LazyCtrl: A Scalable Hybrid Network Control Plane Design for Cloud Data Centers

Kai Zheng, Lin Wang, Baohua Yang, Yi Sun, and Steve Uhlig

Abstract—The advent of software defined networking enables flexible, reliable and feature-rich control planes for data center networks. However, the tight coupling of centralized control and complete visibility leads to a wide range of issues among which scalability has risen to prominence due to the excessive workload on the central controller. By analyzing the traffic patterns from a couple of production data centers, we observe that data center traffic is usually highly skewed and thus edge switches can be clustered into a set of communication-intensive groups according to traffic locality. Motivated by this observation, we present LazyCtrl, a novel hybrid control plane design for data center networks where network control is carried out by distributed control mechanisms inside independent groups of switches while complemented with a global controller. LazyCtrl aims at bringing laziness to the global controller by dynamically devolving most of the control tasks to independent switch groups to process frequent intra-group events near the datapath while handling rare inter-group or other specified events by the controller. We implement LazyCtrl and build a prototype based on Open vSwitch and Floodlight. Trace-driven experiments on our prototype show that an effective switch grouping is easy to maintain in multi-tenant clouds and the central controller can be significantly shielded by staying "lazy", with its workload reduced by up to 82%.

Index Terms—Software defined networks, network control, data center, cloud computing.

1 Introduction

Public clouds are becoming increasingly popular due to their *pay-as-you-go* model, which attracts many small and medium business. Some of them, thanks to their success, have grown very large, each containing hundreds thousand of servers and hosting up to millions of virtual machines [1]. To support flexible and efficient inter-node communication in these large-scale cloud data centers, researchers have proposed many novel designs (*e.g.*, [2], [3]) for data center networks to replace traditional tree-based architectures. However, the routing and forwarding protocols used in most designs are restricted to very specific deployment settings, leading to inflexible configuration and management. The situation has been revolutionized by Software Defined Networking (SDN), where the control plane, separated from the data plane, is implemented with a logically centralized controller. As a result, when adopting SDN, flow-based polices can be conveniently applied to achieve fine-grained control over the data center network.

While flow-based centralized control has been recently employed in several proposals for traffic management in data center networks [4], [5], [6], the excessive coupling of central control and complete visibility has brought many scalability challenges to both the network control and data planes in large-scale data centers. On the one hand, having the controller to set up all flows would bring too much workload to the controller and such centralized bottlenecks are difficult to scale. On the other hand, maintaining visibility of all flows in a large-scale network can require hundreds of thousands of flow table entries at each switch, which is far from practical for commodity switches.

- *Kai Zheng is with Huawei Technologies.*
- *Lin Wang is with SnT, University of Luxembourg.*
- *Yi Sun are with the Institute of Computing Technology, Chinese Academy of Sciences.*
- *Baohua Yang are with IBM Research.*
- *Steve Uhlig is with Queen Mary University of London.*

Manuscript received April 19, xxxx; revised September 17, xxxx.

1.1 Bringing Laziness to the Controller

It has been demonstrated that full control and visibility over all flows are not always necessary and devolving some control authority to the data plane by proactively suppressing frequent events can result in better scalability in software defined data center networks [7]. However, the right granularity of flows to be handled by the controller is still not clear (or hard to define). In this paper, we advocate a new solution for control devolvement in data center networks based on traffic locality. Our idea stems from the observation that traffic distribution in data centers (especially those with multi-tenancy support) could be highly skewed, *i.e.*, frequent communications are more likely to take place inside certain small groups of hosts. As a result, it is possible to shield the global controller from many frequent events inside these groups if distributed control mechanism is applied independently in each of the groups.

We propose LazyCtrl, a hybrid network control plane design for large-scale data centers, which seeks to bring *laziness* to the global controller. In the LazyCtrl design, edge switches are grouped dynamically according to their communication affinity. The central controller devolves the coarse-grained control for frequent intra-group events to each switch group while handling infrequent inter-group and other specified (fine-grained) control tasks by itself. Each switch group autonomously carries out *distributed control* within the group, keeping the intra-group packets in the data plane. The controller groups the switches in such a way that the size of each group is as large as possible to exhaust switches' memory (such as TCAMs) capacity while inter-group traffic is minimized to support the laziness of the controller.

We have completed a full implementation of LazyCtrl based on Open vSwitch and the Floodlight OpenFlow controller. Experiments on our prototype with both real and synthetic traffic traces show that an effective switch grouping is easy to maintain in multi-tenant clouds and the hybrid control design highly reduces the workload of the controller and provides lower delay in packet forwarding. As expected, the laziness we introduced to the controller decouples centralized control and complete visibility and consequently scale the system much better compared with totally centralized designs.

Section 2 reveals some observations that motivate our design. Section 3 presents the LazyCtrl architecture with design details. Section 4 presents our implementation, followed by the performance evaluation in Section 5. Section 7 concludes the paper.

1.2 Related Work

Ethernet stands as one of the most widely used networking technologies today due to its *plug-and-play* semantics such as automatic host location learning and flat addressing, which can highly simplify many aspects of network configuration and ensure service continuity. However, replying on network-wide dissemination of per-host information makes Ethernet-based solutions difficult to scale and forcing paths to comprise a spanning tree introduces substantial inefficiencies. In contrast, IP networks can easily scale to large networks but require massive effort to configure and manage.

As a promising solution for building large-scale data center networks, network overlay can exploit the advantages of both Ethernet and IP networks. An overlay network in a data center consists in creating a dynamic mapping between the overlay (virtual) network and the underlying (physical) infrastructure. This mapping ensures that packets can be transmitted by the routing substrate between any pair of overlay nodes. However, in order to handle location resolution at network edge, a global location information base has to be maintained, which can be challenging in large networks.

There has been a large body of work falling in this category. SEATTLE [8] simplifies network management by flat addressing while providing hash-based resolution of host information (using a one-hop DHT) to ensure scalability. VL2 [9] implements a layer 2.5 stack on hosts and uses IP-in-IP encapsulation to deliver packets. PortLand [10] assigns Pseudo MAC (PMAC) addresses to all end hosts to enable efficient, provably loop-free forwarding with small switch state while leveraging a central fabric manager to address IP to PMAC translation in multi-rooted tree networks. NetLord [11] employs a light-weight agent in the end-host hypervisors to encapsulate and transmit packets over an underlying, multi-path L2 network, using an unusual combination of IP and Ethernet packet headers.

With the rapid evolvement of SDN, flow-based centralized control has been recently adopted as a mainstream control plane design for data center networks. As one of the first SDN solutions for enterprise networks, Ethane [12] enables the direct application of fine-grained flow-based policies to the network by coupling flow switches with a centralized controller. However, exposing all flows to the controller could bring too much workload to the controller, leading to poor scalability. Even after applying multi-threading optimizations that help achieve graceful linear core scaling factors [13], the gap between actual and desired performance of the centralized controller is still very significant. It was shown that the popular OpenFlow controller can only be able to handle approximately 30 thousand flow initiation requests per second on commodity x86 platforms [14]. Unfortunately, a small network consisting of only 100 switches could have a spike of more than 10 million flow arrivals per second [15]. Even after applying multi-threading optimizations that help achieve graceful linear core scaling factors [13], [16], the gap between actual and desired performance of the centralized controller is still very significant.

Recently, massive effort has been devoted to scaling centralized control to large networks. Existing solutions can be roughly classified into three categories:

1) Specific modifications: DIFANE [17] aims at handling all traffic in the data plane by selectively directing packets through intermediate (authority) switches that store the necessary rules pre-installed by the controller. DevoFlow [7] decouples control and global visibility and partly devolves control to switches by employing rule cloning and local actions at switches. The main disadvantage of this category of solutions is the requirement of modifying switches which largely limits its applicability in practice.
2) Distributed solutions: Onix [18], HyperFlow [19], ElastiCon [20], and Pratyaastha [21] are distributed platforms on top of which the network control plane can be implemented as a distributed system. Although they work well in moderate networks, this category of solutions does not solve the super-linear increase of control tasks when network scales to very large.
3) Centralized hierarchical solutions: Kandoo [23] is a two-layer control framework where network applications are classified into local and global control applications are handled by bottom- and top-layer controllers, respectively. These hierarchical solutions have the problem of path stretching, resulting in unnecessary delay in handling control tasks, although the scalability issue is highly mitigated [24].

There are also some distributed control plane proposals for specific networks such as [22], [25] for multi-domain networks, D-SDN for security issues [26]. Recently, Jain *et al.* [27] presented B4, a private WAN connecting Google's data centers worldwide based on a multi-layer software defined networking architecture.

LazyCtrl also targets the scalability issue of centralized control in large-scale data center networks. The most salient feature of LazyCtrl is that it carries out network control in the right granularity by exploring traffic locality in data center networks. We summarize the advantages of LazyCtrl as follows. Firstly, it solves the super-linear complexity problem by devolving control tasks to local control mechanisms. Secondly, it prevents path-stretching by taking advantage of direct distributed control for local traffic-intensive communication identities. Lastly, LazyCtrl requires no modification

on physical switches and it is very easy to implement.

Nevertheless, our solution is also orthogonal to distributed designs in the sense that it employs a hybrid control model, aiming at trying best to offload frequent coarse-grained control tasks from the central controller and handle them using distributed control mechanisms near datapaths. Therefore, the aforementioned research efforts for scaling flow-based fine-grained control is still applicable on top of LazyCtrl to further mitigate the performance bottleneck at the controller and consequently improve control plane scalability in data center networks.

2 Motivation

The following salient features of current cloud data centers largely motivate our design of LazyCtrl.

2.1 Traffic Locality in Data Centers

In cloud data centers, the traffic among the hosts is usually unevenly distributed and is strongly localized within some groups of hosts. To verify the correctness of this notion, we collected a day-long traffic trace from a production data center in Europe running multi-tenant applications and made the following quantitative findings:

- ▷ *The traffic distribution is uneven among hosts.* Among a total of 6509 hosts, only 11,602 of more than 20 million distinct $\langle src, dst \rangle$ host pairs exchanged traffic in the trace. And over 90% of the flows are contributed by about 10% of the host pairs that exchanged traffic.
- ▷ *The traffic appears to be concentrated within some groups of hosts.* For example, when partitioning the 6509 hosts evenly into 5 groups using k-way partitioning, we observe that only less than 9.8% of the traffic traversed different groups. We define the *centrality* of a group as the ratio (in $[0, 1]$) of the intra-group traffic and the total traffic related to the hosts in this group. For the collected trace, the average centrality of the 5 groups is 0.853, indicating a very high concentration of the data center traffic.

The above findings are not accidental and similar evidences can be found in [15], [28]. Actually, in a multi-tenant data center, network traffic tends to be localized within each tenant, as the applications from different tenants are isolated by virtualization techniques [29]. Therefore, we believe that by taking advantage of traffic locality, a global, fine-grained, and real-time network control may not be necessary for multi-tenant data centers.

2.2 Relatively Stable Tenant Size

For multi-tenant cloud data centers, we observe that the number of virtual machines for a single tenant is changing slightly, while the number of tenant users, as well as the total number of hosts in a multi-tenant data center, is experiencing a significant increase. For Amazon, a popular cloud service provider, the number of tenants, as well as total virtual machine instances of Amazon's EC2, grew about 2.5 times annually since 2006 [30]. The total number of objects held by Amazon S3 has grown 150 times from 2006 to 2011 [31]. In contrast, the size of a specific tenant in terms of number of rented virtual machines is constantly around 20–100 [1]. These facts consequently lead to the property that traffic is aggregated within some size-limited groups of hosts in multi-tenant data centers as the traffic exchanged among different tenant slices is very limited. By taking full advantage of this property, we show that the explosive increase in the number of tenants does not necessarily result in scalability issues for centralized control in data center networks.

3 Design

LazyCtrl realizes a hybrid control plane for data center networks. In this section, we discuss four aspects of its design: the architecture, the switch grouping scheme, the packet forwarding routine, and the failover mechanisms. We first provide a high-level overview to state the intuition of our design.

3.1 High-level Overview

In conventional flow-based centralized control environments such as those based on OpenFlow [32], the controller maintains the network-wide state (the host-to-switch mapping here) and handles all the flows between switch pairs that exchange data, bringing extremely high burden to the controller. LazyCtrl mitigates this problem by clustering the switches into multiple switch groups according to their communication affinity and devolving intra-group control to these switch groups (termed Local Control Group, LCG).[1] To support its laziness, the controller prefers clustering the switches into a few big groups in order to reduce inter-group communication. However, larger group size would result in larger distributed forwarding tables and more control tasks inside each local control group. Due to the limited size of high-speed memory in switches, the largest size of a group will be constrained by some constant. *The controller clusters the switches in such a way that the size of each group is maximized under a given limit while the inter-group traffic volume is minimized.*

Example: Consider a multi-tenant cloud data center containing a central controller and five edge switches (namely SA, SB, SC, SD, and SE) with hosts[2] directly attached. We focus on the scenario shown in Fig. 1. There are three tenants, A, B, and C, each of which has some virtual machines. The left figure illustrates the case when centralized controlling is applied directly and thus the central controller has to handle all the flows among all edge switches. LazyCtrl changes this situation by clustering edge switches into independently groups. As can be seen in the right figure, the controller clusters SA, SC, and SE into the first group while SB and SD together form the second group. (We assume that the group size limit is three in our example.) This way, the traffic within the first group (*e.g.*, SA↔SC), as well as the traffic within the second group (*e.g.*, SB↔SD), can be handled by carrying out local control mechanism that is dedicated for each group. The controller then is only

1. We will use group and local control group (LCG) interchangeably in the rest of this paper.

2. With a bit abuse of notation, we will use host to refer to virtual machine that is running in a physical server.

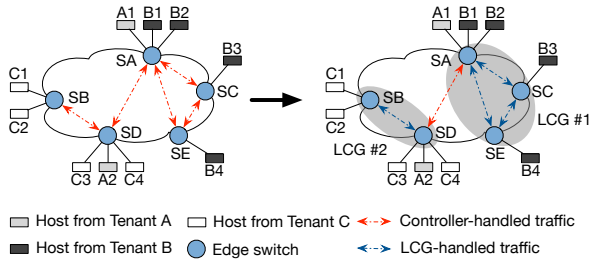

Fig. 1. Example to demonstrate the idea of LazyCtrl. Edge switches are clustered into multiple local control groups according to their communication affinity.

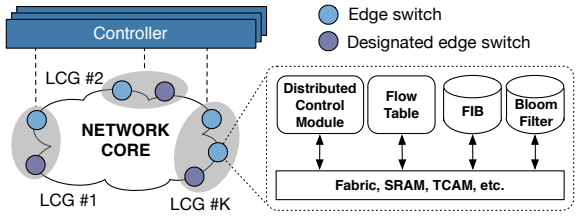

Fig. 2. Architecture design of LazyCtrl, where the network control plane consists of a logically centralized controller and distributed control modules in multiple local control groups.

needed to take charge of the inter-group traffic, *i.e.*, SA↔SD. The switches will be dynamically regrouped in response to traffic variation.

3.2 LazyCtrl Architecture

The architecture design of LazyCtrl is depicted in Fig. 2. In our design, the network is separated into two parts: the core and the edge. We employ a hybrid control model where control tasks are handled by the distributed control mechanisms in LCGs at the network edge, complemented by a central controller.

3.2.1 Core–Edge Separation

Our design splits the core from the edge. The network core can be any simple and scalable network (*e.g.*, an IP unicast network), which serves as the underlay providing connectivity for edge switches. The core–edge separation releases the network core from handling complicated and dynamic network control tasks (*e.g.*, network virtualization, virtual machine migration) and thus allows the network core to be constrained only by performance and reliability. Since our focus is the control plane, we omit the detailed design of the network core.

In contrast, the network edge is in charge of network intelligence, *i.e.*, host-to-switch mapping. The layer two virtual networks (overlays) for providing connectivity for the edge switches are conducted by the network edge via encapsulation or tunneling on top of the underlying physical network core. As a result, one-hop distance can be assumed for each pair of edge switches. We introduce a hybrid control model for the control plane to handle network control tasks.

3.2.2 Hybrid Control Model

To extend the scalability of the control plane, we introduce a hybrid control model in the LazyCtrl design. This hybrid control model involves a central controller and a set of local control groups.

The central controller has holistic visibility over the entire data center network and is responsible for i) maintaining a Central Location Information Base (C-LIB) which preserves host location information, ii) adapting the grouping of the edge switches, and iii) managing the flow tables on the edge switches to handle inter-group traffic and any specific traffic that needs flexible centralized control. The goal of the central controller is to stay lazy by devolving as many control tasks as possible to the local control groups. The central controller can be a stand-alone physical server or a logical controller comprised of a cluster of servers with strong reliability and coherency of network state.

A local control group is a group of edges switches whose clients are observed to have frequent mutual communication. These switches are grouped together by the controller and share the network state with each other consistently. Each local control group employs a distributed control mechanism to take over the control workload of intra-group traffic from the controller. The distributed control mechanism inside each group is carried out by equipping each edge switch with some local forwarding tables that are maintained by the switches themselves. These local forwarding tables keep track of network states such as host-to-switch mapping inside the corresponding group. For each local control group, a designated switch (with some backups) is selected randomly by the controller, which is responsible for aggregating group-wide network states from the edge switches in this group and reporting them to the controller in an asynchronous manner.

3.2.3 Control Message Channels

There are three types of control message channels, *i.e.*, logical links, in the hybrid control model for LazyCtrl.

- ▷ *Control link.* A control link refers to a logical control channel (an IP tunnel or a TCP/SSH connection on top of the underlay network) via which the controller receives forwarding requests, and/or sends commands or rules to individual edge switches. The control link is extended from the secure channel between an OpenFlow controller and an OpenFlow switch by allowing the exchange of switch grouping and other related messages. When a control task cannot be handled by local control groups, packets will be forwarded to the controller and the controller will react to the edge switches by sending them flow rules or other commands, all through the control link.
- ▷ *State link.* A state link is a logical communication channel between the controller and a designated switch. The designated switch in each group aggregates the network states it collects from other edge switches in the group and reports them to the controller periodically via the state link. Thus, global and coherent visibility can be achieved at the controller.
- ▷ *Peer link.* A peer link refers to a logical control channels used for disseminating network states for

address learning and updating among the switches in the same local control group. In principle, peer links would rely on multicasting. However, assuming native multicast support for the underlay may not be practical. Therefore, our design adopts an alternative approach: the designated switch (or its backup, if any) gathers network states from every peer edge switch and then disseminates them to all other switches in the same group with multiple unicast messages.

3.3 Switch Grouping

The design of LazyCtrl is based on the concept of grouping switches to form multiple local control groups. Thus the quality of efficiency of the grouping is essential to the whole design. Given a limit for the group size (determined according to empirical or historical data), a good grouping scheme is defined as one in which the inter-group traffic is small (in order to facilitate the laziness of the controller) and the computational complexity of the grouping algorithm is sufficiently low such that it can fast adapt to traffic dynamics. Our grouping algorithm aims at satisfying the above principles and we base our design on solving the classical graph partition problem, with improvements on time complexity and support for incremental updates.

3.3.1 Problem Modeling

Denote by $\mathbf{S} = \{S_1, S_2, ..., S_N\}$ the set of edge switches in the multi-tenant data center network. Let $\mathbf{W} = \{w_{i,j} \mid S_i, S_j \in \mathbf{S}\}$ be an intensity matrix where each element $w_{i,j}$ represents the normalized traffic intensity (*i.e.*, number of new flows per second) between two edge switches S_i and S_j. A grouping scheme $\mathbf{G}$ is a series of disjoint subsets of edges switches, which can be defined by $\mathbf{G} = \{G_1, G_2, ..., G_K \mid (G_i \subseteq \mathbf{S}) \wedge (G_i \cap G_j = \emptyset)\}$. Then, the normalized inter-group traffic intensity (denoted by W_{inter}) can be represented by

$$W_{\text{inter}} = \sum_{\{x,y \in [1,...,K] \wedge x \neq y\}} \sum_{\{S_m \in G_x, S_n \in G_y\}} w_{mn}.$$

Given an intensity matrix $\mathbf{W}$, the goal of the switch grouping problem is to find out a grouping scheme $\mathbf{G}$ such that the inter-group traffic intensity W_{inter} is minimized. This problem is similar to the graph partition problem where the goal is to partition a given graph into k roughly equal components such that the total weigh of the edges connecting the vertices in different components is minimized (called k-way partitioning). The graph partition problem has been shown to be NP-hard [33]. The switch grouping problem differs slightly from the graph partition problem in terms of that the largest size of a group is strictly contained by a constant while the number of groups is variable.

3.3.2 Solving the Switch Grouping Problem

Our design for the switch grouping algorithm is based on the Multi-Level k-way Partition (MLkP) algorithm proposed by Karypis and Kumar for fast k-way partitioning for a given graph [33]. MLkP first reduces the size of the graph by collapsing vertices and edges. When a k-way partitioning of the smaller collapsed graph is found, the algorithm uncoarsens and refines this partitioning to construct a k-way partitioning for the original graph. The running time of MLkP is linear in the number of edges in the graph. However, direct application of MLkP to the switch grouping problem may lead to infeasible solutions, *i.e.*, the sizes of the resulted partitions may exceed the given group size limit.

```
IniGroup:
 1: // construct the intensity graph
 2: ConstructGraph(history intensity matrix)
 3: // obtain the initial grouping
 4: MLkP(intensity graph, #partition k)

IncUpdate:
 5: // running in background
 6: while(true):
 7:   // the controller is overloaded
 8:   while (controller.load > threshold.high):
 9:     // find two candidate groups with
10:     // the most significant traffic change
11:     cgroups = FindGroups(all groups)
12:     sgroup = MergeGroups(cgroups)
13:     ngroups = SplitGroup(sgroup)
14:     // the controller is underloaded
15:     if (controller.load < threshold.low):
16:       break
```

Fig. 3. Pseudocode for the SGI algorithm.

We propose SGI, a Size-constrained Grouping algorithm with Incremental update support. In the initial stage (function `IniGroup`), SGI first determines the right number k of groups to be generated. This value can be estimated by the number of switches divided by the group size limit. Next, SGI constructs an intensity graph where the vertices in the graph represent all the switches while each edge represents the communication between the two end switches of this edge. The weight on each edge indicates the normalized traffic intensity between any pair of switches, which is estimated based on history traffic statistics. Then, an initial feasible grouping of the switches is produced by using the MLkP algorithm with the constructed graph as input. Hereafter, SGI keeps running by monitoring the traffic condition on the network. Upon a significant change[3] on the traffic distribution, SGI carries out a greedy refinement function called `IncUpdate` to incrementally update the grouping in order to reduce the inter-group traffic. The refinement process runs iteratively and in each iteration, two groups (`cgroups`) between which traffic volume increases the most are merged (as `sgroup`) and split again to ensure minimized communication between the two new groups (`ngroups`). This is identical to finding a minimum bisection cut of a given graph, which can be accomplished efficiently in polynomial time [34]. The refinement process will terminate when the workload of the controller meets some threshold. The pseudocode of the SGI algorithm is given in Fig. 3.

3.4 Packet Forwarding

3.4.1 Setup Phase

Similar to that of typical OpenFlow control, in LazyCtrl, the edge switches are configured to point to the central

3. The controller evaluates the significance of traffic change by measuring the difference in its workloads.

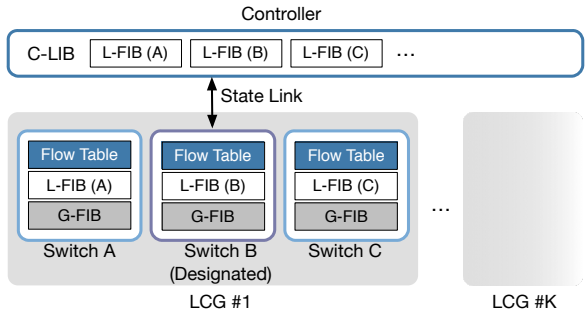

Fig. 4. Table organization in the LazyCtrl design.

controller at the setup phase. Besides generating the local control groups by invoking the SGI algorithm, the controller is also in charge of the following configurations for every group before the whole LazyCtrl system comes into function.

Selecting designated switches. For each local control group, the controller selects a designated switch among all edge switches in this group by applying some given principle such as shortest physical distance, shortest response time to the controller. If necessary, the selection process also includes choosing some backups for the designated switches.

Ordering and informing edge switches. The controller orders all switches in a group according to the physical (MAC) address of switch's management interface. This is for building a logical ring for failure auto-detection (detailed in Section 3.5). The controller then delivers to each switch its neighbors on the logical ring. Besides that, the controller will also inform the switches in a group with the designated switch ID and some global timing and performance parameters such as the group size, the frequency to apply group synchronization or keep-alive heartbeats.

3.4.2 Table Organization

The core–edge separation enables one-hop "logical" distance between any pair of edge switches, leaving basic packet routing to the IP underlay. What remains unsolved is the host-to-switch mapping.

In the LazyCtrl design, each edge switch is associated with a Local Forwarding Information Base (L-FIB), which tracks the hosts or virtual machines that are attached to this switch. To handle intra-group traffic, each edge switch also maintains a replica of the L-FIBs of all other switches in the same group, which we call Group Forwarding Information Base (G-FIB). The central controller retains global visibility of the network by maintaining a Central Location Information Base (C-LIB), which contains the L-FIBs of all edge switches in the network. Using this C-LIB, the controller can handle inter-group traffic and any other specific flows whose control requires global visibility. A general overview of the table organization in the LazyCtrl design is depicted in Fig. 4.

L-FIB: The L-FIB of each edge switch is implemented with a conventional lookup mechanism similar to the MAC/ARP table in ordinary layer two switches.

G-FIB: The G-FIB of each edge switch is a replica of the L-FIBs of all switches in the same group. To save storage space, we implement G-FIB using Bloom Filter (BF), as the storage space required by a BF is independent from the number of elements it contains. The G-FIB of each edge switch is comprised of multiple BFs generated from the L-FIBs of all switches in this group. Given an address of a virtual machine, each BF decides whether this address is under the corresponding edge switch. All the BFs together will return a vector of Boolean values indicating the possible location of this address. Note that it might happen that there are multiple possible locations for one address, which is resulted from the *false positive* of BFs. However, the false positive rate is predictable and controllable by space-time trade-offs [35].

3.4.3 State Dissemination

State dissemination consists of the mechanisms to spread and synchronize network states (*e.g.*, the host-to-switch mapping) and updates in the control plane. In general, there are two types of state dissemination in LazyCtrl:

Live/Synchronized state dissemination. Live state dissemination refers to the host discovery process driven by the end hosts via ARP broadcasting in the bootstrapping stage and at virtual machine migration or removal. In the LazyCtrl design, live state dissemination can be cascaded in three different levels: i) Upon receiving an ARP request, the edge switch learns the source address by inserting or updating an item in its L-FIB and then floods the request to all relevant local ports. ii) If no local hosts (that are attached to this edge switch) answer this request and the requested destination cannot be recognized by the G-FIB of the switch either, this request will be forwarded to the designated switch in this group for an intra-group "broadcasting". iii) Further, if there is still no response from the hosts in this group, the request will be forwarded to the central controller, which relays the request to the designated switches in all other groups that contain hosts belonging to the relevant tenant (*e.g.*, according to tenants' VLAN settings).

Asynchronous state dissemination. When the traffic pattern changes, the grouping of the switches may not be effective for shielding the central controller thus needs to be adjusted. The condition of inter- and intra-group traffic is also changed. Therefore, the host-to-switch mapping must be re-disseminated across the control plane in order for local control groups to handle all intra-group traffic. This is different from the case of virtual machine migration in the sense that there is no change to the host-to-switch mapping. As a result, the end hosts cannot sense this change and thus cannot accordingly drive any updates. Moreover, in an extreme case where all hosts from a certain tenant appear in the same local control group, the controller may want to block all ARP request from that tenant to avoid unnecessary workload of itself. However, this could lead to incomplete visibility for the controller as the traffic from that tenant will be transparent to the controller.

In order to handle the above circumstances, an asynchronized switch-driven state dissemination mechanism must be introduced. In LazyCtrl this mechanism contains two aspects: i) When an update event occurs at an edge switch, this switch sends its updated L-FIB to the designated switch in the group via the peer link; the designated switch then relays this update to all the other switches in the same

```
Upon arrival of a packet P at an edge switch S:
 1: // P originates from a local host
 2: if (P is a local plain packet):
 3:   // handled by flow table
 4:   if (P matches S.flow_table.rule):
 5:     apply S.flow_table.action to P
 6:   // handled by local control group
 7:   else:
 8:     // lookup in the L-FIB
 9:     host = LookUp(L-FIB, P.dest_addr)
10:     // no match found
11:     if (host == none):
12:       // query in the G-FIB
13:       dst_vec = Query(G-FIB, P.dst_addr)
14:       // no match found, handled by controller
15:       if (dst_vec == empty):
16:         send P to the controller
17:       // send P to all possible targets
18:       else: for each (dst in dst_vec):
19:         encap. and send a copy of P to dst
20:     // match a local host
21:     else: forward P to host
22: // P is an encapsulated packet
23: else:
24:   Decapsulate(P)
25:   // lookup in the L-FIB
26:   host = LookUp(L-FIB, P.dst_addr)
27:   // no match found (due to false positive)
28:   if (host == none): Drop(P)
29:   else: forward P to host
```

Fig. 5. Packet forwarding routine in LazyCtrl.

group to synchronize the group-wide network state. The designated switch then sends the update to the controller via the state link to synchronize the network state between the controller and the local control group. ii) When the grouping of the switches has been changed, the controller sends the L-FIBs of the switches in a new group to the designated switches in this group via the state links. The designated switch then "broadcast" the L-FIBs to all the edge switches in this group for updating their G-FIBs.

3.4.4 Packet Forwarding Routine

We describe now how traffic control is carried out in LazyCtrl. The detailed forwarding routine of a packet is shown in Fig. 5. When a packet arrives at an edge switch, depending on packet type, the following two actions will be applied: i) If the packet is plain (which originates from a local host), the switch first carries out a lookup in its flow table to check whether there are matched rules for this packet. If so, the action corresponding to the rule is then applied to the packet; otherwise, the switch continues looking up in its L-FIB to check whether the destination of this packet is a local host. A packet with an address of a local host will be forwarded directly to that host. If no entry matched the L-FIB, the switch carries out a query in its G-FIB. Note that there might be multiple targets for this packet returned from this query due to the false positive of BFs. The switches then send to all the targets a copy of the packet. If all the elements in the Boolean vector are false, it means that the target of this packet is not in the current group and thus the packet will be forwarded to the controller to request inter-group control rules. ii) If the packet is encapsulated, the switch first decapsulates it and then carries out a lookup in its L-FIB to determine its destination host. If no matched entries are found, the switch simply drops the packet as it knows that this packet is mis-forwarded to the switch due to BF's false positive. Optionally, this mis-forwarded packet could also be directed to the controller for installing flow entries on related switches to avoid further false positive for the same destination.

TABLE 1
Inferring failures in the control plane according to the place of packet loss.

Failure	Packet loss		
	$S_n \rightarrow S_{n-1}$	$S_n \rightarrow S_{n+1}$	Controller$\rightarrow S_n$
Control link			✓
Peer link (Up)	✓		
Peer link (Down)		✓	
Switch (S_n)	✓	✓	✓

3.5 Failover

3.5.1 Failure Detection

The switch grouping scheme ensures that switches in the same group are "strongly connected" due to their frequent traffic exchange. As a result, failures in the data plane can be passively detected quickly. In contrast, handling failures in the control plane is more laborious.

In the LazyCtrl design, we propose a self-detection mechanism to handle failures in the control plane based on a group-wide failure-detection wheel with the controller at the center and the switches at the edge. As we have mentioned previously, at the setup phase the controller orders the switches to form a wheel and informing the switches in the same group their neighbors on the wheel. To detect failures, keep-alive messages will be initiated from upstream switches to downstream switches and from the controller to each switch. All possible cases of failures depending on the place of packet loss are listed in Table 1.

3.5.2 Failover of Links

Link failures indicate routing-related issues, *e.g.*, packet loss due to link congestion or temporary routing loops on the underlay. We adopt detour routing based approaches to handle link failures in LazyCtrl. When a data path failure occurs, for instance, between S_n and S_{n-1}, the controller will be notified and an alternative path will be chosen for delivering packets following $S_n \rightarrow S_{n-1}$. For a failure on the control link between the controller and a switch such as S_n, the controller will send a request to the upstream switch of S_n on the failure-detection wheel, *i.e.*, S_{n-1}, to pass on the control message from S_n to the controller. When a peer link failure occurs (between S_n and S_{n-1}), the control functionality is affected only when one of the two end switches is the designated switch. In this case, the controller will ask S_n or S_{n-1} to quit as the designated switch and reselects one from the backups for the designated switch to fulfill the role of designated switch.

3.5.3 Failover of Switches

A switch failure usually turns to be a reboot or a reset of the switch, especially in the case where edge switches are implemented with virtual switches in hypervisors. The controller

is responsible for detecting the malfunction of the switch and then carries out the following actions: i) informing the designated switch in the same group this switch failure and asking the designated switch to spread the temporary outage of the failed switch in the group in order to avoid unexpected detour routing requests; ii) rebooting the failed switch remotely and checking its comeback periodically; iii) removing the outage signal and proactively triggering a state synchronization in the group when the switch is back to function.

If the failed switch is the designated switch in the group, in addition to the above actions, the controller will select a new designated switch for the group. If backups are set for the designated switch, no single point of failure exists since those backups work simultaneously and will be fixed upon a failure independently.

4 Implementation

We implement LazyCtrl by extending the OpenFlow protocol and developing edge switches and the controller based on Open vSwitch [36] and Floodlight [37]. The source code of our implementation can be found on [38].

4.1 Open vSwitch-based Edge Switch

The main forwarding component of Open vSwitch consists of the `ovs-vswitchd` and datapath modules. The `ovs-vswitchd` module works in the user space, handling slow path processing such as learning, remote configuration, full flow-table lookup; the datapath module in the kernel space handles fast path processing including packet forwarding, quick-table lookup, modification, and tunneling. The implementation of the LazyCtrl edge switch follows a similar design principle. Fast path processing, such as L-FIB lookup (including BF matching), packet encapsulation, and forwarding, are integrated into the kernel space (datapath) module while a few slow path modules are integrated into `ovs-vswitchd` which are listed as follows.

- ▷ *Ctrl-IF module* is an interface for the switch to interact with the controller, which also implements the control link. Unknown packets (from inter-group traffic) will be forwarded to the controller using OpenFlow `Packet_In` messages.
- ▷ *State advertisement module* is introduced for collecting and disseminating local host information and traffic statistics among the switches in the same group.
- ▷ *FIB maintenance module* maintains the L-FIB and the Bloom filter based G-FIB structures according to the network states collected by the state advertisement module and then updates the kernel space module for fast path processing.
- ▷ *State reporting module* will only be activated when the switch is selected as the designated switch for the group. This module implements all functions associated with the state link.

4.2 Floodlight-based Controller

The Floodlight OpenFlow controller provides a rich set of components. The central controller in LazyCtrl is implemented based on the existing Floodlight controller by introducing the following extensions.

- ▷ *Encap action* realizes packet encapsulation in edge switches by extending the existing OpenFlow v1.0 protocol. In the LazyCtrl architecture, packet forwarding in the data plane overlay replies on a GRE-like encapsulation. When a rule with this action is applied to a flow, the switch will encapsulate the packets with a new header targeting a given remote IP address.
- ▷ *C-LIB maintenance module* implements the functions of acquiring L-FIBs from the designated switch in every group and building the C-LIB at the controller.
- ▷ *Switch grouping management module* handles the management of the local control groups. We base our implementation of switch grouping on the proposed SGI algorithm. A daemon module is introduced to handle the state reports from the designated switches in all groups and keep analyzing the changes in traffic pattern. Re-grouping will be triggered when i) the workload of the controller suffers from an accumulated growth of up to 30% from last update or ii) it has been two minutes since last update. Setting up a minimum update interval (2 minutes here) is to prevent the oscillation caused by short-term traffic fluctuation.
- ▷ *Tenant information management module* is used to manage tenant information such as VLAN IDs in switches. Being aware of this information, the controller can determine where to spread the ARP messages and when inter-group traffic control is necessary.
- ▷ *Failover module* is in charge of failure detection and recovery as we have discussed in Section 3.5.

5 Evaluation

5.1 Prototype Setup

We conducted experiments based on a real traffic trace collected from an enterprise production data center in Europe, which consists of 272 GigE edge switches and 6509 hosts. Accordingly, we built a prototype system using 6 Pronto 3290 switches and 24 IBM x3550 8-core (two quad cores) servers. The switches were interconnected with a full mesh via 10 GigE links, severing as the network core (IP-based underlay). Each switch was connected with 4 servers via GigE links. To emulate the 272 edge switches in the real data center, we deployed 272 Linux virtual hosts running our modified Open vSwitch implementation on the 24 servers. A custom-made trace re-player was developed and deployed on each of the 272 Linux virtual host to replay the inter-switch traffic generated by the 6509 hosts in the trace. The Floodlight-based central controller was hosted on a standalone Linux PC (with Intel Core 2 Duo CPU 2.2 GHz) and could be configured to run in either lazy or normal mode.

5.2 Datasets

The real traffic trace we collected consists of the traffic among 272 GigE edge switches and 6509 hosts over a whole day. To check the consistency of the performance results under different traffic scenarios, we generated three

TABLE 2
Characteristics of the traffic traces.

Trace	# of flows	Avg. centrality	p (%)	q (%)
Real	271M	0.85	N/A	N/A
Syn-A	2720M	0.85	90	10
Syn-B	3806M	0.72	70	20
Syn-C	5071M	0.61	70	30

synthetic traffic traces based on the real trace. The main characteristics of all traces are summarized in Table 2. In the synthetic traces, traffic is assumed to be exchanged through 2713 edge switches among 65090 hosts, with a scaling-up factor of 10 compared with the real trace. The traffic flows in the synthetic traces were generated in the following manner so that the key characteristics such as the temporal patterns could be retained: $p\%$ of the flows are generated by selecting from a given set of host pairs ($q\%$ of all host pairs) uniformly at random in the synthetic topology, and assigning each selected host pair a payload randomly chosen from the real trace. We vary the values for p and q and three synthetic traces are generated with significant differences in traffic locality represented by average centrality. The rest flows are generated by selecting host pairs uniformly at random from all host paris in the synthetic topology. Each selected host pair is assigned with a payload randomly chosen from the real trace.

5.3 Performance of Switch Grouping

We evaluate the quality of the proposed switch grouping scheme by calculating the normalized inter-group traffic intensity (W_{inter} as defined in Section 3.3). Fig. 6(a) depicts the results of applying the size-constrained MLkP algorithm (the `IniGroup` function in SGI) to the traffic derived from each of the three synthetic traces with various numbers of groups. We observe that the grouping quality varies across different traces. In general for traces with higher average centralities, it tends to have smaller values for W_{inter}, indicating better performance in reducing the workload of the controller. We also observe that W_{inter} increases almost linearly with the increase of the number of groups, confirming that maximizing the sizes of single groups (thus consequently reducing the number of groups) will best facilitate the laziness of the controller.

We also carry out measurements to examine the computation time in switch grouping generation on the three synthetic traffic traces. The results of applying the `IniGroup` function in SGI with various group size limits are shown in Fig. 6(b). It can be seen that switch grouping can be accomplished as fast as less than 5 seconds and the grouping time is inversely proportional to group size limit. Note that switch grouping is only carried out when there is a very significant change in traffic pattern and incremental updates are not possible to retain good grouping quality in reasonable time. During most of the time, applying the `IncUpdate` function is sufficient, which is more than one order of magnitude faster than the `IniGroup` function.

5.4 Effectiveness of LazyCtrl

Controller workload. We validate the effectiveness of LazyCtrl by measuring the controller workload under traffic dynamics. We first conduct a comparison to standard OpenFlow control (with the original Floodlight implementation) using the real traffic trace. For LazyCtrl, the initial grouping is done based on the first-hour traffic pattern and we test in both *static* and *dynamic* cases with and without incremental updates for the grouping, respectively. To further verify the consistency of the results, we expand the real trace by introducing 30% extra flows among the hosts that did not communicate with each other in the real trace during the time interval from 8 to 24. Using the *expanded* trace, we test again in both static and dynamic cases. We compare the workload of the controller in the above cases and the experimental results are illustrated in Fig. 7. It can be observed that i) LazyCtrl can help achieve a significant level of workload reduction (about 61%–82%) for the controller; ii) The controller workload in LazyCtrl is relatively stable during the day on the real trace, which is due to the fact that majority of the traffic growth happens among those "strongly connected" hosts inside local control groups, being transparent to the controller. iii) The controller workload can be significantly reduced when the `IncUpdate` function is applied due to the fact that the additionally introduced flows keep breaking the skewness of the traffic over time and thus grouping updates have to be applied

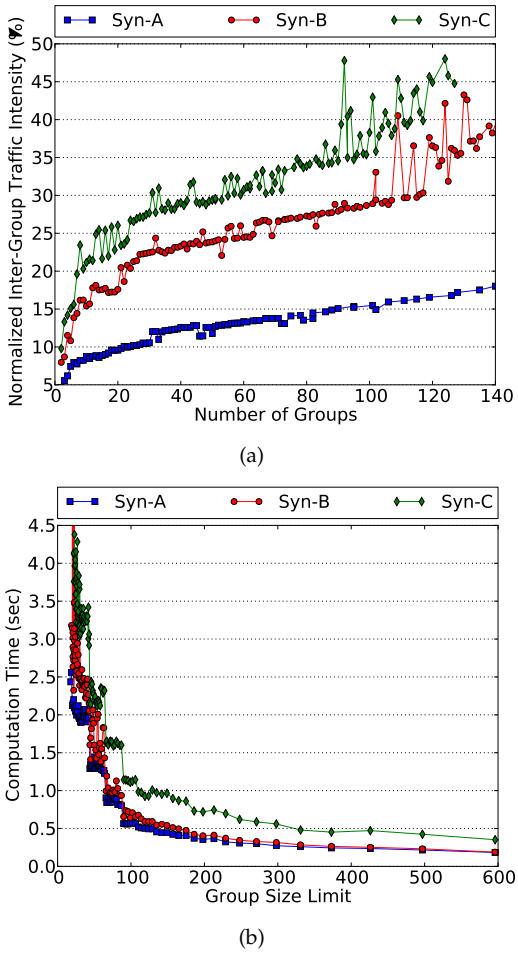

Fig. 6. a) Normalized inter-group traffic intensity when grouping with different numbers of groups on each of the three synthetic traffic traces. b) Computation time of switch grouping under different group size limits on each of the three synthetic traffic traces.

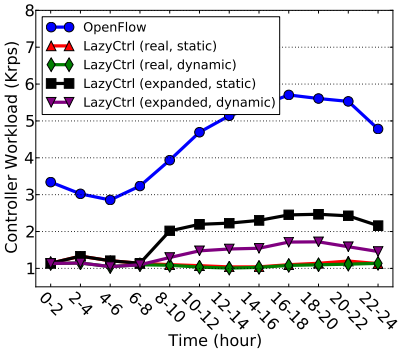

Fig. 7. Controller workload.

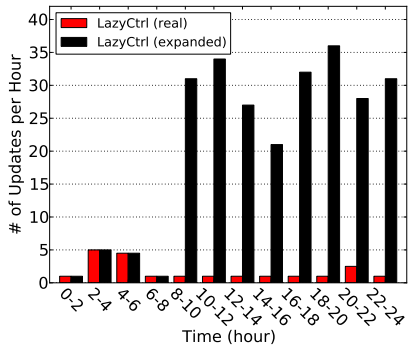

Fig. 8. Switch grouping frequency.

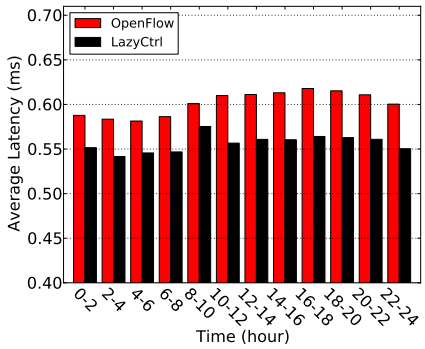

Fig. 9. Steady state latency.

continuously to adapt to the changes in order to prevent the controller from being overloaded.

Grouping update. In addition, we examine the update frequency of switch grouping on both the real and expanded traces. The update frequency results are shown in Fig. 8. It can be noticed that the incremental update function has very limited influence on the controller workload on the real trace. At the same time, the update frequency keeps at a very low level (10 updates per hour), indicating that maintaining a relatively effective grouping is feasible in practice. On the expanded trace, the cost for keeping the controller lazy is a reasonable increase in update frequency (with a maximum of 34 updates per hour).

Storage overhead. The storage cost of the BF-based G-FIB on each switch is linear with the group size. For example, when a group consists of 46 switches, for each switch the BF-based G-FIB contains 45 bloom filters. Assuming that each bloom filter has 16 128-byte entries, the memory required for the BF-based L-FIB on each switch is $45 \times 16 \times 128 = 92,160$ bytes, resulting in a false positive rate of less than 0.1%.

5.5 Latency Overhead

Cold-cache forwarding latency. We evaluate the forwarding latency under "cold-cache" scenarios upon the first packet of a fresh flow is injected into the network. We emulate cold-cache scenarios by launching 45 new flows among 5 newly deployed hosts and compare the average forwarding latency of the first packets of these flows in LazyCtrl to that in the standard OpenFlow control. For intra-group traffic, the cold-cache forwarding latency in LazyCtrl (0.83 ms) is more than an order of magnitude smaller than that in OpenFlow (15.06 ms). This is due to the fact that packets from intra-group traffic will be forwarded locally without involving the controller. The data plane operations such as L-FIB lookup and packet encapsulation are very fast and thus packets can be processed at line speed. For inter-group traffic, LazyCtrl also outperforms standard OpenFlow by achieving a cold-cache latency of 5.38 ms. This is because LazyCtrl requires no passive learning of the network topology through all ARP flooding as is the case of standard OpenFlow (the `learning-switch` module in Floodlight), which is another benefit brought by the lazy principle in LazyCtrl.

Steady-state latency. Steady-state latency refers to the average forwarding latency of all processed packets over a relatively long period of time (2 hours here). The experimental results on the real trace with a 24-hour span are illustrated in Fig. 9. It can be observed that on average a 10% reduction on latency can be achieved by LazyCtrl compared with standard OpenFlow. Moreover, this improvement is a byproduct of reducing the workload of the controller as less load on the controller leads to higher processing speed. Moreover, the synchronized state dissemination speeds up topology learning, which implicitly help reduce the response time of the controller.

6 Discussion

6.1 Scalability

Both distributed and centralized control approaches have some instinct limitations in scaling to large-scale data center networks. In general, distributed control such as link-state protocols depends on broadcasting to synchronize network states, which will be a disaster when the network becomes inconceivably large. Moreover, the edge switches need to learn the locations of all hosts, leading to explosive forwarding table sizes. While eliminating the need for state synchronization in principle, centralized control suffers from the stress of handling frequency and resource-exhaustive events such as flow arrivals and network-wide statistics collection events at the controller, which consequently limits the scalability of network control in data centers.

LazyCtrl aims at finding out the right balance between distributed and centralized control and integrates the advantages from both sides to fundamentally solve the scalability issue of network control in data centers. The scalability of LazyCtrl is interpreted in three aspects:

- *Table organization and state dissemination.* Due to the group size limit, the L-FIB and G-FIB on each switch will be constrained to limited sizes and thus do not have any scalability issue. Switch-wide state dissemination is also constrained in specific group and is decoupled from the size growth of the data center. The controller is designed to passively receive network states from local control groups for state dissemination and address leaching, which scales as well.
- *Location resolution.* Location resolution responsibility in LazyCtrl is shared among switches and the controller. Switches handle the majority of tasks locally

in local control groups while the controller is only involved when intra-group control is not sufficient. As a result, the workload of the controller can be kept at a very low level, mitigating the scalability issue.

- ▷ *Failure detection and failover.* Clustering switches into "strongly connected" groups with limited sizes simplifies the process of failure detection and recovery of a large network system as failover tasks can be carried out independently in each of the groups. In addition, control authority in LazyCtrl is shared by local control groups and the controller, avoiding a single performance bottleneck.

6.2 Optimizations on Switch Grouping

For simplicity of exposition, we omitted some optimization efforts we carried out for switch grouping in Section 3.3. Now we highlight some of them.

- ▷ *Host exclusion in switch grouping.* When a edge switch is connected to hosts belonging to many tenants, it may be difficult for a greedy method to generate a grouping with superb quality. In this case, the controller can choose some hosts and exclude them from the grouping process. The control tasks for these hosts will be accordingly handled by the controller.
- ▷ *Preload for seamless grouping update.* During grouping updates, the L-FIBs on the related switches will be modified, leading to forwarding interruptions. To relieve this, the controller can preload some rules to the related switches to temporarily handle the control tasks for them. These rules will be removed when the grouping becomes stable.
- ▷ *Acceleration by parallelism.* The `IncUpdate` function in the SGI algorithm can be easily parallelized by carrying out merge and split operations simultaneously for multiple group pairs. Consequently, the computation overhead brought by the regrouping process can be further reduced.

6.3 Determining the Right Group Size

Determining the right sizes for groups plays an important role in keeping LazyCtrl effective. Intuitively, the larger the group size, the lower the expected workload for the controller due to less inter-group traffic. On the other hand, the larger the group size, the higher the control overhead on the switch side, as a larger group means more network states to spread among the switches in the group and more L-FIBs and G-FIBs to maintain.

Compared with empirically driven or static group sizes, we believe that a dynamic group size negotiation between the controller and the switches can be helpful, as networks can be heterogeneous and the switches might differ significantly in terms of performance and capacity. Furthermore, the flexibility of on-demand group size makes it possible for the controller to customize its workload (*e.g.*, during peak hours). As an alternative, we also implement a game-based (modified Rubinstein Bargain Model) dynamic group size limit negotiation approach in LazyCtrl. Before the controller calculates the grouping, the switches are allowed to dynamically bargain the group size limit with the controller according to their real-time monitored and self-evaluated data.

7 Conclusions

In this paper we present LazyCtrl, a novel hybrid control plane design for data center networks. LazyCtrl is based on a core-edge separated architecture and the control functionality is implemented in a hybrid fashion: frequent coarse-grained control tasks are largely devolved to network edge by clustering edge switches into local control groups according to traffic locality and carrying out distributed control independently inside each group; the central controller is only in charge of very limited number of inter-group or other fine-grained control events. The central controller keeps adapting the grouping of edge switches to maintain its laziness. Our evaluation on the LazyCtrl prototype with both real and synthetic traffic traces show that LazyCtrl can help reduce the workload of the central controller by up to 82%, improving the scalability of standard OpenFlow to a large extend. Moreover, LazyCtrl is fully compatible with existing solutions for scaling flow-based centralized control to large networks.

Acknowledgments

The authors would like to thank...

References

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.
[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008, pp. 63–74.
[3] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *SIGCOMM*, 2009, pp. 63–74.
[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI*, 2010, pp. 281–296.
[5] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks," in *NSDI*, 2010, pp. 249–264.
[6] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Co-NEXT*, 2011, pp. 8–20.
[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *SIGCOMM*, 2011, pp. 254–265.
[8] C. Kim, M. Caesar, and J. Rexford, "Floodless in seattle: a scalable ethernet architecture for large enterprises," in *SIGCOMM*, 2008, pp. 3–14.
[9] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *SIGCOMM*, 2009, pp. 51–62.
[10] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *SIGCOMM*, 2009, pp. 39–50.
[11] J. Mudigonda, P. Yalagandula, J. C. Mogul, B. Stiekes, and Y. Pouffary, "Netlord: a scalable multi-tenant network architecture for virtualized datacenters." in *SIGCOMM*, 2011, pp. 62–73.
[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *SIGCOMM*, 2007, pp. 1–12.
[13] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Hot-ICE*, 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/tootoonchian

[14] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the datacenter," in *HotNets*, 2009.

[15] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010, pp. 267–280.

[16] E. Ng, "Maestro: A system for scalable openflow control," TSEN Maestro-Technical Report TR10-08, Rice University, Tech. Rep., 2010.

[17] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE." in *SIGCOMM*, 2010, pp. 351–362.

[18] 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 *OSDI*, 2010, pp. 351–364.

[19] Y. Ganjali and A. Tootoonchian, "Hyperflow: A distributed control plane for openflow." in *Internet Network Management Workshop / Workshop on Research on Enterprise Networking*, 2010. [Online]. Available: https://www.usenix.org/conference/inmwren-10/hyperflow-distributed-control-plane-openflow

[20] A. A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella, "Towards an elastic distributed SDN controller," in *HotSDN*, 2013, pp. 7–12.

[21] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson, "Pratyaastha: an efficient elastic distributed sdn control plane," in *HotSDN*. ACM, 2014, pp. 19–24.

[22] S. Schmid and J. Suomela, "Exploiting locality in distributed sdn control," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 121–126.

[23] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *HotSDN*. ACM, 2012, pp. 19–24.

[24] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE, 2014, pp. 569–576.

[25] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–4.

[26] M. Santos, B. Nunes, K. Obraczka, T. Turletti, B. T. de Oliveira, C. B. Margi *et al.*, "Decentralizing sdn's control plane," in *Local Computer Networks (LCN), 2014 IEEE 39th Conference on*. IEEE, 2014, pp. 402–405.

[27] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined wan," in *ACM SIGCOMM*, 2013, pp. 3–14.

[28] A. Brodersen, S. Scellato, and M. Wattenhofer, "Youtube around the world: geographic popularity of videos," in *WWW*, 2012, pp. 241–250.

[29] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese, "Netshare and stochastic netshare: predictable bandwidth allocation for data centers," *Computer Communication Review*, vol. 42, no. 3, pp. 5–11, 2012.

[30] Amazon usage estimates. `http://blog.rightscale.com/2009/10/05/amazon-usage-estimates`.

[31] Amazon S3. `http://aws.typepad.com/aws/2011/07/amazon-s3-more-than-449-billion-objects.html`.

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

[33] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs." *J. Parallel Distrib. Comput.*, pp. 96–129, 1998.

[34] M. Stoer and F. Wagner, "A simple min-cut algorithm." 1997, pp. 585–591.

[35] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

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

[37] Floodlight. `http://floodlight.openflowhub.org/`.

[38] LazyCtrl open source project. `https://github.com/yeasy/lazyctrl`.

Kai Zheng received his Ph. D and M.S. degree from Tsinghua University, China, in 2006 and 2003, respectively, and his B.S. degree from Beijing University of Posts and Telecommunications, China, in 2001. He joined IBM Research in 2006 as a Staff Researcher. Kai now is Chief Architect at Huawei Technologies. His current research interests include many fields of data center networking, software defined networking, network security, green networking, named data networking, and high performance and high availability data center network, among others. He is a senior member of the IEEE.

Lin Wang received his Ph.D. from the Institute of Computing Technology, Chinese Academy of Sciences and B.S. from the Beijing Institute of Technology in 2015 and 2010 respectively, both in Computer Science. He is now a Research Associate at SnT, University of Luxembourg. During 2012-2014, he worked in IMDEA Networks Institute, Madrid, Spain, as a research intern. His current research interests include networked systems, energy-efficient computing and large-scale data analytics.

Baohua Yang received his Ph.D. degree in Computer Science from Tsinghua University in 2013. Now he is a researcher at IBM Research. His research interests include cloud computing, system performance, networking, and distributed system. He has published tens of papers in high-quality conference and journals, and he is also TPC or Invited Reviewer of numbers of international conferences and journals.

Yi Sun received his Ph.D. degree in Computer Science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences in 2007. Now he is an associate professor at ICT since 2009. His research interests cover network resource management, network architecture, and video streaming. He has already published more than 60 academic papers, and received the Outstanding Young Scientist Award of the Chinese Academy of Sciences in 2014.

Steve Uhlig obtained a Ph.D. degree in Applied Sciences from the University of Louvain, Belgium, in 2004. From 2004 to 2006, he was a Postdoctoral Fellow of the Belgian National Fund for Scientific Research (F.N.R.S.). His thesis won the annual IBM Belgium/F.N.R.S. Computer Science Prize 2005. Between 2004 and 2006, he was a visiting scientist at Intel Research Cambridge, UK, and at the Applied Mathematics Department of University of Adelaide, Australia. Between 2006 and 2008, he was with Delft University of Technology, the Netherlands. Prior to joining Queen Mary, he was a Senior Research Scientist with Technische Universitt Berlin/Deutsche Telekom Laboratories, Berlin, Germany. Starting in January 2012, he is the Professor of Networks and Head of the Networks Research group at Queen Mary, University of London.