along these paths. More specifically, the dissemination structure of CP-config maintained at the control plane (and DP-config maintained implicitly at the data plane) represents the aforementioned spanning tree (cf. Section II). As a result, a spanning tree maintained at the control plane, a CP-config and a DP-config are synonymous in the rest of this article.

As mentioned earlier, installing paths between publishers and subscribers involves reading the existing flows of each switch (along the path), taking decisions on flow changes and writing these changes to the switch. To do so, the *configurator* uses CP-config to read existing flows and decide on flow changes under the assumption that CP-config is identical to DP-config. In a later section, we discuss the process of ensuring consistency between CP-config and DP-config even in the presence of failures. In order to understand the decision-making process to determine flow changes on a switch, it is important to understand how the *configurator* processes each type of control request, which is the subject of discussion in the remaining part of this subsection.

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

a) Advertisements and Subscriptions: On arrival of an advertisement, denoted by $DZ(p)$, from a publisher p, the *configurator* notes each dz_i in $DZ(p)$ and adds p to the spanning tree, denoted by t. The *configurator* then checks for already existing subscribers in t whose subscriptions overlap with $DZ(p)$. If there is no overlap, then no further actions are taken. However, if an overlap exists, then the *configurator* establishes paths between the publisher p and all subscribers with overlapping subscriptions in t. Each path between a publisher p and a subscriber s on t only forwards the events matching the subspaces overlapped between $DZ(s)$ and $DZ(p)$ (cf. Algorithm 1, lines 1-6). In this way false positives (events delivered to a subscriber that is not interested in receiving them) are avoided.

Subscription requests are handled similarly as described formally in lines 7-12 of Algorithm 1. On arrival of a subscription, as a first step, the *configurator* calculates the *route* between the subscriber s and each relevant publisher p on the

Algorithm 1 Publish/Subscribe maintenance at a single *configurator*

1: **upon event** Receive(ADV, p, $DZ(p)$) **do**
2: **for all** $dz_i \in DZ(p)$ **do**
3: subSet = $\{s \in S \wedge \exists dz_j \in DZ(s) : dz_i \succ dz_j \vee dz_j \succ dz_i\}$ // Subscribers with overlapping $DZ(s)$
4: **for all** $s \in$ subSet **do**
5: overlapWithSub = $dz_i \cap DZ(p)$
6: **flowAddition**(overlapWithSub, $\langle p, s, t \rangle$, t)

7: **upon event** Receive(SUB, s, $DZ(s)$) **do**
8: **for all** $dz_i \in DZ(s)$ **do**
9: pubSet = $\{p \in P_t \wedge \exists dz_j \in DZ(p) : dz_i \succ dz_j \vee dz_j \succ dz_i\}$ // Publishers with overlapping $DZ(p)$
10: **for all** $p \in$ pubSet **do**
11: overlapWithPub = $dz_i \cap DZ(p)$
12: **flowAddition**(overlapWithPub, $\langle p, s, t \rangle$, t)

13: **procedure** flowAddition(dz, $\langle p, s, t \rangle$, t) **do**
14: destIP = $(binary(\texttt{ff0e:b400}) \& (dz \ll 112 - |dz|)) \setminus 16 + |dz|$
15: **for all** $r_i \in \langle p, s, t \rangle$ **do**
16: Flow $fl_n = MF \cup IS \cup PO$
17: $fl_n.MF =$ destIP
18: $fl_n.PO =$ default value
19: $fl_n.IS.oP = \{r_i.oP_i\}$
20: curFlow = **getCurrentFlowsFromSwitch**($r_i.R_i$)
21: **if** r_i is last entry in $\langle p, s, t \rangle$ **then**
22: $fl_n.IS$.set-destIP = s.IP
23: **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
24: **for all** $fl_c \in$ curFlow : $fl_n \succ fl_c$ **do** // Case 3
25: **deleteFlowFromSwitch**(fl_c, $r_i.R_i$)
26: **for all** $fl_c \in$ curFlow : $fl_c \gtrapprox fl_n$ **do** // Case 4
27: $fl_n.IS.oP = fl_n.IS.oP \cup fl_c.IS.oP$
28: **increasePriority**($fl_n.PO$)
29: **for all** $fl_c \in$ curFlow : $fl_n \gtrapprox fl_c$ **do** // Case 5
30: $fl_c.IS.oP = fl_c.IS.oP \cup fl_n.IS.oP$
31: **increasePriority**($fl_c.PO$)
32: **modifyFlowOnSwitch**(fl_c, $r_i.R_i$)
33: **addFlowOnSwitch**(fl_n, $r_i.R_i$)

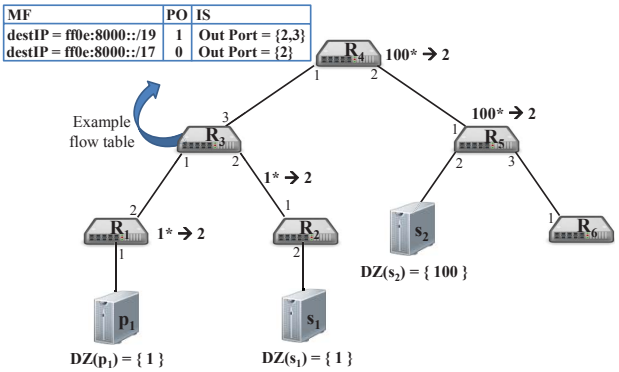

Fig. 3: Forwarding in the switch network. Match fields of flows in R_1, R_2, R_4-R_6 are shown as dzs. Flows follow the notation MF $\rightarrow$ IS : PO

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 *configurator* 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 that only the events matching the overlapped subspaces (i.e., $DZ(s) \cap DZ(p)$) are forwarded on the path. The process of establishing paths along the switch network is discussed in detail later in this section.

b) Flow installation: The installation of flows on the switches requires to specify the *match field (MF)*, *instruction set (IS)*, and *priority order (PO)* of a flow [31]. The matching field defines the header information against which packets are matched. Recall that PLEROMA uses for interoperability with other services IP-multicast ranges to embed dz-expressions. For instance, subspaces with $dz = 101101$ and $dz = 101$ are converted to IPv6 multicast addresses `ff0e:b400:*` and `ff0e:a000:*`, respectively. Therefore, an event $dz = 101101$ can be matched against a flow with $dz = 101$ by a hardware switch during forwarding, i.e., `ff0e:a000::/19` $\succ$ `ff0e:b400::/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. Also, the priority order needs to be defined to decide on the order in which flows will be applied to a packet. For example, in Figure 3, an incoming event ($dz = 1001$) on switch R_3 matches multiple flows with $dz = 1$ and $dz = 100$. However, the switch only follows the instructions of the first match. Therefore, to ensure proper forwarding, the flow installation gives higher priority to the flows with longer dz. In Figure 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 IP address in the match field of fl_2 is covered by the dz of fl_1, and (ii) the out ports to which a packet matching fl_2 is forwarded are a subset of those specified in the IS of fl_1. Likewise, a partial containment relation ($\gtrapprox$) can be defined between flows of a switch (or flows to be installed on a switch). A flow fl_1 partially covers (or contains) another flow fl_2, denoted by $fl_1 \gtrapprox fl_2$, if dz associated with the match field of fl_1 covers dz of fl_2, but not all the out ports used for forwarding packets matching fl_2 are listed in the IS of fl_1.

The procedure *flowAddition* is used by the *configurator* 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 13 - 33). The dz used for creating the match field of the new flows (to be added in the switch network) is determined from the overlap between $DZ(s)$ and $DZ(p)$, as mentioned earlier.

In more detail, the *configurator* 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. Continuing the example from Figure 3, the cases are explained w.r.t. the changes to the flow tables of the switches on the arrival of new subscriber s_3 with subscription $DZ(s_3) = \{10\}$ as depicted in Figure 4. (1) If the flows are not currently installed on a switch, then the new flow fl_n is simply added to the flow table of that switch, e.g., a new flow

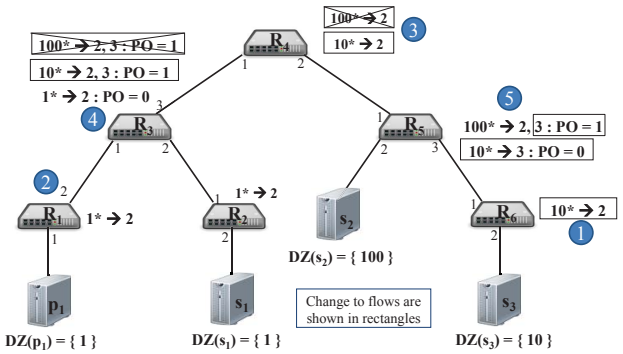

Fig. 4: Flow maintenance on the arrival of s_3. The numbers in the circles correspond to the cases explained in text.

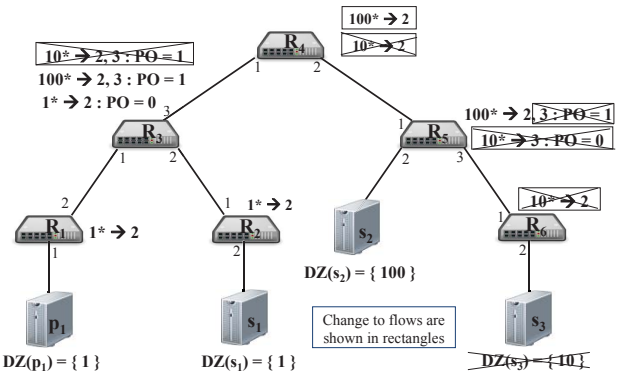

Fig. 5: Flow maintenance on the departure of s_3

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 when s_3 subscribes. The flow $\{10*\}$ that needed to be installed on R_1 to direct required traffic towards s_3 is covered by the already existing flow $\{1*\}$ which directs traffic that includes required traffic for s_3 along the same direction. So, an additional flow in this case will be redundant. (3) If an existing flow fl_c is covered by the new flow fl_n, then the new flow fl_n is added and fl_c is deleted from the flow table as it is no longer needed, e.g., in Figure 4 existing flows associated with $dz = 100$ are replaced by new flows with $dz = 10$ on R_3 and R_4. This follows from the argument of case (2). So, the existing flow which is covered by the new flow should be replaced to avoid redundancy. (4) If the new flow fl_n is partially covered by an existing flow fl_c (i.e., $fl_c \gtrsim fl_n$), then fl_n should be added with high priority and should include the out ports in the IS of fl_c, as depicted by R_3 in Figure 4. This ensures that traffic specific to the flow $\{10*\}$ (subscription of s_3) strictly matches it and gets forwarded towards both s_3 and s_1. The remaining traffic that is specific only to s_1 and that does not match the new flow will now be forwarded by the existing flow $\{1*\}$ only to s_1. (5) Finally, if the existing flow fl_c is partially covered by the new flow fl_n, then besides adding fl_n to the flow table, the existing flow fl_c should be updated to include out ports used by fl_n and to hold higher priority than fl_n, e.g., in Figure 4 an additional out port (i.e., $oP = 3$) and a higher priority order is assigned to an existing flow $\{100*\}$ on R_5. This follows similar logic as case (4).

c) Unsubscriptions and Unadvertisements: We, also, briefly discuss the handling of unsubscriptions and unadvertisements by the *configurator*. Handling of an unsubscription or unadvertisement is the exact reverse process of handling a subscription or advertisement. On the arrival of an unsubscription, the subscriber s, associated with the corresponding subscription, is removed from t. This is accomplished by removing previously established paths between s and all publishers with overlapping advertisements. To remove a path on t, the flows are either deleted or downgraded depending upon other subscribers reachable (w.r.t. their relevant publishers) via a particular switch. For example, on arrival of an unsubscription from s_3 in Figure 5, the path between p_1 and s_3 comprising of switches R_1, R_3, R_4, R_5 and R_6 needs to be removed. However, the existing flows on these switches determining this path cannot simply be removed as each of these flows may share paths to other subscribers based on the covering relations between flows as seen earlier in this section. For example, the flow with $dz = 10$ is deleted from the flow table of R_6 as no other subscriber is reachable w.r.t. p_1 via R_6. However, the flows installed on switches R_3, R_4, and R_5 have to be downgraded from $dz = 10$ to 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. Downgrading not only ensures that no further events are forwarded to s_3 but also ensures that no other subscriber paths get affected due to these updates. So, in this example, s_2 continues to receive relevant events as downgrading of flows does not affect its path from p_1. Likewise, an unadvertisement from a publisher p is handled by removing the previously established paths in the switch network between the publisher p and all subscribers with overlapping subscriptions on t with which the publisher p is associated.

C. 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 publish/subscribe. Similarly, for an event space with many attributes, the number of dz in DZ (i.e., subspaces) for an accurate subscription (or advertisement) representation may be very high and may produce flow tables with large numbers 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, namely, selectivity of subscriptions and distribution of events along that dimension. The main idea is to deem dimensions where event traffic matches most subscriptions as less important for dimension selection. Therefore, PLEROMA selects those dimensions that have high variability (or in other words variance) in the set of subscriptions matched by

the events (according to current event traffic) to perform in-network filtering of events. Further details of this mechanism are discussed in [34], [3]. Also, we show the benefits of dimension selection in our evaluations in Section V.

IV. Scalable Handling of Control Requests

The topology reconfiguration efforts are significant in an SDN-based pub/sub middleware. In a scenario with frequent concurrent control requests from multiple participants, a design with a single *configurator* will result in very poor control plane responsiveness. Here, we define *response time* as the time from the issuance of a control request by a participant till the completion of all topology reconfiguration associated with this request by the control plane. For example, the response time to a subscription is the time elapsed from the issuance of the subscription until the subscriber starts receiving events. As a single *configurator* processes each control request sequentially, the response time increases significantly in the face of high dynamics. This problem motivates us to introduce multiple *configurator* instances in the control plane enabling concurrent processing of control requests. However, scaling the control plane implies concurrent processing of requests for improved responsiveness which in turn raises questions on control plane consistency.

In general, two important problems have to be addressed to ensure control plane consistency in an SDN-based pub/sub middleware. These problems are i) maintaining consistent network configuration (i.e., CP-config and DP-config) in the presence of concurrent updates by multiple *configurators*, and ii) keeping CP-config consistent with DP-config in the presence of failures. In Section IV-A, we strictly focus on the first problem and address the second problem in Section IV-B. For simplicity and without loss of generality, we discuss the first problem only with respect to CP-config, as consistent maintenance of CP-config in the face of concurrency (and absence of failures) implies consistent DP-config.

In more detail, the *configurators* execute the same control logic and operate on the same CP-config concurrently. On receiving a request, a *configurator* performs operations on switches along the paths between publishers and subscribers in order to deploy flow updates. As mentioned in Section III, at each switch, the *configurator* performs an action that consists of an ordered sequence of three operations. The three operations include reading flows from a switch, deciding on the changes to be made to the flows, and finally writing these changes back to the switch. The concurrent execution of such actions by two or more *configurators* can result in their sequences being interleaved. This raises concurrency related issues resulting in false negatives (events not delivered to a subscriber despite its interest in receiving them) or false positives (events delivered to a subscriber that is not interested in receiving them) at the subscriber end.

While understanding the above mentioned concurrency issues, we identify conflicting actions in an SDN-based pub/sub middleware. Typical concurrency issues are expected to arise when two or more *configurators* try to access the same resource, more specifically the same flow on a switch. Moreover, as the decisions to make further flow updates depend on already existing related flows ($\succ$ and $\succsim$), interleaving sequences result in conflicting actions when two or more *configurators* concurrently affect not only the same flow but also flows that are related to each other ($\succ$ and $\succsim$) at a switch as this may result in reading from, deciding on, or writing to inconsistent related flow-set states.

Referring to Figure 4, let us look at an example where two overlapping subscription requests sub_x=$\{00\}$ from subscriber s_2 and sub_y=$\{00\}$ from subscriber s_3 are simultaneously dispatched to two *configurators* c_x , $c_y \in C$ respectively. Both follow the aforementioned request handling process and perform actions on relevant switches. We specifically focus on switch R_5 where two separate flows will be installed by the two *configurators* with the exact same filter $\{00\}$ but different IS, i.e., out port 2 will be set for the flow installed by c_x (fl_x) and out port 3 for the flow installed by c_y (fl_y). Since deploying flows on CP-config implies deploying them on DP-config, now, if an event packet lying in subspace $\{00\}$ arrives at R_5 in the data plane, it follows the instruction set of either fl_x or fl_y, but never both as the matching of a packet at a switch is terminated as soon as the first match is found. In either case, one of the two subscribers is affected by false negatives compromising correctness of the system. This is a simple case of false negatives at a subscriber due to the interleaving of sequences of operations constituting two actions and belonging to two *configurators*. Clearly, false negatives occurred because flows fl_x and fl_y concurrently added by c_x and c_y are in aforementioned flow containment relation, which essentially results in addressing the same filter in R_5.

Therefore, from the above discussion, we formally define conflicting actions in the PLEROMA middleware as follows :

Definition 1 *Two actions are in conflict if (i) they belong to different* configurators*, (ii) both of them access the same switch, and (iii) both of them affect flows that are bound by the flow relations, i.e., complete containment ($\succ$) and partial containment ($\succsim$).*

To ensure consistency, conflicting actions must be serialized.

A. Scaling by State Partitioning

The identification of conflicting actions (cf. Definition 1) leads us to the idea of using flow relations to identify the actions that need to be serialized on a switch. Since the dzs representing the subscriptions/advertisements (in control requests) are directly mapped to flows added to switches (cf. Section III), two control requests where one dz covers or is identical to the other (overlapping subspaces in Ω) yield flows related ($\succ$ and $\succsim$) to each other. This means that concurrent processing of overlapping control requests at a switch will result in conflicting actions and must be ordered sequentially. Control requests with non-overlapping subspaces in Ω, however, can undergo concurrent processing without any issues. This directly leads us to the idea of partitioning the event-space in a disjoint way such that flows corresponding to different partitions in Ω are maintained in separate CP-configs.

So, we divide the event-space (Ω) into multiple disjoint, continuous partitions. A partition is nothing but a subspace

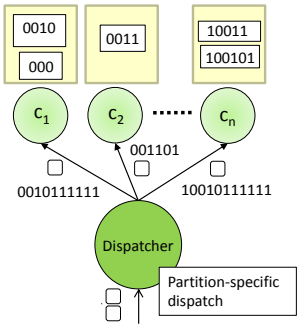

Fig. 6: State-Partitioning Approach

in Ω and may be represented in the same way, i.e., by a dz. Disjoint event-space partitioning may yield equal or unequal partitions depending on the partitioning criteria. However, it is important to note that, in any case, the partition set, denoted by P, is non-overlapping and fully covers Ω. Mechanisms for content or event-space partitioning have been extensively researched in various fields of computer science [37], [38] and will not be discussed further in this article. Henceforth, we assume that P consists of k partitions and $k >> n$ where n denotes the total number of *configurators*. The middleware maintains a set of independently configurable CP-configs (denoted by CP) having a one-to-one mapping with these partitions. This results in the creation of k CP-configs where each configuration, $cp \in CP$, is represented by the dz of the corresponding partition. Again, each switch in each cp contains only those flows that are associated with the event-space partition that this configuration represents. This implies that the spanning tree maintained by CP-config is responsible for the dissemination of only a set of events that lies in its designated subspace. In the remaining part of this article, a CP-config ($cp_i \in CP$) is considered to be synonymous with a partition ($p_i \in P$).

Having partitioned the event space, our scaling approach, the state-partitioning approach (SPA), now operates on these disjoint CP-configs. In SPA, each partition is assigned exclusively to exactly one *configurator*. To ensure consistency, each *configurator* is restricted to performing reconfigurations on its assigned partitions only. So, two or more *configurators* may process different requests concurrently as they operate on completely different subspaces in Ω, i.e., they may modify the flows on the same switch concurrently without any inconsistencies as the flows affected in each case are completely unrelated. This ensures that no two *configurators* interfere with each other while performing parallel topology reconfigurations on the same network. As our design assumes $k >> n$, each *configurator* may be responsible for multiple partitions. Please note, each *configurator* needs to maintain only those CP-configs that have been assigned to it. Such a mechanism avoids all kinds of coordination overhead among *configurators* while ensuring control plane consistency in a distributed setting.

1) Topology Reconfiguration: The *dispatcher* plays a significant role in this approach. It maintains a map of the *configurators* and their associated partitions and performs partition-specific dispatch of control requests (cf. Figure 6). But first, it performs an additional step to prepare the requests for further processing. Let us denote the dz representing a control request by dz_c and that representing any partition p_i by dz_{p_i}. When a control request arrives at a *dispatcher*, it is processed by the *dispatcher* in two ways depending on whether (i) $dz_{p_i} \succeq dz_c$ or (ii) $dz_c \succ \{dz_{p_i},...,dz_{p_j}\}$. In the first case, the *dispatcher* simply dispatches the request to a *configurator* as the request is contained by one partition and affects a single CP-config. However, the second scenario portrays a case where the control request subspace spans more than a single partition. Under such circumstances, the *dispatcher* splits up the request into multiple dzs depending upon the nature of the partitions and dispatches these partial requests. This guarantees the mapping of a request to a single partition enabling the *dispatcher* to directly forward a request to a *configurator* responsible for the corresponding partition. For example, in Figure 6, if a request corresponds to $\{00\}$, the *dispatcher* first splits it up into three requests $\{000\}$, $\{0010\}$, $\{0011\}$ and then dispatches them to c_1 and c_2 as $\{00\} \succ \{000, 0010, 0011\}$. Consequently, all three CP-configs are reconfigured for this single request.

Each *configurator* maintains a request queue for each partition it is responsible for. Processing of control requests at a *configurator* takes place sequentially. This, in turn, ensures consistency within each partition. Moreover, topology reconfiguration follows the usual mechanisms discussed in Section III.

The state-partitioning approach enables concurrent processing of requests corresponding to disjoint partitions at multiple *configurators*, thus reaping the benefits of scaling. However, the true potential of this design can be realized if the workload can be balanced between *configurators*. There may be scenarios where the workload is much higher for certain partitions which burdens a few *configurators* while others remain idle. This degrades responsiveness of the control plane to control requests. For this reason, load balancing among *configurators* bears considerable significance and features as the subject of discussion in the remaining part of this section.

2) Partition Assignment: The most naive way of assigning partitions to *configurators* is a random approach where k partitions are randomly distributed among n *configurators*. However, this may result in the request load being unevenly distributed among the *configurators* which creates bottlenecks at certain *configurators* while others remain idle. As a result, instead of a random approach we employ a partition assignment approach based on previous trends of control requests.

The aforementioned assignment problem is similar to a bin-packing problem where maximum load has to be minimized among all *configurators* and we simply employ a greedy approach from literature [13] for the same. First, every partition $p \in P$ is given weights depending upon its popularity. The popularity of a partition is determined by the percentage of control requests covered by this partition that feature in history over a time window. After weighting each of the k partitions, they are sorted in decreasing order of weights. Now, at each step, the next partition is assigned to the current least loaded *configurator*. This approach results in near-optimal performance. In practice, subscriptions/advertisements change dynamically and, therefore, over time the assignment may

become sub-optimal. However, it is not feasible to frequently use this technique for load balancing as it involves reassignment of all partitions among all *configurators* incurring significant migration costs. As a result, we only use it after extended periods while a more light-weight approach is used for adaptive load balancing during these periods.

3) Adaptive Load Balancing: We identify load of a *configurator* at a given time by request queue lengths of all partitions assigned to it. A request queue, specific to a partition (say, p_j), consists of all control requests waiting to be processed by the *configurator* for an assigned partition. So, load at a *configurator* c_i may be defined as $l_i = \sum_{j=1}^{m} QL_j$, where m is the number of partitions assigned to c_i and QL_j represents queue length at p_j. When an overload condition is detected at a heavily loaded *configurator*, one or more of its assigned partitions are migrated to a *configurator* with current minimum load. This implies that the task of processing all current and future requests for the migrated partitions now lies with the newly chosen *configurator*. An overload detection is carried out by the *monitor* component. The *monitor* periodically collects load information of every *configurator*. With every periodic collection, the *monitor* calculates the average queue length at each *configurator*, denoted by l_{avg}. If the ratio of the load at a *configurator*, i.e., l_i, to l_{avg} is greater than a threshold value, then the monitor detects an overload and proceeds with partition migration. More formally, an overload is detected if, $\frac{l_i}{l_{avg}} > threshold$, where $l_{avg} = \frac{\sum_{s=1}^{n} l_s}{n}$. However, in order to avoid partition thrashing, the monitor initiates migration only if the overload condition at a *configurator* is monotonically increasing with time. Initially, the most heavily loaded partition at the overloaded *configurator* is selected for migration and the effects of migrating it to the minimally loaded *configurator* is calculated. If this results in a potential overload condition at the minimally loaded *configurator*, the monitor proceeds to calculate the feasibility of migration of the next most heavily loaded partition until a balanced migration is achieved or all partitions considered for migration.

B. Control Plane Consistency in Presence of Failures

As we have not considered failures previously, it has been sufficient to assume that CP-config is consistent with DP-config and therefore sufficient to only deal with inconsistencies arising due to concurrency between multiple *configurators*. However, lost connections (between *configurators* and switches) and switch failures may result in inconsistencies between the two configs, irrespective of whether the control plane is centralized or distributed.

Let us first consider a case where the connection between a *configurator* and a switch is lost. As a result, the updates that were pushed by a *configurator* onto a switch may not be reflected on the TCAM memory of the switch at all. If the *configurator* continues processing of requests assuming that the said changes are deployed on the switch, then this would imply inconsistencies between the two configs, resulting in incorrect system behavior. To avoid this, our middleware pushes out the flow modification requests, generated while processing a control request, to the switch and waits until the switch acknowledges the successful completion of these updates within

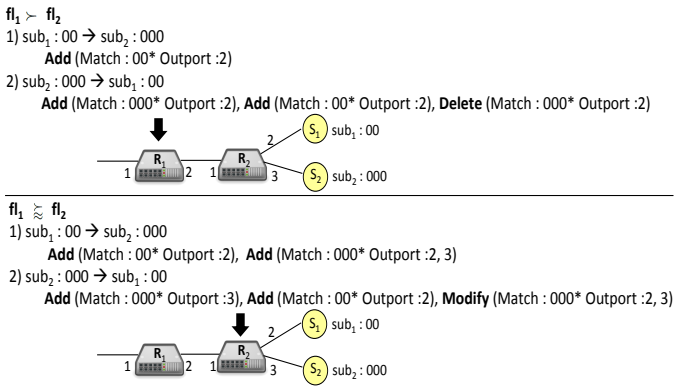

Fig. 7: Reducing Flow Operations

a given timeout. Various functionality such as bundle messages or flow monitoring available in OpenFlow version 1.4 can be efficiently used to allow a *configurator* to be notified by a switch about flow operations (addition/modification/deletion) performed on its tables. Such switch notifications can serve as acknowledgments of completed flow table updates. On receiving an acknowledgement from the switch, the *configurator* writes these changes to the CP-config and considers the control request as fully processed. If an acknowledgement does not arrive at a *configurator* within the given timeout, the *configurator* marks all the unacknowledged flow changes as undefined in CP-config. The processing of all subsequent requests that depend on undefined flows must be stalled. A *configurator* must explicitly read the current status of the switch with a missing acknowledgement using the OpenFlow standard and reflect the same in the CP-config.

Inconsistencies between CP-config and DP-config also arise due to switch failures. In case of a switch failure, the spanning tree maintained by CP-config has to be modified accordingly, which means that all paths need to be recalculated according to the new topology. The same has to be done in case of a switch recovery as this also involves a change in the network topology and must be reflected in CP-config to ensure consistency.

C. Reducing Flow Operations

Increasing responsiveness of the control plane to control requests also increases the rate at which network updates are pushed onto the switches by multiple *configurators*. With today's hardware switches supporting around 40-50 flow-table updates per second [16], it would be really beneficial if the total number of flow updates could be reduced. However, this would have to be achieved while ensuring correctness of the system, i.e., no false positives and false negatives.

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

would benefit from ordering are complete containment, i.e., $fl_i \succ fl_j$, and partial containment, i.e., $fl_i \succsim fl_j$.

Referring to the two subscriptions in the above example and their relations, we first look at complete containment between flows. The following updates would be done on a switch depending on the order in which the two subscriptions are processed. 1) If sub_i is processed before sub_j, sub_i first produces one add flow (fl_i) operation on the switch. When sub_j is processed, it does not produce any other flow updates on the switch as fl_i fully covers all events that need to be forwarded in response to sub_j. 2) If sub_j is processed before sub_i, sub_j also produces one add flow (fl_j) operation on the switch. After this, when sub_i is processed another flow (fl_i) add operation has to be performed to cover forwarding of all events matching sub_j and also those matching sub_i but not sub_j. Also, a delete operation has to be performed on fl_j as it is now redundant. Given the limitations of the flow table size on a switch, redundant flows cannot be afforded. This clearly indicates that the first ordering yields two operations less as compared to the second. Figure 7 illustrates the above discussion with an example where the ordering of two subscriptions sub_1 ($\{00\}$) and sub_2 ($\{000\}$) that would independently produce fl_1 and fl_2 yield different numbers of operations on switch R_1 as $fl_1 \succ fl_2$.

Let us now consider the second relation of partial containment between the flows. Again, we look at the number of operations required on ordering sub_i and sub_j differently. 1) If sub_i is processed before sub_j, sub_i produces one add flow (fl_i) operation. When sub_j is processed, a second flow (fl_j) add operation needs to be performed as this time the flows are only partially related and a different out port needs to be added only for sub_j. 2) However, if sub_j is processed before sub_i, sub_j produces one add flow (fl_j) operation on the switch. Now, when sub_i is processed, first a flow (fl_i) gets added for this subscription. Also, since the events relevant to fl_j are also relevant to fl_i (as $sub_i \succ sub_j$), a modify operation is performed on fl_j to accommodate the out port for sub_i. Again, the first ordering yields lesser operations as compared to the second. Please note that the reordering of subscriptions does not have an impact on the correctness of the system as, no matter how processing of requests is ordered, the final set of flows deployed on the switches is always the same. In Figure 7, at the end of processing sub_1 and sub_2, both switches have the same flows irrespective of the order in which they were processed. However, ordering may have an effect on the response time to certain requests that get scheduled later (cf. Section V).

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

V. 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 on the data plane and on the control plane. The measurements on the data plane include end-to-end delay for event dissemination as well as bandwidth efficiency in terms of false positives w.r.t. length of dz and number of flows. Those on the control plane include control plane throughput, average processing latency of control requests, and required number of flow operations on switches. We also provide micro-benchmarks by evaluating and comparing the forwarding delay of a hardware switch and a virtual switch implemented in software [12].

A. Experimental setup

We have conducted our evaluations under three environments—1) a physically distributed network of software switches (SDN-t), 2) an emulated network running on a single machine using Mininet (SDN-m), and 3) micro-benchmarks on a single hardware switch NEC PF5240. The majority of the experiments have been conducted on SDN-t [34] consisting of commodity PC hardware and virtualization technologies as used in datacenters. For SDN-t, we use a hierarchical fat-tree topology consisting of a cluster of hosts (running on commodity rack PCs) constituting 10 switches and 8 end systems. Some of these hosts act as OpenFlow switches with four physical ports by executing a production-grade software switch (Open vSwitch [32]) attached to the 4-port NIC. The other hosts act as 8 end systems (end hosts) by executing virtual machines on two physical machines. The end hosts implement the functionality to publish and subscribe events. Besides SDN-t, we have also conducted experiments on a prominent tool for emulating software-defined networks, namely, Mininet [26] (SDN-m). Mininet is an extremely flexible tool that allows to conduct experiments with different types of topology and application traffic. In order to evaluate the performance of a scaled control plane, we host the distributed control plane in a small local area network which includes a cluster of physical machines. Scaling is realized by hosting multiple *configurators* on multiple physical machines where each machine in the cluster has 4 cores, 3.4 GHz processor, and 8 GB of RAM. Two separate machines host the *dispatcher* and the *monitor*.

In order to generate workload, i.e., events and subscriptions we use both synthetic as well as real world data. With regards to synthetic data, the workload was generated using parameters similar to those used in well established publish/subscribe literature [8], [30], [41]. So, we used a content-based schema containing up to 10 attributes [30], where the domain of each attribute varies in the range $[0, 1023]$. Most real world applications, e.g., stock quote dissemination systems, perform content-based routing with not more than 10 attributes and similar domain ranges. Experiments are performed on two predominantly used models for the distributions of subscriptions and events [30], [8]. The uniform model generates random subscriptions and events independent from each other. The

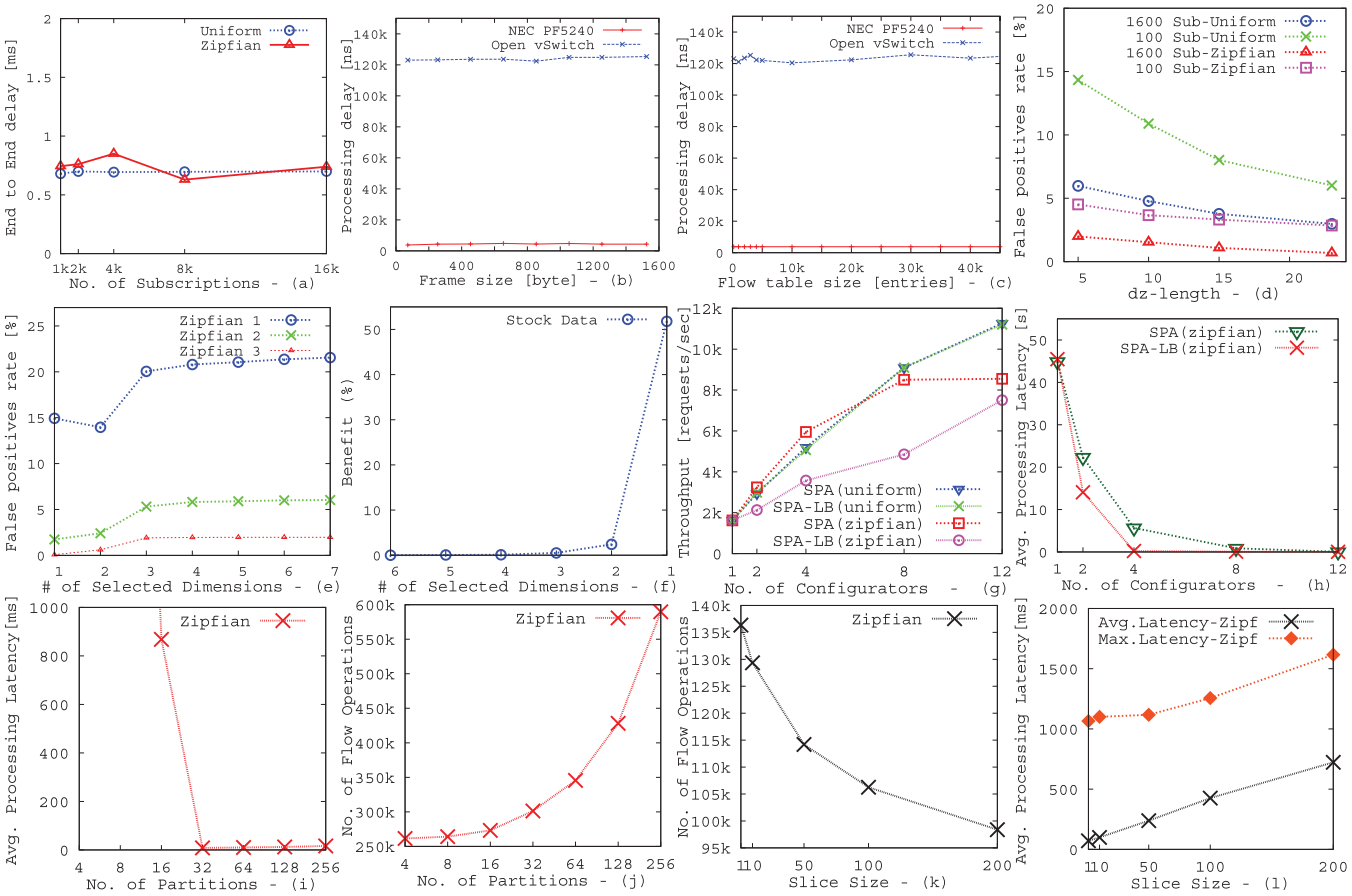

Fig. 8: Performance evaluations

interest popularity model chooses up to 8 hotspot regions around which subscriptions/events are generated using the widely used Zipfian distribution. The rate at which control requests are sent by the participants (i.e., publishers and subscribers connected to an SDN network) to the dispatcher also follows two models of distribution, namely, uniform and Poisson which are popularly used in literature [41] to model dynamics of subscription changes (control requests) over time. A uniform rate implies that the occurrences of incoming requests at the *dispatcher* are distributed uniformly on an interval of time. However, Poisson rate involves a fluctuating workload while maintaining an average rate of incoming requests at the *dispatcher* within a given interval of time. So, there may be bursts of incoming requests from time to time along with lull periods to ensure an average rate at the *dispatcher*. We also use real-world workload in the form of stock quotes procured from Yahoo! Finance containing a stock's daily closing prices [7] to show the performance of our system in a realistic environment.

B. Data Plane Performance

Our first set of experiments deals with evaluating the data plane performance. This includes evaluation of end-to-end latency and bandwidth efficiency of forwarding.

1) End-to-end delay: This experiment studies the delay characteristics of the aforementioned SDN-t (with fat-tree topology). We 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 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 8(a) indicates that the number of subscriptions does not significantly impact end-to-end delay. It is worth noting that, for the 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 8(a).

We would like to stress that using virtual switches does not invalidate our results, but rather gives very conservative performance bounds. In fact, we also performed micro-benchmarks to evaluate and compare the forwarding delay using an NEC hardware switch and a virtual switch to validate our results and indicate additional gain that is expected from using hardware switches in the following.

2) Comparing Hardware and Software Switches: We performed micro-benchmarks on hardware and software switches. To this end, we measured the round-trip time of packets that are looped back by the switch to the sending host. By measuring the round trip time, and with a knowledge of the propagation, transmission, and queueing delays for our specific setup, we can calculate the processing time of the hardware/software switch while varying different parameters.

In the first experiment, we vary the packet size. The packet size directly influences the transmission delay while the propagation delay should stay constant. Since the time for making a forwarding decision is also independent of the packet size, we also expect the processing delay to be nearly constant. We measured the round trip times for eight different packet sizes leading to eight frame sizes ranging from the minimum allowed Ethernet framesize up to the maximum frame size for an MTU. These sizes include the layer 2-4 headers and payload of the UDP datagram (8 byte UDP header, 20 byte IPv4 header, 26 byte Ethernet header, UDP payload). We programmed the switch with 200 flow table entries. Each entry includes a match on a different destination MAC address. We sent UDP datagrams to random IP addresses, each one associated with one of the 200 MAC addresses of the installed flows. We sent 10000 datagrams and calculated the average round trip time and standard deviation as well as the minimum round trip time and average processing delay. Figure 8(b) depicts the processing delay of the hardware and software switch. On average, the processing delay of the hardware switch is only 3.1% of the processing delay of the software switch for 72 byte frames showing that the hardware switch is significantly faster than the software switch.

We also analyze the latency of forwarding for a varying number of flow table entries. We sent frames of the minimum size of 72 bytes and varied the flow table size. The hardware switch is restricted in the maximum size of the forwarding table. Therefore, the maximum flow table size tested for the hardware switch in our experiments is 45000 entries. Flow table entries use the network destination (IPv4) address (exact match), ether type, and ingress port as match criteria. For each flow table size, we sent 10000 UDP datagrams and calculated processing delays. Figure 8(c) depicts the processing delays for different flow table sizes for both switches. As can be seen, the processing delay of the hardware switch is independent of the size of the forwarding table. For the software switch, the processing delay increases very slowly: the minimum and maximum average processing delay only differ by about 10%. Therefore, we conclude that both types of switches scale well with the number of forwarding table entries. Again, hardware support leads to much smaller forwarding latencies. From the above results, we can say that microsecond network delay per hop can be reached for our pub/sub middleware even for larger forwarding tables.

To see the performance of hardware switches w.r.t. variable-length prefixes as used in PLEROMA, we extended our evaluations with PLEROMA in [4] on a real testbed comprising a hardware whitebox Openflow-enabled switch from Edge-Core. Evaluations show that event forwarding delay is not impacted by variable-length prefixes needed by the publish/subscribe model when IPv4 multicast addresses were used with prefixes ranging between 23 and 32.

3) Bandwidth efficiency w.r.t. false positives: We define the false positive rate (FPR) as a percentage of the number of unnecessary events received to the total number of events received by subscribers. Clearly, false positives are undesirable and the aim of any publish/subscribe system is to keep them to a minimum. We observe that the longer the dz, the 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 8(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 III-C). For this reason we introduced the concept of dimension selection in this article. To portray the effectiveness of this concept, we conducted a set of experiments where the subscriptions were generated using Zipfian distribution and divided equally among the end hosts. Events for the experiments are also generated 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 8(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. This is because our dimension selection strategy provides means to increase the expressiveness of certain dimensions while ignoring others for effective filtering of events.

To ensure that the proposed PLEROMA middleware is effective in realistic scenarios, we conducted experiments with real-world stock data where the subscriptions and events were mapped according to the proposed spatial indexing scheme to enable line-rate forwarding of events in a stock-based system.

In fact, Figure 8(f) shows the benefits of employing dimension selection in such a system. We plot the benefit in terms of the percentage of false positives reduced in the system on reducing the number of selected dimensions. As can be seen in the figure, the benefit increases with the reduction in number of dimensions in this real-world scenario.

C. Control Plane Performance

We also evaluate throughput, average processing latency, and required number of flow operations in a scaled control plane. We especially compare the performances of state partitioning without load balancing (SPA) and state partitioning with load balancing (SPA-LB) approaches in order to show the effects of load balancing on this approach. We partition the event-space into 64 disjoint partitions unless otherwise specified. Also, 64 subscribers in SDN-m issue up to 200,000 subscriptions and unsubscriptions at various uniform and Poisson rates to generate load at the control plane.

1) Throughput: Figure 8(g) shows the throughput of a scaled control plane for uniform and Zipfian data respectively. In both SPA and SPA-LB, the throughput increases with increasing number of *configurators* for both distributions. Scaling out provides a lot of flexibility and can be used effectively to increase control plane throughput as shown in the graphs. Figure 8(g) also shows that, for control requests following Zipfian distribution, the throughput of SPA-LB is higher as compared to SPA. This is because, for Zipfian data, the workload is not evenly distributed among the partitions. This means that in SPA, some *configurators* may be more heavily loaded while others remain relatively idle. Since SPA does not attempt to balance this load in contrast to SPA-LB, SPA-LB clearly outperforms it. Not surprisingly, there is not much difference between the plots of SPA and SPA-LB for uniform data.

2) Average Processing Latency: In the context of our article, responsiveness is directly related to the overall time it takes for a control request to be processed by the control plane (i.e., processing latency). We define processing latency as the time elapsed from the issuance of the request by a publisher/subscriber to the time when all partial requests for this request have been processed by the control plane. In this experiment, we plot the average processing latency of control requests with increasing number of *configurators* in a scaled control plane where subscriptions and unsubscriptions are generated using both uniform and Zipfian data and sent to the *dispatcher* at a Poisson rate of 5000 requests/sec. Figure 8(h) shows that, for Zipfian data, processing latency reduces significantly with scaling. We also evaluated with uniform distribution where the plots were similar for both SPA and SPA-LB, whereas SPA-LB performs better when Zipfian data is used due to additional load balancing as can be seen in the figure. As mentioned before, with uneven load corresponding to different partitions, and a Poisson rate of incoming request, the queues formed at different *configurators* are of different lengths for SPA. This implies much longer waiting times for some requests waiting at the end of long queues resulting in a higher average processing latency. On the contrary, SPA-LB provides a possibility to migrate partitions to manage the maximum length of the waiting queues.

It is also interesting to observe the average processing latency of a control request with increased partitioning of the event-space when SPA-LB is used. The more the number of partitions, the more is the possibility of load balancing in SPA-LB, when dealing with requests following Zipfian distribution. If a *configurator* has a large partition with very high load, moving it to any other *configurator* will not balance the load. However, if the partitions are smaller, the possibility of the load being distributed among these partitions is more, which increases the flexibility of balancing the load between multiple *configurators*. Figure 8(i) shows that for Zipfian data, the average processing latency reduces significantly with increasing number of partitions up to a point. However, beyond this point further partitioning has no benefits as no further load balancing is possible for the considered workload. In fact, the graph indicates that once these benefits are no longer applicable, further partitioning may increase the average latency to some extent. This is because increased partitioning has an effect on the number of partial requests that are constructed from control requests. If the partitioning is more fine granular, the probability of a control request spanning multiple partitions is more. This means that multiple CP-configs will be affected resulting in increased number of flow operations. Figure 8(j) plots the effects of partitioning on total number of flow operations. The graph clearly shows that partitioning increases the number of flow operations significantly which can have an impact on the flow updates on the network.

3) Reducing Flow Operations: In order to reduce the number of flow operations on switches, we order control requests as discussed in the previous section. However, continuous sorting of a waiting queue at a *configurator* not only poses a significant overhead but also results in starvation for some fine-grained subscription requests that get continuously pushed down in the sorted queue. As a result, we sort only slices of contiguous subscriptions at a time and not the complete waiting queue. This set of experiments plots the number of flow operations required to process a set of 5000 subscriptions with increasing slice size. Figure 8(k) clearly shows that with increasing slice size, the number of flow operations reduces. However, Figure 8(l) shows that due to starvation of certain requests, the average latency is affected on increasing the slice size. We also plot the maximum processing latency for each slice size that contributes to increasing the average processing latency. So, there is always a trade-off between the slice size and fairness in request processing that directly affects the responsiveness to certain requests. It is important to note that a slice size of 1 implies an unsorted queue.

VI. Related Work

Various approaches to the many aspects of content-based pub/sub have been presented in literature [17], [19], [10]. A common drawback of most of these existing systems is their dependence on the application layer mechanisms to optimize pub/sub operations. Only a few systems explicitly take into account the properties of the underlying network

and its topology to organize pub/sub broker network [18], [29], [35]. 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). The recent advent of new networking technologies, such as SDN and NetFPGA, have raised some research efforts towards realizing pub/sub middleware that can support event filtering and routing within the network. LIPSIN [20] uses bloom filters in data packets to enable efficient multicast of events on the network layer. However, the expressiveness of LIPSIN is limited to topic-based pub/sub. Zhang et al. [40] address impact of SDN on the future design of pub/sub middleware and describe the realization of logically centralized pub/sub controller in a distributed manner. Nevertheless, to the best of our knowledge we are the first to thoroughly evaluate the performance of SDN-enabled content-based pub/sub middleware. Also, there has been a lot of research in the field of SDN in general, especially with regards to the limitations of TCAM (e.g., limited flow entries, flow update time, etc.) [39], [21], [15] and maintaining consistency in the data plane of software-defined networks [42], [33]. These works are orthogonal to ours and can be incorporated in our system. However, in this article, we, especially, focus on realizing the pub/sub middleware on software-defined networks.

Efficient maintenance and handling of dynamically changing subscriber interests has also been a subject of much research in overlay-based pub/sub [19], [10]. For instance, Jayaram et al. [19] propose mechanisms to efficiently handle subscriptions that change dynamically w.r.t. various parameters (such as location) by introducing the concept of parametric subscription. These methods, however, cannot be directly applied to the problems addressed in this article.

In the recent past, the emerging cloud computing model prompted the realization of pub/sub as a cloud service. In this respect, the importance of a scalable and elastic pub/sub with high throughput has been impressed upon in literature. Li et al., present an attribute-based pub/sub service, BlueDove [28], that organizes multiple servers into an overlay and achieves high throughput filtering (or matching) of events by forwarding events to be matched to the least loaded servers. Likewise, Barazzutti et al. design a scalable pub/sub service, StreamHub [1], followed by the elastic e-StreamHub [2], where a set of independent operators take advantage of multiple cores on multiple servers to perform pub/sub operations which include subscription partitioning and event filtering. It is important to note that all these systems target parallelism of event filtering and do not need to take care of concurrency control as the servers enabling concurrent filtering of events do not share any resources.

Scaling the control plane in SDN, however, involves concurrent access to the network, acting as a shared resource, and has been subject to much research in recent times [27], [6], [11], [36], [24], [14]. Levin et al. [27] explore the trade-offs of state distribution in a distributed control plane and motivate the importance of strong consistency in their work. They investigate the impact of eventual consistency on the performance of a load-balancer implemented using SDN and infer that the lack of strong consistency severely degrades application performance. To ensure strong consistency of network state between multiple controller instances, Onix [24] provides a transactional persistent database backed by a replicated state machine. However, it claims that, for applications requiring frequent network updates, dissemination of state updates using this technique yields severe performance limitations. As a result, to accommodate such applications, Onix also proposes a mechanism for obtaining eventual consistency using a memory-only DHT which has its limitations w.r.t. consistency guarantees. Similarly, Hyperflow [36] only provides guarantees of maintaining weak consistency by passively synchronizing the global network views of all controllers. This article, in contrast to the aforementioned literature, focuses on line-rate forwarding of events in the data plane and on achieving high responsiveness while ensuring strong consistency in the control plane.

VII. Conclusion

In this article we have proposed the PLEROMA middleware leveraging line-rate performance for content-based publish/subscribe in software-defined computer networks with an application-aware control that is capable of enhancing the responsiveness of the control plane while ensuring consistent changes to the data plane with low synchronization overhead even in the presence of network failures. 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) allows for expressive content filtering in the presence of hardware limitations, iii) realizes application-aware control distribution that drastically decreases the response time to control requests (up to 99% in comparison to a centralized controller) while ensuring control plane consistency, and iv) reorders control requests resulting in up to 28% less flow updates on the SDN switches.

References

[1] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J.-F. Pineau, M. Pasin, E. Rivière, and S. Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proc. of the 7th ACM Int. Conf. on Distributed Event-based Systems*, 2013.

[2] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Proc. of 34th IEEE Int. Conf. on Distributed Computing Systems*, 2014.

[3] 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 and Event-based Systems*, DEBS '16.

[4] 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.

[5] 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 '15, 2015.

[6] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust SDN control plane for transactional network updates. In *2015 IEEE Conf. on Computer Communications, INFOCOM 2015*.

[7] A. Cheung and H.-A. Jacobsen. Green resource allocation algorithms for publish/subscribe systems. In *Proc. of the 31st IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[8] 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.

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

[10] 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.

[11] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *Proc. of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13.

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

[13] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 17(2):416–429, 1969.

[14] Z. Guo, M. Su, Y. Xu, Z. Duan, L. Wang, S. Hui, and H. Jonathan Chao. Improving the performance of load balancing in software-defined networks through load variance-based synchronization. *Computer Networks*, 2014.

[15] Z. Guo, Y. Xu, M. Cello, J. Zhang, Z. Wang, M. Liu, and H. J. Chao. Jumpflow: Reducing flow table usage in software-defined networks. *Computer Networks*, 2015.

[16] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proc. of 2nd ACM SIGCOMM Wshop. on Hot Topics in SDN*, 2013.

[17] 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*. 2010.

[18] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *Proc. of the 2007 ACM Symposium on Applied Computing (SAC)*, 2007.

[19] 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.

[20] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: line speed publish/subscribe inter-networking. *ACM SIGCOMM Computer Communication Review*, 2009.

[21] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite cacheflow in software-defined networks. In *Pro. of the 3rd Workshop on Hot Topics in Software Defined Networking*, HotSDN '14.

[22] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel. The power of software-defined networking: line-rate content-based routing using Openflow. In *Proc. of the 7th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '12. ACM, 2012.

[23] B. Koldehofe, B. Ottenwälder, K. Rothermel, and U. Ramachandran. Moving Range Queries in Distributed Complex Event Processing. In *Proc. of Int. Conf. on Distributed Event-Based Systems (DEBS '12)*.

[24] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proc. of USENIX Conf. on OS Design & Implementation*, 2010.

[25] D. Kreutz, F. M. V. Ramos, P. J. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 2015.

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

[27] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proc. of Hot Topics in Software Defined Networks*, 2012.

[28] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei. A scalable and elastic publish/subscribe service. In *Proc. of IEEE Int. Parallel & Distributed Processing Symp.*, 2011.

[29] A. Majumder, N. Shrivastava, R. Rastogi, and A. Srinivasan. Scalable content-based routing in pub/sub systems. In *Proc. of the 28th IEEE Int. Conf. on Computer Communications (INFOCOM)*, 2009.

[30] V. Muthusamy and H.-A. Jacobsen. Infrastructure-free content-based publish/subscribe. *IEEE/ACM Trans. Netw.*, 2014.

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

[32] Open vSwitch. http://openvswitch.org/.

[33] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. of the ACM SIGCOMM 2012 Conf. on Applications, Technologies, Architectures, and Protocols for Comp. Comm.*, SIGCOMM '12.

[34] 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 Conf.*, 2014.

[35] M. A. Tariq, B. Koldehofe, and K. Rothermel. Efficient content-based routing with network topology inference. In *Proc. of the 7th ACM Int. Conf. on Distributed Event-Based Systems*, 2013.

[36] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for OpenFlow. In *Proc. of Internet Network Management Conf. on Research on Enterprise Networking*, 2010.

[37] G. Vaněček, Jr. BRep-Index: A multidimensional space partitioning tree. In *Proc. of 1st ACM Symp. on Solid Modeling Foundations and CAD/CAM Applications*, 1991.

[38] Y. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription partitioning and routing in content-based publish/subscribe systems. In *Proc. Of Int. Symp. on Distributed Computing*, 2004.

[39] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao. Cab: A reactive wildcard rule caching system for software-defined networks. In *Proc. of the 3rd Workshop on Hot Topics in Software Defined Networking*, HotSDN '14.

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

[41] Y. Zhao, K. Kim, and N. Venkatasubramanian. Dynatops: A dynamic topic-based publish/subscribe architecture. In *Proc. of the 7th ACM Int. Conf. on Distributed Event-based Systems*, DEBS '13, 2013.

[42] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proc. of the 12th USENIX Conf. on Networked Systems Design and Implementation*, NSDI'15.

Sukanya Bhowmik received her M.Sc. degree in Information Technology from University of Stuttgart, Germany, in 2013. She is currently pursuing her doctoral studies at the Distributed Systems research group, University of Stuttgart, Germany. Her research interests include high performance communication middleware using software-defined networking with focus on line-rate performance, bandwidth efficiency, and control plane distribution.

Muhammad Adnan Tariq received the doctoral degree from the University of Stuttgart, Germany. He is working as a postdoctoral researcher at the Distributed Systems department of the University of Stuttgart, where he is involved in the projects related to data stream processing, complex event processing, software-defined networking, and geo-distributed cloud computing, with focus on scalability, fault-tolerance, security and adaptability aspects.

Boris Koldehofe is currently managing director of the DFG Collabortive Research Centre MAKI and group head of the adaptive overlay communication group at TU Darmstadt. Formerly he worked as senior researcher and lecturer at the IPVS of the University of Stuttgart and postdoctoral researcher at the EPFL. He obtained his Ph.D in 2005 at Chalmers University of Technology. He has extensive research and teaching experience in the area of Networked and Distributed Systems and Algorithms. In particular, he has focused in the past adaptive communication middleware and distributed event-based systems.

Frank Dürr is a senior researcher and lecturer at the Distributed Systems Department of the Institute of Parallel and Distributed Systems (IPVS) at University of Stuttgart, Germany. He received both his doctoral degree and diploma in computer science from University of Stuttgart. He has given a keynote and several tutorials on Software-defined Networking (SDN) at national and international conferences and symposia. Besides SDN, his research interests include Time-sensitive (real-time) Networking (TSN), mobile and pervasive computing, location privacy, and cloud computing aspects overlapping with these topics like mobile cloud computing, edge-cloud computing, or datacenter networks.

Thomas Kohler received the M.Sc. degree in Computer Science from Augsburg University, Germany, in 2013. He is currently pursuing the Ph.D. degree at the Distributed Systems research group, University of Stuttgart, Germany. His research interests include consistency and determinism in Software-defined Networking as well as Whitebox networking hardware. In particular, his research focuses on update consistency, local switch logic and control plane distribution.

Kurt Rothermel received his doctoral degree in Computer Science from University of Stuttgart in 1985. Since 1990 he is a Professor for Computer Science at the University of Stuttgart. From 2003 to 2011 he was head of the Collaborative Research Center Nexus (SFB 627), conducting research in the area of mobile context-aware systems. He is a Director of the Institute of Parallel and Distributed Systems. His current research interests are in the field of distributed systems, computer networks, and mobile systems.