Distributed Control Plane for Software-defined Networks: A Case Study Using Event-based Middleware

Sukanya Bhowmik[†], Muhammad Adnan Tariq[†], Boris Koldehofe[∓], André Kutzleb[†] 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

Realizing a communication middleware in a software-defined network can leverage significant performance gains in terms of latency, throughput and bandwidth efficiency. For example, filtering operations in an event-based middleware can be performed highly efficiently in the TCAM memory of switches enabling line-rate forwarding of events. A key challenge in a software-defined network, however, is to ensure high responsiveness of the control plane to dynamically changing communication interactions. In this paper, we propose a methodology for both vertical and horizontal scaling of the distributed control plane that is capable of improving the responsiveness by enabling concurrent network updates in the presence of high dynamics while ensuring consistent changes to the data plane of a communication middleware. In contrast to existing scaling approaches that aim for a general-purpose distributed control plane, our approach uses knowledge of the application semantics that is already available in the design of the data plane of a communication middleware, e.g. subscriptions and advertisements in an event-based middleware. By proposing a methodology for an application-aware control distribution, we show, in the context of PLEROMA, an event-based middleware, that application-awareness is significantly beneficial in avoiding the synchronization bottlenecks for ensuring consistency in the presence of concurrent network updates and thus greatly improves the responsiveness of the control plane.

Categories and Subject Descriptors

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

Keywords

Software-defined Networking, Publish/Subscribe, Content-based Routing, Middleware, Consistency

1. INTRODUCTION

The emergence of software-defined networking (SDN) has enabled network designers to go beyond the limitations of traditional network architectures and to allow software to flexibly configure the network. With the help of standards like OpenFlow [6], 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 carry out network updates accordingly by modifying the state of network devices (i.e., switches).

The SDN technology can especially be exploited by existing middleware to enhance application performance on the data plane w.r.t. throughput, end-to-end latency and bandwidth efficiency. In particular, event-based systems can largely benefit from SDN. This is because the expressive filtering of events which was previously done at the application layer is now performed on the TCAM memory of switches (in the date plane) at line-rate [20]. Moreover, since the logically centralized controller has a global view of the underlying topology, it is possible to avoid dissemination of the same packet multiple times over the same physical link in contrast to an overlay network [14].

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

Not surprisingly, research efforts [23, 15, 10, 8] propose a distributed implementation of the control plane, which essentially operates as a logically centralized controller. A distributed control plane hosts multiple controller instances capable of performing concurrent network updates, thus improving responsiveness and throughput of the control plane. While increasing the rate at which network reconfigurations can be realized, it has been well established in literature that a distributed control plane is subject to inconsistencies [17, 3]. Inconsistencies may arise due to unsynchronized global network state views at the distributed controller instances. Every controller instance maintains a datastructure representing the view of the global network state. This implies that the network acts as a shared resource. Depending on the nature of the application, network updates are made by each controller instance based on the state of its local datastructure. Inconsistencies between the global network state maintained at each controller may lead to incorrect application-specific behavior. Performing updates based on a stale copy of the network view may result in routing loops and black holes on the data plane. Existing literature [17] shows the severity of degraded application performance in the absence of strong consistency.

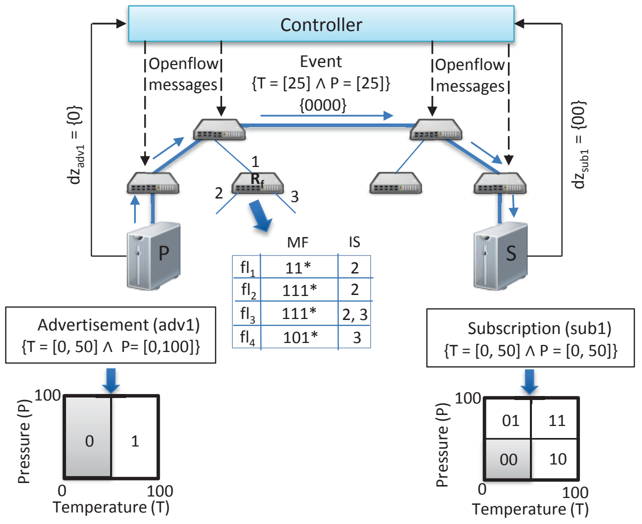
To ensure strong consistency of network state, synchronization mechanisms must be employed among all controller instances such that all network updates are coordinated. Synchronization involves state distribution and, depending on the desired level of consistency, various classical approaches available in the field of distributed systems may be used for the same [15, 3]. However, synchronization techniques always come with a cost that may compromise responsiveness to data plane requests. For instance, according to literature [15], synchronization techniques using transactional persistent database backed by a replicated state machine yields severe performance limitations for applications requiring frequent network updates.

The significant overhead in synchronization cost can be attributed to the attempt of designing a general-purpose distributed control plane capable of supporting any SDN use-case. However, as SDN can help to shape application-aware data plane, it is worth exploring how application-aware control distribution can help to reduce this overhead. In this paper, we illustrate the benefit of application-aware control distribution in the context of the very popular event-based middleware, i.e., publish/subscribe, to allow for increased responsiveness while ensuring strong consistency (in the context of control plane) even in the presence of failures. We design our system based on PLEROMA [20], an SDN-based middleware. It is worth mentioning that our concepts can apply to any application using event-based middleware.

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

2. PRELIMINARIES AND SYSTEM ARCHITECTURE

In this section, we first give an overview of PLEROMA, an SDN-based middleware with a centralized controller de-



	MF	IS
fl_1	11*	2
fl_2	111*	2
fl_3	111*	2, 3
fl_4	101*	3

Figure 1: SDN-based Pub/Sub Middleware

signed to support content-based publish/subscribe (in short pub/sub) [20]. We then present an architecture to scale such a middleware for improved control plane performance.

2.1 SDN-based Pub/Sub Middleware

A content-based pub/sub system consists of mainly two participants – publishers and subscribers – which are connected to switches in a software-defined network. Publishers specify the information they intend to publish by sending advertisements to the SDN controller. Likewise, subscribers specify information they are interested in receiving by sending subscriptions. The *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 (cf. Figure 1). 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 [6]. The aforementioned paths between publishers and their interested subscribers enable line-rate forwarding of published events through header-based matching of packets at the TCAM memory of switches on the data plane (cf. Figure 1).

The PLEROMA middleware ensures high expressiveness and low bandwidth usage by following the content-based subscription model where events are represented by attribute-value pairs. To ensure the aforementioned packet-header-based filtering of events at the data plane, we need an efficient mapping between content attributes and flow identifiers (i.e., one or more header fields that uniquely identify flow entries in the flow tables of switches). There are two steps to this mapping process.

The first step yields a binary representation of content following the principle of spatial indexing [14, 21]. The entire event-space (denoted by Ω) comprising of say d attributes is modeled as a d-dimensional space where each dimension represents an attribute. Recursive binary decomposition of Ω generates regular subspaces that serve as enclosing approximations for advertisements, subscriptions and events which are represented by binary strings called dzs. As a consequence of spatial indexing, dzs have certain characteristic properties depending upon the subspaces they represent. For instance, 1) a dz with smaller length, denoted as $|dz|$, represents a bigger subspace in Ω, and 2) a subspace represented by dz_i is *covered* (contained) by the subspace represented by dz_j, i.e., $dz_j \succ dz_i$, iff dz_j is a prefix of dz_i. For instance, in Figure 1, subspace $dz_{adv1} = \{0\}$ mapped by the advertisement {T = [0,50] $\wedge$ P = [0,100]} covers the subspace $dz_{sub1} = \{00\}$ mapped by the subscription {T = [0,50] $\wedge$ P = [0,50]}. As a result, $dz_{adv1} = \{0\}$ has shorter length and forms the prefix of $dz_{sub1} = \{00\}$.

The second step involves the mapping of the generated binary strings (dzs) to flow identifiers. 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. For instance, a subscription with $dz = \{001\}$ is converted to an IPv6 multicast address ff0e:2000:*. In Figure 1, the flow table of switch R_f shows the flow fields that are relevant for our middleware. This constitutes the match field (MF), in our case an IPv6 multicast address representing a dz, e.g., 11* in fl_1, and the instruction set (IS), which specifies the port through which an event should be forwarded on account of a match, e.g., 2 in fl_1. 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 the outgoing port of the instruction set (IS) on account of a match.

An efficient approach to topology reconfiguration is central to pub/sub on SDN. For this purpose, a spanning tree (comprising switches) is maintained to account for an acyclic dissemination structure on which paths are embedded between publishers and subscribers by installing appropriate flows (filters) on switches along these paths. A path is nothing but a sequence of switches (denoted as R) on which flows are deployed to ensure connectivity between the publisher and the subscriber. The flows to be deployed on a switch depend largely on the already existing pub/sub flows on that switch and as a result it is important for the controller to identify the state of each switch in the network.

In more detail, the network state is represented by network configuration that consists of (i) all switches constituting the network, (ii) all links connecting the switches in a spanning tree to account for an acyclic dissemination structure, and (iii) all pub/sub flows deployed on each switch. In general, the network configuration is maintained both at the data plane and the control plane of a software-defined network. The network configuration at the data plane (denoted as DP-config) is maintained implicitly as a result of pub/sub flows deployed in the TCAM memory of hardware switches. On the other hand, the control plane network configuration (denoted as CP-config) is maintained by the controller and serves as a reflection of DP-config. 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.

In order to understand the decision-making process to determine flow changes on a switch, it is important to note the containment relation between flows w.r.t. a single switch. A flow fl_i covers (or contains) another flow fl_j, denoted by $fl_i \succ fl_j$, if the following two conditions hold: (i) the dz associated with the destination IP address in the match field of fl_j is covered by the dz of fl_i, and (ii) the out ports to which a packet matching fl_j is forwarded are subset of those specified in the IS of fl_i. Likewise, a partial containment relation ($\gtrapprox$) can be defined between flows of a switch (or flows to be installed on a switch). A flow fl_i partially covers (or contains) another flow fl_j, denoted by $fl_i \gtrapprox fl_j$, if dz associated with the match field of fl_i covers dz of fl_j, but not all the out ports used for forwarding packets matching fl_j are listed in the IS of fl_i. For example, in Figure 1, $fl_1 \succ fl_2$ and $fl_1 \gtrapprox fl_3$, whereas fl_4 is unrelated to all other flows in the flow table of R_f. Based on these containment relations, flows are either installed or modified or no actions are taken w.r.t a switch while establishing the routing path. Assume that a new flow fl_n is to be installed on a switch, then decisions are taken as follows : 1) If an existing flow fl_e already covers fl_n , then no further actions are taken. This also includes the special case where fl_n is identical to fl_e. 2) If an existing flow fl_e is covered by fl_n, then fl_n is added and fl_e is deleted from the flow table as it is redundant. 3) If fl_n is partially covered by an existing flow fl_e, then fl_n should be added with higher priority and should include the out ports in the IS of fl_e. A higher priority ensures that if a packet has multiple matches in the flow table, it would be matched against and follow the IS of the flow with highest priority. 4) If fl_n partially covers the existing flow fl_e, then besides adding fl_n to the flow table, fl_e should be updated to include out ports used by fl_n and to hold higher priority than fl_n.

2.2 Distributed Control Plane

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

Figure 2 depicts a two-tiered architecture in the control plane; a *dispatcher* collects control requests from publishers and subscribers in a software-defined network, and a set of components, known as *configurators* (denoted by C), processes these requests and carries out network updates ac-

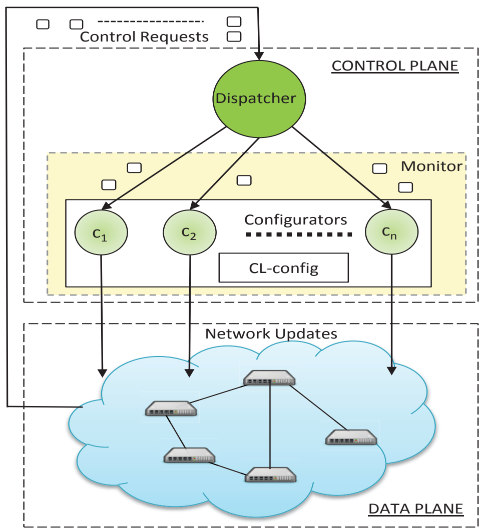
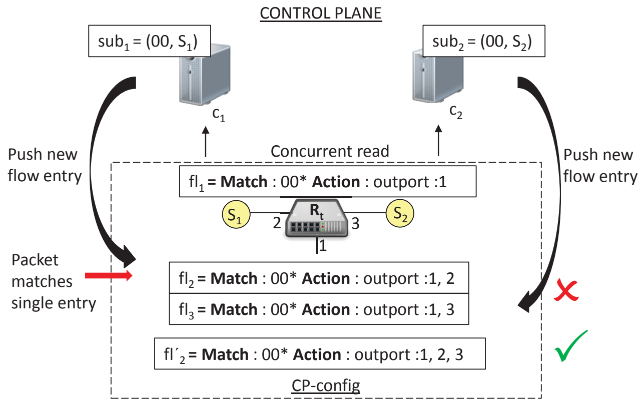


Figure 2: Distributed Control Plane



CONTROL PLANE
$sub_1 = (00, S_1)$
$sub_2 = (00, S_2)$
c_1
c_2
Push new flow entry
Concurrent read
Push new flow entry
fl_1 = Match : 00* Action : outport :1
S_1
R_t
S_2
Packet matches single entry
fl_2 = Match : 00* Action : outport :1, 2
fl_3 = Match : 00* Action : outport :1, 3
fl'_2 = Match : 00* Action : outport :1, 2, 3
CP-config

Figure 3: Control Plane Inconsistency

cordingly. SDN allows the *dispatcher* and the *configurators*, residing in the control plane, to acquire a global view of the entire network and configure it as needed. The *dispatcher* serves as the entry point to the control plane. It collects all data plane control requests and forwards them to the *configurators*. The *configurators* serve as the main workers that are capable of modifying the state of every switch in the network. Each of them receive control requests forwarded by the *dispatcher* and process them as described in Section 2.1, resulting in concurrent updates to the CP-config and DP-config. The *monitor* is an additional component connected to the *configurators* and the *dispatcher*. It plays an important role in maintaining load statistics of each *configurator*, which contributes to improved system performance. The relevance of the *monitor*, in context of our designed middleware, will be explained in details later in this paper.

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

In the subsequent sections, we first discuss control plane consistency in the context of pub/sub middleware and identify conflicting actions that may induce inconsistencies (cf. Section 3). Afterwards, we present approaches for vertical and horizontal scaling of the control plane that ensure consistency by enabling concurrency control for conflicting actions with low synchronization overhead (cf. Section 4).

3. CONTROL PLANE CONSISTENCY IN PUB/SUB

In general, two important problems have to be addressed to ensure control plane consistency in a 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 this section, we strictly focus on the first problem and address the second problem in Section 5. 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 2, 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.

Figure 3, depicts an example of a simple case of false negatives at a subscriber due to the interleaving of sequences of operations constituting two actions and belonging to two *configurators*. Let us suppose that two overlapping subscription requests sub_1=$\{00\}$ from subscriber S_1 and sub_2=$\{00\}$ from subscriber S_2 are simultaneously dispatched to *configurators* c_1 , $c_2 \in C$ respectively. Both follow the aforementioned request handling process and perform actions on relevant switches. We specifically focus on the terminal switch R_t which already has a flow, fl_1, to match event packets for subspace $\{00\}$ (cf. Figure 3). We consider a case where both *configurators* perform concurrent read on this switch in CP-config. On reading the state, c_1 and c_2 independently decide on required flow updates and replace the existing flow (fl_1) by adding two new flows fl_2 and fl_3 respectively (cf. Figure 3). As a consequence, there now exists two flows with the exact same match field but with different IS at R_t. Since deploying flows on CP-config implies deploying them on DP-config, now, if an event packet lying in subspace $\{00\}$ arrives at R_t in the data plane, it

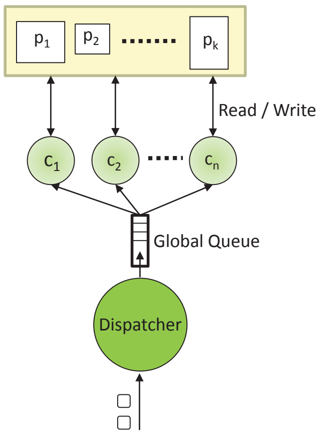


Figure 4: Shared Everything Approach

follows the instruction set of either fl_2 or fl_3, but never both as the matching of a packet at a switch is terminated as soon as the first match is found. In either case, one of the two subscribers is affected by false negatives compromising correctness of the system.

Clearly, false negatives at a subscriber in Figure 3 occurred because flows fl_2 and fl_3 concurrently added by c_1 and c_2 are in aforementioned partial flow containment relation (i.e., $\gtrapprox$), which essentially results in updating the same flow in R_t. In general, concurrent updates of the flows with containment relations (i.e., $\succ$ or $\gtrapprox$) have an effect of one of the updates being overwritten by the other.

While understanding the above mentioned concurrency issues, we identify conflicting actions in a SDN-based pub/sub middleware.

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 ($\gtrapprox$).*

In order to ensure consistency, all conflicting actions must be serialized.

4. SCALING APPROACHES

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

4.1 Shared Everything Approach

The shared everything approach (SEA) works on the principle that all *configurators* share CP-config among themselves. This implies that all of them read from as well as write to every switch in CP-config. Section 3 explained the undesirable consequences of such concurrent access of shared state which means that the SEA approach must employ certain additional mechanisms for concurrency control. SEA uses a locking mechanism that allows a *configurator* to acquire exclusive access on CP-config at various granularity levels. This means that no other *configurator* can access the locked part of CP-config unless the *configurator* holding the lock relinquishes it. Locks can be held at different levels of granularity in CP-config. In the absence of application knowledge, a plausible strategy is to assign locks at the granularity of switches. For example, with the advent of a subscription, a *configurator* can determine the paths between the associated subscriber and all relevant publishers, acquire locks on all switches in these paths, read, decide on flow changes, deploy changes on the switches, and finally release the locks. Acquiring locks at switch level, however, would imply that no other *configurator* can execute an action on a locked switch even if its action does not conflict with the current action being executed. So, with respect to our definition of conflicting actions (cf. Definition 1), locking at switch-level may not be ideal.

Here, we propose an application-aware method that uses knowledge of advertisements and subscriptions to control the granularity at which CP-config can be accessed concurrently. Since the dzs representing the subscriptions/advertisements (in control requests) are directly mapped to flows added to switches (cf. Section 2), two control requests where one dz covers or is identical to the other (overlapping subspaces in Ω) yield flows related ($\succ$ and $\gtrapprox$) to each other. This means that concurrent processing of overlapping control requests at a switch will result in conflicting actions and must be ordered sequentially. Control requests with non-overlapping subspaces in Ω, however, can undergo concurrent processing without any issues. For example, concurrent processing of two subscriptions $\{00\}$ and $\{000\}$ which results in state modification of at least one common switch will lead to incorrect system behavior as $\{00\} \succ \{000\}$. However, two unrelated subscriptions, $\{00\}$ and $\{11\}$, can be processed concurrently by two *configurators* without any issues as processing will not yield any related flows. This directly leads us to the idea of partitioning the event-space in a disjoint way such that flows corresponding to different partitions in Ω are maintained in separate CP-configs. This enables concurrent processing of disjoint control requests that operate on different CP-configs. Locking would only be required at the level of a CP-config to ensure sequential processing of overlapping control requests.

So, we divide the event-space (Ω) into multiple disjoint, continuous partitions. A partition is nothing but a subspace in Ω and may be represented in the same way, i.e., by a dz. Disjoint event-space partitioning may yield equal or unequal partitions depending on the partitioning criteria. However, it is important to note that, in any case, the partition set, denoted by 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 [24, 25] and will not be discussed further in this paper. 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 paper, a CP-config ($cp_i \in CP$) is considered to be synonymous with a partition ($p_i \in P$).

We focus on a SEA approach where locking is carried out at the level of a $cp \in CP$ which essentially means locking a set of flows across all switches that correspond to a partition in Ω. This ensures concurrent access of unrelated flows on the same switch. Each *configurator* maintains a pointer to each CP-config/ partition. Figure 4 illustrates the same with n *configurators*, c_1,.., c_n, operating on k partitions, p_1,..., p_k. SEA ensures serial processing of requests within a single partition while allowing concurrency otherwise.

The *dispatcher* collects all control requests from the data plane and adds them to a global queue accessible to all *configurators*. But before adding them, it performs an additional step to prepare the requests for further processing. Let us denote the dz representing a control request by dz_c and that representing any partition 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 adds the request to the global queue as the request is contained by one partition and affects a single CP-config. However, the second scenario portrays a case where the control request subspace spans more than a single partition. Under such circumstances, the *dispatcher* splits up the request into multiple dzs depending upon the nature of the partitions and adds these partial requests to the queue. For example, if we consider a system with 4 partitions – 00, 01, 10, 11 – and a request with dz {001101} arrives at the *dispatcher*, the *dispatcher* immediately adds the request to the global queue as $\{00\} \succ \{001101\}$. However, if the request corresponds to {0}, the *dispatcher* first splits it up into two partial requests {00}, {01} and then adds them to the queue as $\{0\} \succ \{00, 01\}$. Consequently, two CP-configs are reconfigured for this single request. Processing of a control request is considered complete only when all its partial requests have been processed.

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

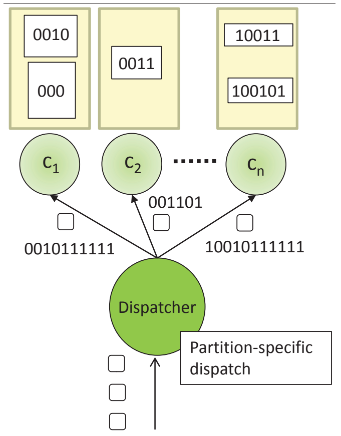


Figure 5: Shared Nothing Approach

The advantages of using such an approach for increased responsiveness in a vertically scaled control plane are significant. This is because vertical scaling works with a shared memory architecture where sharing of multiple CP-configs does not cause much overhead. However, horizontal scaling implies repeated remote access of multiple CP-configs by every *configurator*. This involves transfer of large amount of data over the control network resulting in severe performance limitations. Not surprisingly, SEA cannot match up to the requirements of a horizontally scaled control plane. To overcome these limitations and exploit the benefits of horizontal scaling, we propose the shared nothing approach which is the subject of discussion in the remaining part of this section.

4.2 Shared Nothing Approach

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

Figure 5 depicts a middleware where each *configurator* has one or more partitions assigned to it and each partition is represented by its corresponding dz. Such a representation enables the direct mapping of advertisements/subscriptions (represented by one or more dzs) to partitions. For example, two partitions {000}, {0010} have been assigned to $c_1 \in C$, implying that c_1 only maintains CP-configs for these two partitions, affecting flows related to these subspaces.

4.2.1 Topology Reconfiguration

The *dispatcher* plays a significant role in this approach. It maintains a map of the *configurators* and their associated partitions and performs partition-specific dispatch of control requests. Again, the *dispatcher* first prepares a control request for further processing by splitting it into partial requests, if necessary, depending on the partitions (cf. Section 4.1). This guarantees the mapping of a request to a single partition enabling the *dispatcher* to directly forward a request to a *configurator* responsible for the corresponding partition. For example, as per Figure 5, if a request corresponds to {00}, the *dispatcher* first splits it up into three requests {000}, {0010}, {0011} and then dispatches them to c_1 and c_2 as $\{00\} \succ \{000, 0010, 0011\}$. Consequently, all three CP-configs are reconfigured for this single request.

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

The shared nothing 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 the responsiveness of the control plane to control requests. For this reason, adaptive load balancing among *configurators* bears considerable significance and features as the subject of discussion in the remaining part of this subsection.

4.2.2 Adaptive Load Balancing

In the face of a dynamic incoming workload, an adaptive policy is central to the load balancing approach. We identify load of a *configurator* at a given time by request queue lengths of all partitions assigned to it. A request queue, specific to a partition (say, 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,

$$l_i \quad = \quad \sum_{j=1}^{m} QL_j \tag{1}$$

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 is migrated to a *configurator* with current minimum load. This implies that the task of processing all current and future requests for the migrated partitions now lies with the newly chosen *configurator*. An overload detection is carried out by the *monitor* component. The *monitor* periodically collects load information of every *configurator* and hence can easily identify an overload condition. With every periodic collection, the *monitor* calculates the average queue length at 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,

$$\frac{l_i}{l_{avg}} > threshold \tag{2}$$

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

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

It should be noted that SNA is suitable for both vertical as well as horizontal scaling. However, in the case of vertical scaling, it may perform worse than SEA in the presence of fluctuating unevenly distributed workload. Subject to such workload, adaptive load balancing of SNA will always be outperformed by the implicit optimal load balancing achieved in SEA. This is further confirmed by our evaluation results (cf. Section 7).

5. 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* on to 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 a

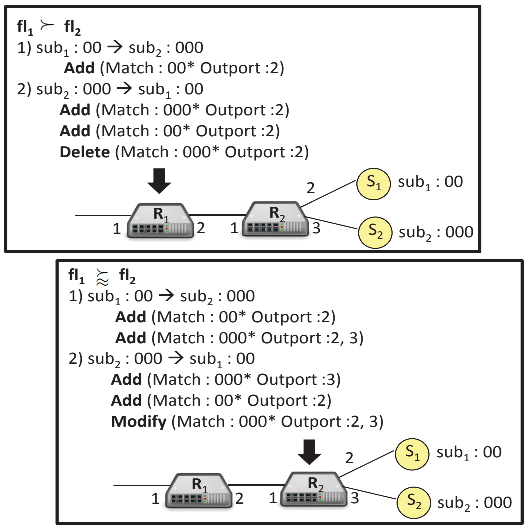

$fl_1 \succ fl_2$
1) sub_1 : 00 → sub_2 : 000
Add (Match : 00* Outport :2)
2) sub_2 : 000 → sub_1 : 00
Add (Match : 000* Outport :2)
Add (Match : 00* Outport :2)
Delete (Match : 000* Outport :2)
R_1
R_2
1
2
1
3
2
S_1
sub_1 : 00
S_2
sub_2 : 000
$fl_1 \gtrapprox fl_2$
1) sub_1 : 00 → sub_2 : 000
Add (Match : 00* Outport :2)
Add (Match : 000* Outport :2, 3)
2) sub_2 : 000 → sub_1 : 00
Add (Match : 000* Outport :3)
Add (Match : 00* Outport :2)
Modify (Match : 000* Outport :2, 3)
R_1
R_2
1
2
1
3
2
S_1
sub_1 : 00
S_2
sub_2 : 000

Figure 6: Reducing Flow Operations

given timeout. The flow monitoring functionality introduced by OpenFlow version 1.4 can be efficiently used to allow a *configurator* to be notified by a switch about flow operations (addition/modification/deletion) performed on its tables. Such switch notifications can serve as acknowledgments of completed flow table updates. On receiving an 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.

6. REDUCING FLOW OPERATIONS

Increasing responsiveness of the control plane to control requests also increases the rate at which network updates are pushed on to the switches by multiple *configurators*. With today's hardware switches supporting around 40-50 flow-table updates per second [11], 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 that ordering would benefit from are complete containment, i.e., $fl_i \succ fl_j$, and partial containment, i.e., $fl_i \gtrapprox fl_j$.

Referring to the two subscriptions in the above example and their relations, we first look at complete containment between flows. The following updates would be done on a switch depending on the order in which the two subscriptions are processed. 1) If sub_i is processed before sub_j, sub_i first produces one add flow (fl_i) operation on the switch. When sub_j is processed, it does not produce any other flow updates on the switch as fl_i fully covers all events that need to be forwarded in response to sub_j. 2) If sub_j is processed before sub_i, sub_j also produces one add flow (fl_j) operation on the switch. After this, when sub_i is processed another flow (fl_i) add operation has to be performed to cover forwarding of all events matching sub_j and also those matching sub_i but not sub_j. Also, a delete operation has to be performed on fl_j as it is now redundant. Given the limitations of the flow table size on a switch, redundant flows cannot be afforded. This clearly indicates that the first ordering yields two operations less as compared to the second. Figure 6 illustrates the above discussion with an example where the ordering of two subscriptions sub_1 ($\{00\}$) and sub_2 ($\{000\}$) that would independently produce fl_1 and fl_2 yield different number of operations on switch R_1 as $fl_1 \succ fl_2$.

Let us now consider the second relation of partial containment between the flows. Again, we look at the number of operations required on ordering sub_i and sub_j 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. This is again illustrated in Figure 6 and this time the operations w.r.t. both orders are tracked on switch R_2 where a partial containment relation between fl_1 and fl_2 ($fl_1 \gtrapprox fl_2$) occurs.

It is important to note that the reordering of subscriptions does not have an impact on the correctness of the system. This is because, no matter how processing of requests is ordered, the final set of flows deployed on the switches is always the same. In Figure 6, 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 7).

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

7. PERFORMANCE EVALUATIONS

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

7.1 Experimental Setup

We scale the control plane both vertically and horizontally on a test-bed consisting of a small local area network which includes a cluster of physical machines capable of hosting each component of the proposed architecture. Vertical scaling is realized by hosting multiple *configurators* on a single physical machine with 4 cores, 3.4 GHz processor and 8 GB of RAM. On the other hand, horizontal scaling is achieved by hosting multiple *configurators* on multiple physical machines where each machine in the cluster has 4 cores, 3.4 GHz processor, and 8 GB of RAM. Two separate machines host the *dispatcher* and the *monitor*.

The aforementioned setup deals with the control plane. In order to realize the data plane, our setup uses Mininet [16], a very prominent tool for emulating software-defined networks. Mininet is an extremely flexible tool that allows to conduct experiments with different types of topology and application traffic. Since we use a very large fat-tree topology with 64 hosts (publishers and/or subscribers) and 102 OpenFlow-compliant switches for all our experiments, Mininet is an ideal choice. It is also important to note that since our evaluations focus on control plane performance, they are independent of a real or emulated data plane.

We use a content-based schema that consists of upto 10 attributes for our event-space (Ω), where the domain of each attribute varies in the range $[0, 1024]$. Experiments are performed using two different models of data distribution for generating control requests ((un)advertisement/(un)subscription). The uniform model generates control requests uniformly over Ω, whereas, the interest popularity model chooses 8 hotspot regions around which control requests are generated using the widely used zipfian distribution. The rate at which control requests are sent by the participants (i.e., publishers and subscribers connected to a SDN network) to the dispatcher also follows two models of distribution, i.e., uniform and poisson. A uniform rate implies that the occurrences of incoming requests at the *dispatcher* are distributed uniformly on an interval of time. However, poisson rate involves a fluctuating workload while maintaining an average rate of incoming requests at the *dispatcher* within a given interval of time. So, there may be bursts of incoming requests from time to time along with lull periods to ensure an average rate at the *dispatcher*.

7.2 Vertical Scaling

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

The first set of experiments measures the maximum rate at which the control plane can process control requests, i.e., throughput, with increasing number of *configurators*. It is important to note that control requests may be further broken down into partial requests to contain them in different partitions. A control request is considered to be processed only when all its partial requests have been processed. Figure 7(a) and 7(b) show that, with increasing number of *configurators*, the throughput of the control plane increases significantly for both uniform and zipfian data till the control plane is scaled up by 4 *configurators* for both approaches. Not surprisingly, as the *configurators* are hosted by a machine with a 4-core architecture, there is not much benefit if the control plane is scaled beyond 4. Figure 7(b) shows that, for control requests following zipfian distribution, the throughput of SEA is higher as compared to SNA-LB. This is because, for zipfian data, the workload is not evenly distributed among the partitions. This means that in SNA-LB, some *configurators* may be more heavily loaded while others remain relatively idle. Even though SNA-LB tries to balance this load, it does so only after a threshold limit is crossed while SEA ensures that no *configurator* is ever idle unless there are no more requests to process. SEA implies optimal load balancing among *configurators*. In case of uniform data, where all partitions are equally loaded, Figure 7(a) shows that the performance of SEA is slightly worse than the other as once the benefits of load balancing are not visible, the additional synchronization overhead required in SEA renders it less effective as compared to SNA-LB.

In the context of our paper, responsiveness is directly related to the overall time it takes for a control request to be processed by the control plane (i.e., processing latency). We define processing latency as the time elapsed from the issuance of the request by a publisher/subscriber to the time when all partial requests for this request have been processed by the control plane. In this experiment, we plot the average processing latency of control requests with increasing number of *configurators* in a vertically scaled control plane. We show a comparison of both the approaches when subscription and unsubscription requests are generated using both uniform and zipfian distributions and are sent by the subscribers to the *dispatcher* at a poisson rate of 2500 requests/sec. Figure 7(c) and Figure 7(d) show that, for both uniform and zipfian data and for both approaches, the average processing latency reduces significantly with increasing number of *configurators* till 4 *configurators*. Again, scaling beyond 4 *configurators* may not have any benefits due to the reason mentioned above. Figure 7(c) suggests that there is not much difference in performance between the approaches for uniform data as all partitions get similar amount of workload. This implies that all *configurators* get similar amount of workload in SEA and SNA-LB. However, the difference in benefits between the approaches is visible for zipfian data and as a result we focus on comparing their performances by zooming the graph in Figure 7(d). In general, with dynamically changing incoming workload, SEA performs better as

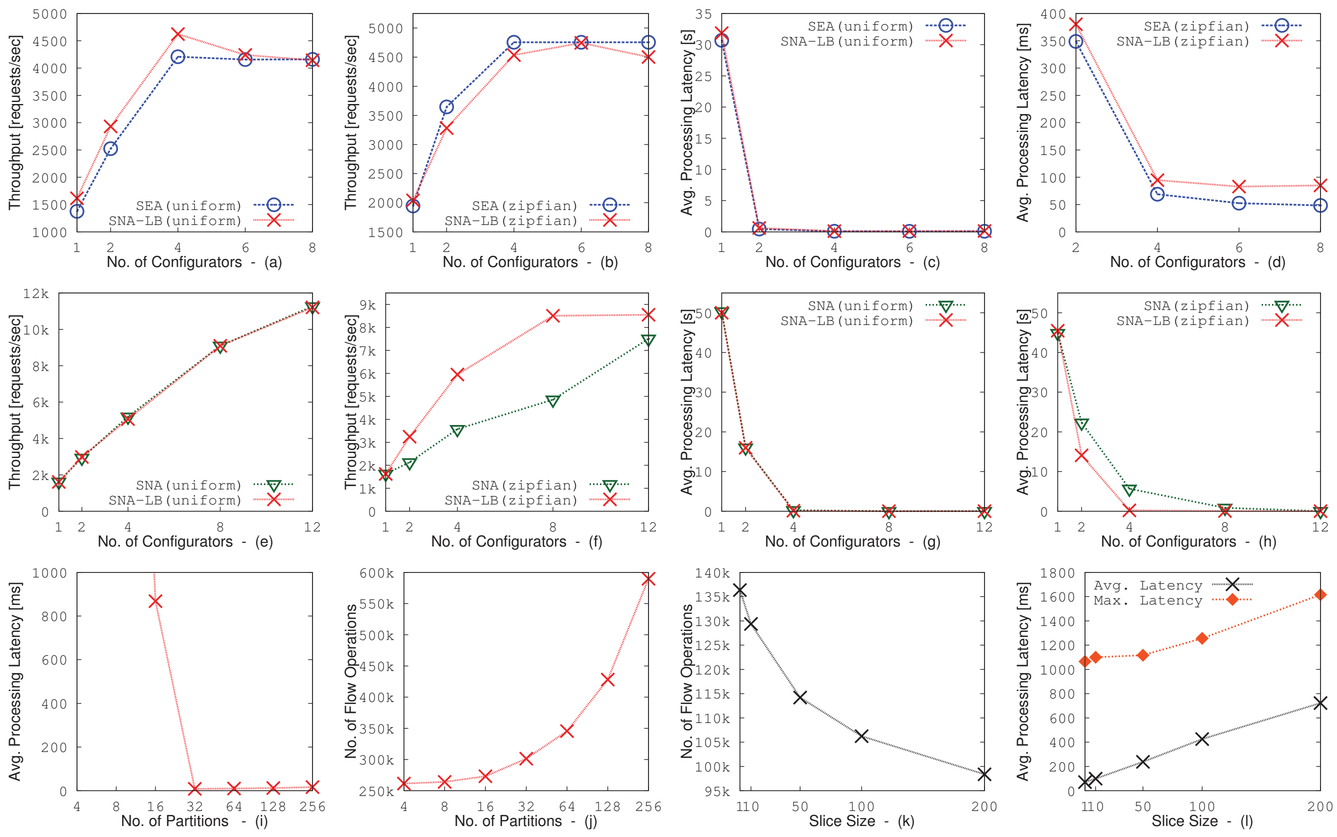


Figure 7: Performance Evaluations

compared to SNA-LB as it ensures optimal load-balancing. As mentioned before, with uneven load corresponding to different partitions, and a poisson rate of incoming request, the queues formed at different *configurators* are of different lengths for SNA-LB. This implies much longer waiting times for some requests waiting at the end of long queues resulting in a higher average processing latency.

7.3 Horizontal Scaling

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

Figure 7(e) and Figure 7(f) show the throughput of a horizontally scaled control plane for uniform and zipfian data respectively. In both SNA and SNA-LB, the throughput increases with increasing number of *configurators* for both distributions as a horizontally scaled setup does not suffer from the limitations of a vertically scaled one in terms of number of cores. Scaling out provides a lot of flexibility and can be used effectively to increase control plane throughput as shown in the graphs. Not surprisingly, there is not much difference between the plots of SNA and SNA-LB for uniform data. However, the benefits of load balancing are again visible for zipfian data where SNA-LB outperforms SNA.

We also conducted experiments which measure average processing latency of control requests with increasing number of *configurators* when subscriptions and unsubscriptions are generated using both uniform and zipfian data and sent to the *dispatcher* at a poisson rate of 5000 requests/sec. Figure 7(g) and Figure 7(h) show behavior similar to that obtained in vertical scaling where the average processing latency reduces significantly with scaling. The plots for uniform distribution are similar for both SNA and SNA-LB, whereas SNA-LB performs better when zipfian data is used due to additional load balancing. This means that SNA-LB provides a possibility to migrate partitions to manage the maximum length of the waiting queues, whereas SNA has no such possibility, because of which the average processing latency for SNA-LB is lower than that of SNA.

It is also interesting to observe the average processing latency of a control request with increased partitioning of the event-space when SNA-LB is used. The more the number of partitions, more is the possibility of load balancing in SNA-LB, when dealing with requests following zipfian distribution. If a *configurator* has a large partition with very high load, moving it to any other *configurator* will not balance the load. However, if the partitions are smaller, the possibility of the load being distributed among these partitions is more, which increases the flexibility of balancing the load between multiple *configurators*. Figure 7(i) shows that for zipfian data, the average processing latency reduces significantly with increasing number of partitions upto 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 7(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.

7.4 Reducing Flow Operations

In order to reduce the number of flow operations on switches, we order control requests as discussed in the previous section. However, continuous sorting of a waiting queue at a *configurator* not only poses a significant overhead but also results in starvation for 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 7(k) clearly shows that with increasing slice size, the number of flow operations reduces. However, Figure 7(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.

8. RELATED WORK

Various approaches to the many aspects of content-based pub/sub have been presented in literature [5, 12, 19, 21, 4, 13, 7, 22]. However, built on top of overlay networks, these systems cannot take advantage of the properties of the underlying physical topology and lack in performance, in terms of throughput, end-to-end latency and bandwidth efficiency as compared to network layer implementations of communication protocols. Realizing the potential of an implementation on the network layer, middleware such as DDSFlex [9] and PLEROMA [20] build efficient pub/sub systems by using the capabilities of SDN. We have already discussed the PLEROMA middleware at length in this paper and have identified the inherent scalability limitations of a single controller instance with respect to reconfiguration efforts in the face of high dynamics. PLEROMA also introduces the concept of multiple controllers responsible for separate administrative domains in its design where each controller performs reconfiguration in its designated domain. However, within a domain, the middleware may suffer from the same aforementioned limitations. Also, in the presence of inter-domain communication, the responsiveness of the control plane may suffer. Efficient maintenance and handling of dynamically changing subscriber interests has also been a subject of much research in overlay-based pub/sub middleware [4, 13, 7]. For instance, Jayaram et al. [13] propose mechanisms to efficiently handle subscriptions that change dynamically w.r.t. various parameters (such as location) by introducing the concept of parametric subscription. These methods, however, cannot be directly applied to the problem addressed in this paper.

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 [18], 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], 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. Since StreamHub only supports scale out, Barazzutti et al., further propose e-StreamHub [2], an elastic pub/sub which is capable of scaling in and scaling out depending on the load observations of the system. 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. Levin et al. [17] 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 [15] 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 [23] only provides guarantees of maintaining weak consistency by passively synchronizing the global network views of all controllers. On the other hand, Botelho et al. [3] show that by using a classical state machine replication technique the cost of coordination to guarantee strong consistency may become bearable for certain SDN applications, but not in general. This paper, in contrast to the aforementioned literature, not only focuses on line-rate forwarding of events in the data plane but also on achieving high responsiveness while ensuring strong consistency on the control plane.

9. CONCLUSION

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

10. 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] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *Second European Wshop. on Software Defined Networks, EWSDN*, 2013.

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

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

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

[7] 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.

[8] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed SDN controller. In *Proc. of 2nd ACM SIGCOMM Wshop. on Hot Topics in Software Defined Networking*, 2013.

[9] A. Hakiri, P. Berthou, P. P. Patil, and A. Gokhale. Towards a publish/subscribe-based open policy framework for proactive overlay software defined networking. (ISIS-15-115), 2015.

[10] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proc. of 1st Wshop. on Hot Topics in Software Defined Networks*, 2012.

[11] 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.

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

[13] 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.

[14] B. Koldehofe, F. Dürr, and M. A. Tariq. Tutorial: Event-based systems meet software-defined networking. In *Proc. of the 7th ACM Int. Conf. on Distributed Event-based Systems*, DEBS '13.

[15] 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 9th USENIX Conf. on Operating Systems Design and Implementation*, 2010.

[16] 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.

[17] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proc. of 1st Wshop. on Hot Topics in Software Defined Networks*, 2012.

[18] 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.

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

[20] 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.

[21] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, and K. Rothermel. Meeting subscriber-defined QoS constraints in publish/subscribe systems. *Concurrency and Computation: Practice and Experience*, 2011.

[22] 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.

[23] 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.

[24] 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.

[25] 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.