SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2017, Vol. 60 032202:1–032202:16
doi: 10.1007/s11432-016-0333-0

Flow Stealer: lightweight load balancing by stealing flows in distributed SDN controllers

Ping SONG[1,2*], Yi LIU[1,2*], Tianxiao LIU[1] & Depei QIAN[1,2]

[1]*Sino-German Joint Software Institute, Beihang University, Beijing* 100191*, China;*
[2]*Beijing Key Laboratory of Network Technology, Beijing* 100191*, China*

Received April 19, 2016; accepted August 25, 2016; published online January 23, 2017

Abstract Software-defined networking (SDN) introduces centralized controllers to increase network programmability drastically. Distributed control planes, in which multiple SDN controllers work together to manage a network, have been proposed to satisfy the requirements of large-scale networks, and different kinds of load-balancing approaches have been proposed to balance the workloads among these controllers. Current load-balancing approaches generally use switch migration, which adjusts the mapping between switches and controllers dynamically according to controller workloads. These switch migration-based approaches face challenges under burst traffic as a result of their overhead and longer detection periods. This paper proposes Flow Stealer, a lightweight load-balancing method for distributed SDN controllers. Flow Stealer uses a low-cost flow-stealing method, in which idle controllers share workloads temporarily with overloaded controllers by stealing flow events from them. The flow-stealing method not only can react to changes of network traffic more quickly, but can also reduce the frequency of switch migration. In addition, Flow Stealer incorporates both flow stealing and switch migration to adapt to burst traffic and long-term traffic changes. Experimental results show that Flow Stealer can balance the workloads among controllers more efficiently, especially under burst traffic.

Keywords software defined networking, load balancing, flow stealing, distributed controller, switch migration

Citation Song P, Liu Y, Liu T X, et al. Flow Stealer: lightweight load balancing by stealing flows in distributed SDN controllers. Sci China Inf Sci, 2017, 60(3): 032202, doi: 10.1007/s11432-016-0333-0

1 Introduction

Software-defined networking (SDN) isolates the control plane from the data plane in a network, thereby allowing developers to write applications to control switches directly, without concerning themselves with low-level network infrastructure. In large-scale networks, such as data center netwotks, fully physically centralized control cannot satisfy numerous remote requests from switches [1]. To improve control-plane scalability, researchers have proposed distributed SDN controllers [2–6], in which multiple SDN controllers manage disparate network regions, and each controller accesses a global shared network view to handle remote requests. Therefore, compared with a physically centralized controller, a distributed control plane can provide higher throughput and fault tolerance.

In a distributed control plane, statically configuring the map between controllers and switches can cause load imbalance among controllers, as network traffic changes continuously. This load imbalance

* Corresponding author (email: songping691@buaa.edu.cn, yi.liu@buaa.edu.cn)

problem influences the performance of the control plane and becomes a key limitation of distributed solutions. Switch migration is used by current load balance approaches [7–11] for distributed controllers, in which the mapping between controllers and switches is reconfigured dynamically to achieve balanced load distribution among controllers. In current methods, each controller detects its workload periodically and synchronizes the load information with other controllers. When a load imbalance occurs, the overloaded controller calculates a switch reassignment according to factors such as latency and distance, and then adjusts controller workload by migrating switches from overloaded to idle controllers.

In data centers, network traffic comes from applications, with some generating burst traffic during their execution, thereby causing hotspots in network regions. For example, MapReduce [12] can generate burst traffic among different nodes in the shuffling phase. In large-scale data center networks, network traffic changes frequently, and the peak flow arrival rate for a switch can be 10–100 times higher than the average [13]. This kind of situation presents challenges to current load-balancing approaches, including response latency, calculation and switch-migration overhead, and workload thrashing.

To improve the efficiency of load balancing in distributed controllers, this paper proposes Flow Stealer, a lightweight load-balancing approach for distributed SDN controllers. Inspired by work stealing [14] in task scheduling for multi-/many-core processers, Flow Stealer provides a low-cost flow-stealing method to balance workloads under burst traffic or frequent traffic fluctuations. In Flow Stealer, idle controllers share the workload of the overloaded controller temporarily by stealing flow events from overloaded controllers. The overhead of flow stealing is low, since it neither recalculates switch reassignment nor migrates switches.

If the load imbalance among controllers lasts for a long time, Flow Stealer still uses switch migration to adjust load distribution. Compared with current switch migration-based load-balancing approaches, Flow Stealer not only can react more quickly to changes in network traffic by stealing flows, but can also reduce the frequency of switch migration under burst traffic.

The remainder of this paper is structured as follows. Section 2 analyzes problems in current load-balancing approaches for distributed SDN controllers, and introduces the motivation of this paper. Section 3 proposes our solution, and Section 4 describes the implementation of Flow Stealer. Section 5 presents a performance evaluation of Flow Stealer with a discussion of the results. Section 6 presents related work, and Section 7 concludes the paper.

2 Motivation

According to measurements over real data centers [13], there are frequent hotspots in the network of data centers running MapReduce applications, because burst traffic occurs in different MapReduce execution phases, e.g., the shuffling phase needs to carry out an all-to-all communication, which generates traffic in most of the links in the network. We can also observe that the number of flows generated per second in data center networks changes frequently, and there is a significant difference between peak and median flow arrival rates, with a peak value up to 300 M and a median rate between 1.5 M and 10 M. In these situations, traffic spikes can lead some controllers in a distributed control plane to be overloaded, with the result that the resources of their controllers cannot satisfy the requirement of network regions with traffic spikes.

In current switch migration-based load-balancing approaches, each controller in the control plane checks its workload periodically, and once a load imbalance occurs, any idle controllers check to share the workload of overloaded controllers. If idle controllers are found, the switch connections of the overloaded controller are migrated to idle controllers after a switch-reassignment calculation; otherwise, additional controllers join in the control plane, with recomputed switch reassignment. This mechanism faces challenges in large-scale data centers as a result of its overhead and latency.

We analyze the overhead of load balancing in the following two respects:

(i) Switch reassignment calculation overhead. When calculating switch reassignment, multiple factors (CPU utilization, latency, distance, and memory) must be considered to find optimal reassignment,

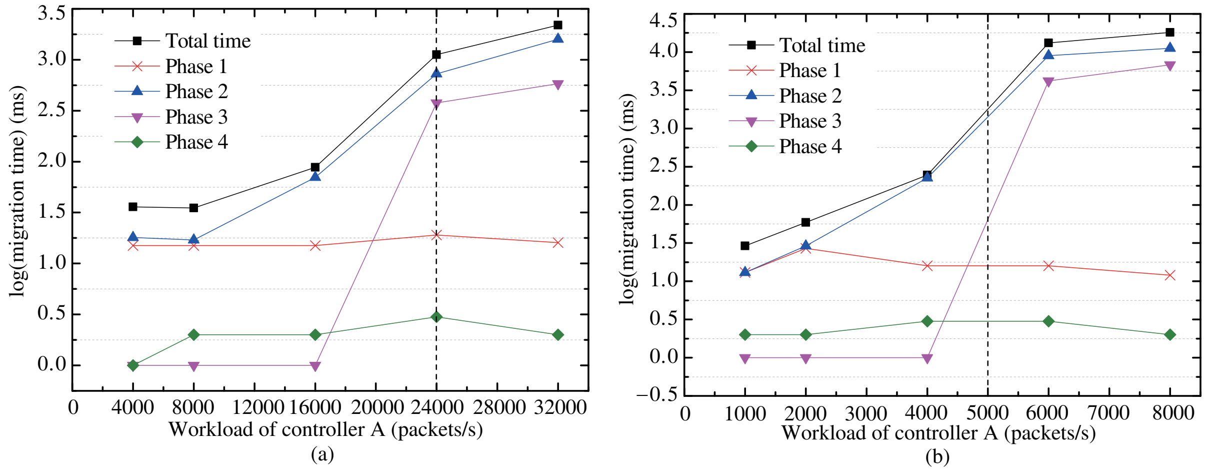

Figure 1 (Color online) Migration time under different SDN applications. (a) Learning switch application; (b) routing application.

typically formulated by researchers as an integer linear program (ILP) [11]. Therefore, various heuristic algorithms are used to compute reassignment, e.g., the greedy knapsack (GK), simulated annealing, and first-fit decreasing heuristics. Although these algorithms can discover optimal reassignment, they have a high computational complexity that limits scalability and requires developers to trade off efficiency against reactivity [8].

(ii) Switch-migration overhead. After calculating reassignment, workloads are adjusted by migrating switches using specific protocols, e.g., the four-phase switch-migration protocol proposed by ElastiCon [7]. During migration, that protocol can guarantee that the migrated switch connection remains active, while avoiding packet loss. However, the execution time of this protocol is affected by controllers' workloads and flow handling times. When a switch is migrated out from an overloaded controller, the migration time will increase from millisecond-level to second-level.

To verify our analysis, we test the migration time for various controllers. In our experiment, an initial controller A migrates one switch connection to a target controller B. Both controllers A and B execute the same application. All switches have the same flow arrival rate. Before the migration, controller B manages only one switch, and we adjust the number of switches managed by controller A to generate different workloads. The migration time is also tested under two different applications with disparate flow handling times. Learning switch application with shorter flow handling time (about 42 μs) maintains a global shared hash table which contains all of the mac tables of switches in the whole network. When handling a new flow, learning switch application will transmit flows directly by querying the hash table. Compared to learning switch application, routing application has longer flow handling time (about 200 μs). When a new flow arrives, routing application will calculate a forwarding path for the flow by Dijkstra algorithm, and configure the switches along the forwarding path.

The results are shown in Figure 1, in which the dashed line indicates the maximum throughput of controller A. As the figure shows, the migration time grows from millisecond-level to second-level as the workload of controller A increases, because the execution times of phases 2 and 3 increase dramatically. According to the four-phase switch-migration protocol, controller A must communicate with the migrated switch repeatedly during phases 2 and 3, and if controller A is overloaded, the migration messages sent to controller A cannot be handled in time, resulting in increases in the execution times of both phases 2 and 3. Furthermore, the flow handling times of different applications also affect the migration time, since they can block the handling of migration messages in controller A. Comparing Figure 1(a) with Figure 1(b), we can observe that when the controllers execute the routing application, which has a longer flow handling time, the migration time exceeds 10 s (log(total time)> 4 ms), while in a similar situation, the migration time for the learning switch application is approximately 3 s.

In addition to the problem of overhead and latency, switch migration-based approaches can cause thrashing in switch migration with burst traffic. For instance, when burst traffic occurs in a network

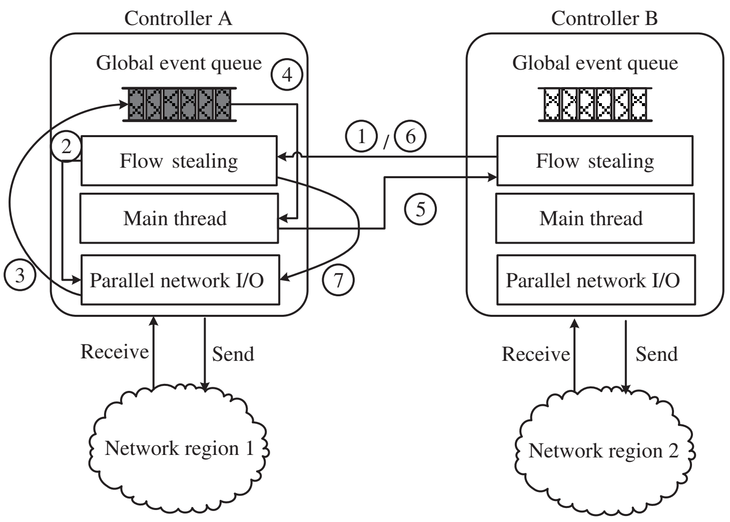

Figure 2 Flow stealing between SDN controllers.

region managed by a controller and causes it to overload, the controller will migrate some switches to its neighbor controller to balance workloads. When the burst traffic has passed, the neighbor controller can migrate switches back to minimize the standard deviation of utilization. If the burst traffic occurs periodically, switches will be migrated back and forth between controllers.

In switch migration, simply prioritizing switch-migration messages over the other switch-to-controller traffic in the overloaded controller can reduce the overhead of switch migration. However, this method neither reduces the overhead of switch-reassignment calculation nor avoids workload thrashing.

Some solutions, like DevoFlow [15], preconfigure switches statically to forward new flows directly without sending out Packet_in messages to avoid controller overload. This mechanism must notify switches regarding which flows can be forwarded directly; however, it is difficult for network managers to identify burst traffic before the network runs.

Based on the above discussion, we can see that switch migration-based approaches have overhead problems, and the detection period is expected to be relatively longer, with a longer response latency, to control overhead and avoid thrashing. A lightweight load-balancing mechanism is required to solve this problem.

3 Solution

This paper proposes Flow Stealer, in which a low-cost flow-stealing approach (mentioned in Subsection 3.1) is used to solve the load imbalance problem caused by burst traffic or frequent traffic fluctuation to improve efficiency of load balance in distributed SDN controllers. Throughout load-balance processing (mentioned in Subsection 3.2), Flow Stealer incorporates flow stealing with switch migration and balances workloads among controllers in a fine-grained manner.

3.1 Flow-stealing approach

Compared with switch migration, flow stealing is a lightweight load-balancing approach for distributed SDN controllers. In flow stealing, when a controller is idle, it steals flow events from an overloaded controller and handles them immediately without migrating switches; if the overload duration exceeds a threshold, the overloaded controller still migrates switches to other controllers to reduce its workload. By stealing flows from overloaded controllers temporarily, an idle controller can react to burst traffic more quickly. Furthermore, flow stealing can avoid switch migration in many cases, especially under burst traffic.

Figure 2 shows the process of flow stealing. During the detection period, if controller B is idle, its flow-stealing module chooses the nearest neighbor controller A as a victim, and then sends a steal request to controller A (①), which contains the IP address, port, and remaining computing resources of controller B. After receiving the request, controller A calculates the number of flows sent to controller B according to the remaining computing resources of A and B, and then the flow-stealing module of A informs all threads in the Parallel Network I/O module to offer flow events (②). All threads in the Parallel Network I/O module push flow events to the global event queue by competing for a mutex (③). If fails, the thread stops to handle the flow event without waiting for the mutex; The main thread in controller A obtains flow events from the global event queue (④). If the number of flows satisfies the steal request, the main thread informs threads in the Parallel Network I/O module to stop pushing flow events and sends all obtained events to controller B immediately (⑤). After controller B receives those flow events, multiple threads handle them concurrently and return the configuration messages to controller A (⑥). Finally, the Parallel Network I/O module of controller A sends the received configuration messages to switches to route flows (⑦).

If a controller receives a steal request while it is stealing flows, it denies the request. If a controller fails a stealing attempt, it chooses the next-nearest controller to steal from, and so on. If it fails three times, it rechecks its workload and then decides whether it must continue stealing.

Using flow stealing, controllers neither calculate switch reassignment nor migrate any switches. As a result of this low-overhead characteristic, flows can be stolen frequently among controllers in a traffic-fluctuation or burst-traffic situation. If all of the controllers are idle or overloaded, current elastic methods [7] can be used to adjust the number of controllers.

3.2 Load-balancing procedure

As mentioned previously, Flow Stealer incorporates both flow stealing and switch migration. Throughout the load-balancing procedure, Flow Stealer extends the detection period of switch migration, which can reduce the frequency of switch migration. During a detection period, Flow Stealer balances the workloads among controllers by using lightweight flow stealing, and at the end of the detection period, if a long-term load imbalance among controllers is detected, Flow Stealer calculates switch reassignment according to the status of threads, and migrates switches to achieve load balance.

If switch migration is needed at the end of a detection period, two steps will be taken to reconfigure the mapping between controllers and switches at that time: reassignment calculation and switch migration, both of which consider the status of threads.

The switch-reassignment algorithm of Flow Stealer is shown in Algorithm 1, in which each controller manages its network region and a network region can be further divided into multiple zones. In the switch-reassignment algorithm, a region contains a set of switches managed by the same controller, and a zone is a subset of its region. There is no intersection between any two zones. We use the “zone” to distinguish the switches in a region according to a switch’s alternative controller. The network region managed by an overloaded controller is referred to as an Overloaded Region (OR). The algorithm first chooses an alternative controller for every switch in an OR according to the switch-to-controller distance and divides the OR into multiple zones. For every switch in the OR, the switch-reassignment algorithm chooses the nearest non-overloaded controller as its alternative controller. Switches with the same alternative controller will comprise a zone.

Figure 3 shows an example of switch reassignment. There are five network regions in Figure 3, each managed by a unique controller. Supposing that regions 1 and 4 are the ORs, region 1 is divided into three zones according to the above process. Regions 2, 5 and 3 are the alternative controllers of zones 1–3 respectively. Region 4 cannot be an alternative controller, since it is the OR.

In a multithreaded controller, a switch connection is usually served by a unique thread, such as Floodlight[1)] and Ryu[2)]. If a controller is overloaded, there must be one or more overloaded threads in the controller. The switch-reassignment algorithm counts the “hit times” for each zone, which indicates the

1) Floodlight. Version 1.0. http://www.projectfloodlight.org/floodlight.
2) Ryu. Version 1.0. http://osrg.github.io/ryu/.

number of switches served by overloaded threads for the zone. Then, the algorithm ranks zones in descending order according to the hit times of each zone (lines 11–13). In Figure 3, the hit times of zones 1–3 are three, one and two, respectively. Thus, we attempt to migrate switches in zone 1 preferentially.

Algorithm 1 Switch reassignment algorithm

```
Input:
    Topology, load information, set of switches in an overload region $S$, set of controllers $C$, controller capacity vector, set
    of zones $Z$, set of marked switches $R$, set of alternative controllers $B$
Output:
    New assignment
 1: $Z = R = B = \emptyset$
 2: for each switch $s$ in $S$ do
 3:     /* Find a nearest controller b without overload for a switch*/
        $b \leftarrow$ get_new_controller()
 4:     $B \leftarrow b$ /*Add the alternative controller to B*/
 5:     if $b$ has a corresponding zone $z$ in $Z$ then
 6:         $z \leftarrow s$
 7:     else
 8:         create new zone $z$ in $Z$
 9:         $z \leftarrow s$
10:     end if
11:     if $s$ is served by an overload thread then
12:         $R \leftarrow s$ and rank $Z$
13:     end if
14: end for
    /*Try to migrate each zone*/
15: for each zone $z$ in $Z$ do
16:     try_migrate($z$)
17:     if success then
18:         upgrade $R$
19:     end if
20: end for
21: if satisfy Eq. (4) then
22:     return
23: end if
    /*Assign switches by GK Algorithm*/
24: while True do
25:     for each switch $s$ in $R$ do
26:         for each controller $b$ in $B$ do
27:             /*Calculate the value if migrate s to b*/
                value($b$,$s$)
28:         end for
29:     end for
30:     if a ($b$,$s$) pair has the minimum value then
31:         migrate $s$ to $b$
32:         upgrade $R$
33:     else
34:         return
35:     end if
36:     if satisfy Eq. (4) then
37:         return
38:     end if
39: end while
```

When choosing the target controller in switch migration, we use the following three limitations:

$$\sum_{i\in P_k} a_i \leqslant c_T, \quad \forall k \in Z, \tag{1}$$

$$\sum_{i\in P_k} b_i \leqslant m_T, \quad \forall k \in Z, \tag{2}$$

$$d_i T \leqslant \mathrm{distance}_{\mathrm{Max}}, \quad \forall i \in P_k,\ \forall k \in Z, \tag{3}$$

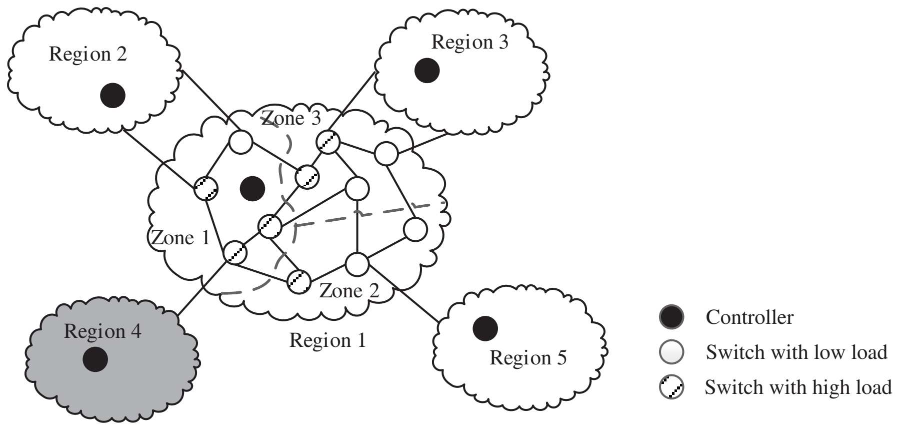

Figure 3 Example of switch reassignment.

where Z is the set of zones in an OR, k is the k-th zone in Z, P_k is a set of migrated switches in k, i is the i-th switch in P_k, T is the alternative controller of k, a_i is the computing requirement of switch i, b_i is the memory requirement of switch i, and d_iT is the distance between switch i and controller T. c_T and m_T are the computing and memory resources available in controller T, respectively, and $\text{distance}_{\text{Max}}$ is the allowed maximum distance between a switch and a controller. Eqs. (1) and (2) specify the computing and memory limits of resources, whereas Eq. (3) limits the communication cost between a switch and a controller.

The switches served by overloaded threads in a zone are marked, and only these marked switches can be migrated. There are two steps in switch reassignment. First, the algorithm accesses each zone and attempts to assign all marked switches to their corresponding alternative controllers. If Eqs. (1)–(3) are satisfied, then all marked switches in a zone will be migrated. Otherwise, the algorithm does nothing and accesses the next zone (lines 15–20). After all zones are traversed, all of the remaining marked switches in the region compose a new set R. If the size of R satisfies (4), the switch-reassignment algorithm stops. $\sum_{k=1}^{\text{Num}(Z)}\text{Num}(P_k)$ is the total number of migrated switches in network, and the default value of α is 0.5.

$$\text{Num}(R) \leqslant \alpha \sum_{k=1}^{\text{Num}(Z)} \text{Num}(P_k), \quad 0 < \alpha < 1. \tag{4}$$

If Eq. (4) is not satisfied, we use GK algorithm to continue assigning switches in R to B, which is a set of alternative controllers (lines 24–38). We model each controller in B as a knapsack. The capacity of each knapsack is equal to the processing capacity of its corresponding controller. We consider the switch as to be objects that have to be added to the knapsack, and model the weight of a switch as a_i and b_i. Each iteration activates a single controller. This controller is chosen such that the distance between the migrated switch and the alternative controller is the minimum. If Eq. (4) is satisfied or no switch can be chosen, the switch-reassignment algorithm stops.

For most current reassignment algorithms, the number of chosen switches and that of alternative controllers determine the reassignment cost. The switch-reassignment algorithm used by Flow Stealer limits the number of switches to be migrated and the number of target controllers by choosing marked switches and adjacent controllers. This algorithm not only can avoid reassigning the switches served by non-overloaded threads, but also can reduce the computing cost.

When migrating switches, Flow Stealer utilizes the four-phase switch-migration protocol proposed by ElastiCon [7]. To ensure that the migrated switches are served in time, the target controller assigns the newly created switch connection to a thread with a lower load.

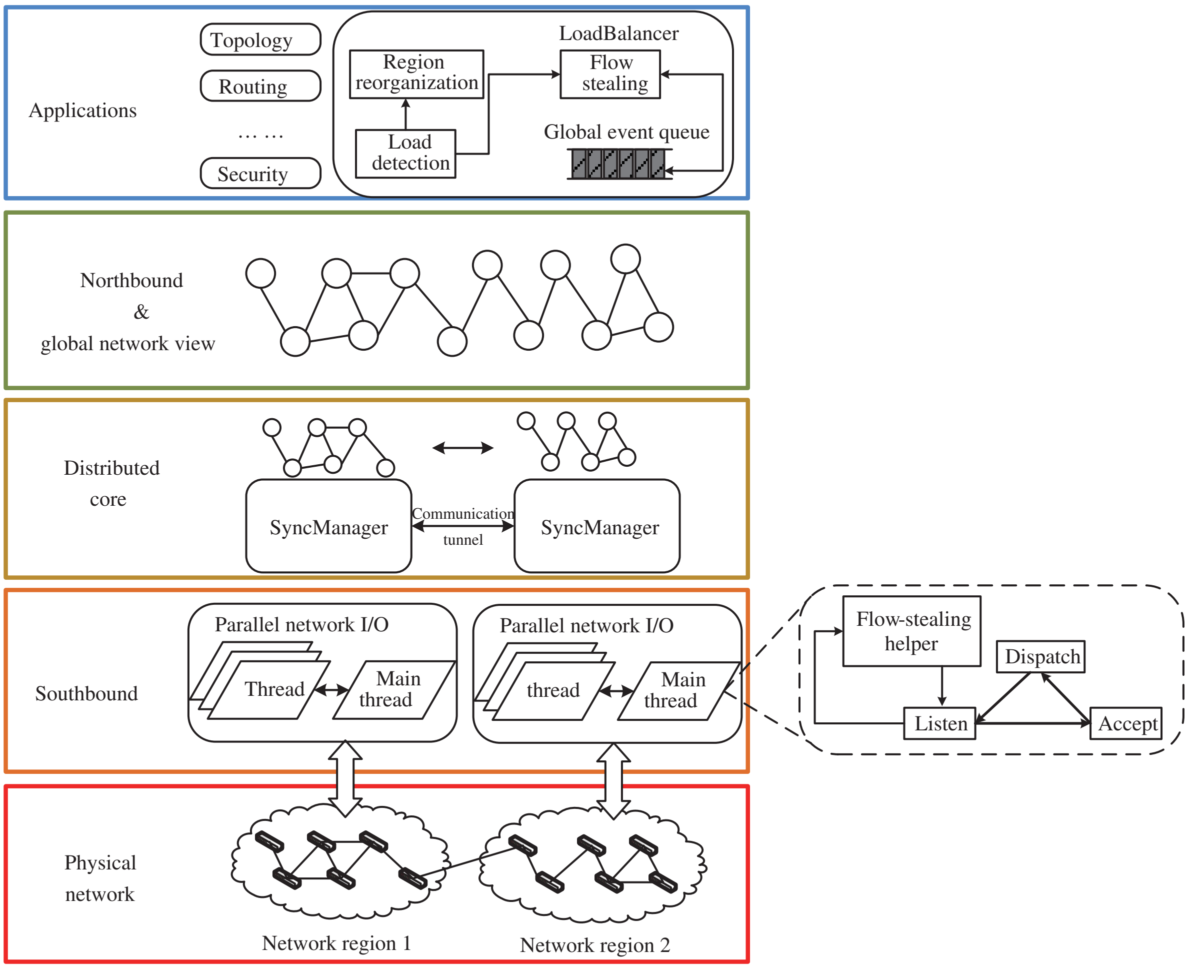

Figure 4 (Color online) Flow Stealer architecture.

4 Flow Stealer architecture

4.1 Flow Stealer overview

Flow Stealer uses a flat control plane, in which each controller manages a network region independently. The global network view and workload information are shared among controllers. A Transmission Control Protocol (TCP) connection is created between any two controllers, and information synchronization is realized using the Publish/Scribe mechanism.

We built a prototype of Flow Stealer in Java by extending a Floodlight controller. The Flow Stealer architecture is shown in Figure 4. In Flow Stealer, each controller has a unique main thread with two responsibilities. First, it listens for and accepts requests for creating new connections, then it gains a thread number from the idle thread queue, and assigns the newly created switch connection to an idle thread. If the idle thread queue is empty, the main thread assigns the switch connections by round robin. Second, during the execution of flow stealing, the main thread is in charge of notifying threads to insert flow events into the global event queue and sending flow events to the stealer.

Each thread in a controller has a unique epoll instance and is bound to a CPU core. Different switch connections are registered to disparate epoll instances to realize concurrent asynchronous network I/O. Each thread listens to the read- or write-ready events from its switch connections. As shown in Figure 5, when receiving write-ready events, the thread sends messages out directly; when receiving read-ready events, the thread pushes the events into its event queue preferentially. When there are no read- or write-ready events to receive, and the size of the event queue is larger than a threshold K (in this

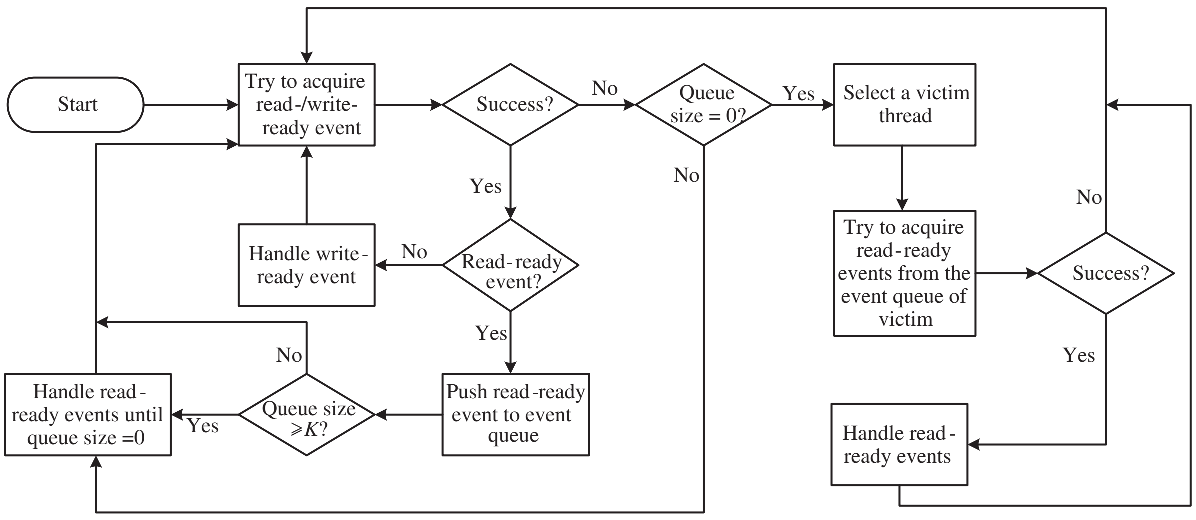
Start
Try to acquire read-/write-ready event
Success?
No
Queue size = 0?
Yes
Select a victim thread
No
Yes
Try to acquire read-ready events from the event queue of victim
Success?
No
Yes
Handle write-ready event
No
Read-ready event?
Yes
Push read-ready event to event queue
Queue size ⩾K?
No
Yes
Handle read-ready events until queue size =0
Handle read-ready events

Figure 5 Execution of threads in a controller.

paper $K = 100$), the thread will handle the read-ready events in the event queue. If the thread's event queue is empty, it chooses a victim thread randomly and attempts to acquire read-ready events from the victim thread's event queue. If the event stealing succeeds, the thread will handle the events immediately; otherwise, the thread will continue by stealing another thread until it obtains read- or write-ready events. Every thread detects its workload periodically, if a thread is idle, it will put its thread number into the idle thread queue.

LoadBalancer is a special SDN application running on the controller, which contains a global event queue and three components: Load Detection, Flow stealing, and Switch Reassignment. The Load Detection component is responsible for load calculation and synchronizing results with other controllers. It decides whether to execute flow stealing or switch reassignment according to the controller's workload.

During the detection period, if a controller is idle, its flow-stealing component will choose another controller and execute the flow-stealing method to resolve load imbalances caused by frequent traffic fluctuation or burst traffic. At the end of a detection period, if the controller's workload remains high during the period, the Switch Reassignment component will use the switch-reassignment algorithm to migrate one part of the switches to other controllers to reduce the overloaded controller's workload. The execution of the LoadBalancer application has no impact on other applications' execution.

4.2 Load calculation

As mentioned in Subsection 4.1, workloads among threads in a controller can be balanced by stealing read-ready events from overloaded threads. On this basis, the system calculates the workload of a controller according to the status of the controller's threads. According to the execution of a thread shown in Figure 5, when a thread is idle, it steals events from other threads. Therefore, the corresponding CPU core remains busy, with the result that the workload of a thread cannot be expressed by CPU utilization of each thread. Alternatively, we can use the success rate of stealing to indicate the workload of a controller, since unsuccessful stealing often indicates that the victim thread is also idle. Therefore, the more frequently stealing fails, the higher the number of idle threads in a controller, and the lower is the workload of the controller, and vice versa. Flow Stealer divides a detection period into M equivalent time intervals. The Load Detection component calculates the controller's workload during an interval by counting the stealing results, as shown in (5).

$$U(Q_m) = 1 - \frac{\sum_{j=1}^{N} \text{unsuccess\_steal}_j}{\sum_{j=1}^{N} \text{total\_steal}_j}. \tag{5}$$

In (5), Q_m is the m-th interval of the detection period H, $U(Q_m)$ is the controller's workload during

Q_m, unsuccess_steal$_j$ is the failure count of the j-th thread, total_steal$_j$ is the steal count of the j-th thread, and N is the number of threads in the controller. Developers can adjust the frequency of flow stealing by setting Q_m.

At the end of Q_m, if $U(Q_m) < \beta$, the controller executes flow stealing (the default value of β is 0.5), and the remaining computing resource U_{remain} in the steal request equals $1 - U(Q_m)$. The victim controller computes $\text{Send}_{\text{victim}}$, the number of stolen flow events according to (6).

$$\text{Send}_{\text{victim}} = \begin{cases} U_{\text{remain}} \cdot \text{Handle}_{\text{Max}}, & \text{Arrive}_{\text{victim}} \geqslant \text{Handle}_{\text{Max}}, \\ \frac{U_{\text{remain}} \cdot \text{Arrive}_{\text{victim}}}{2}, & \text{Arrive}_{\text{victim}} < \text{Handle}_{\text{Max}}. \end{cases} \tag{6}$$

In (6), $\text{Arrive}_{\text{victim}}$ is the number of arriving events in an interval, and $\text{Handle}_{\text{max}}$ is the maximum number of events handled by a controller during an interval. Eq. (6) guarantees that the victim controller provides an appropriate number of flows for the stealer controller.

At the end of a detection period H, the Load Detection component computes the average load $U(H)$ for H according to $U(Q_m)$, the workload stolen by other controllers $\text{VICTIM}(Q_m)$ and the workload stolen from other controllers $\text{STEAL}(Q_m)$, as shown in (7). If $U(H) \geqslant 0.8$, switch reassignment is triggered to balance the load.

$$U(H) = \frac{\sum_{m=1}^{M}(U(Q_m) + \text{VICTIM}(Q_m) - \text{STEAL}(Q_m))}{M}. \tag{7}$$

Flow Stealer calculates the average load of a thread during a detection period H according to (8).

$$U(\text{Thread}_i) = \frac{N_{\text{Conn}_i}}{N_{\text{Steal}_i} + N_{\text{Conn}_i}}. \tag{8}$$

In (8), Thread_i is the i-th thread in a controller ($1 \leqslant i \leqslant N$). N_{Conn_i} is the number of events coming from the connections served by Thread_i in H, and N_{Steal_i} is the number of events stolen from other threads in H. In the switch reassignment, if $U(\text{Thread}_i) \geqslant \theta$ (the default value of θ is 0.8), then Thread_i is marked as an overloaded thread.

5 Evaluation

5.1 Methodology

To evaluate the load-balancing effect of Flow Stealer, we first test the load-balancing effect and overhead of flow stealing. Then, we test the execution time of the switch-reassignment algorithm.

All of the experiments are executed on a quad-way server equipped with four six-core Intel Xeon processors and 32 GB memory, in which 24 threads can be executed simultaneously. Flow Stealer also runs on this server. Our experimental testbed uses Mininet[3)] to emulate a network of Open vSwitches[4)] that are software-based virtual Openflow switches. We run Mininet on a PC machine that is connected to the server via a Huawei switch equipped with two 10 Gb/s ports. In our experiments, we simulate network traffic using a method similar to that used in ElastiCon [7], which modifies Open vSwitch to inject Packet_In messages to the controller without transmitting packets on the data plane. We use the 4-phases switch-migration protocol proposed by ElastiCon to realize the switch-migration process in all of the experiments.

5.2 Flow-stealing test results

In the test of flow stealing, controllers run the Routing application that handles Packet_in events by using Dijkstra algorithm and returns Flow_mod messages after calculation, while Mininet computes the control-plane throughput by counting the number of received Flow_mod messages.

3) Mininet. Version 2.2.1. http://mininet.org/.

4) Open vSwitch. Version 2.4.1. http://www.openvswitch.org/.

5.2.1 *Rebalancing under different burst-traffic duration times*

To verify the advantage of flow stealing, we first test the load-balancing effect of both flow stealing and switch migration in the situation of burst traffic. In this experiment, controllers A and B manage a network containing 20 switches assigned to controllers randomly. Q_m is set at 1 s.

We use Mininet to simulate burst traffic with different duration times. Prior to the burst traffic, both controllers have the same low workload. When burst traffic arrives, the network generates 10000 Packet_in messages per second, and the flow arrival rate is distributed across all the switches in the network using a Pareto distribution. At that time, controller A is overloaded, while controller B still maintains the same workload as it had before the burst traffic. After the burst traffic, the flow arrival rate of each switch recovers its initial value. We compare the effect of flow stealing with that of switch migration under different burst-traffic duration times. In the test, flow stealing and switch migration are triggered at the same time. In switch migration, we only migrate pre-specified switches and do not execute any reassignment algorithms. The results are as shown in Figures 6 and 7.

Figure 6 shows changes of workload on controllers A and B during the test. In Figure 6, t_1 and t_2 indicate the start and end times of burst traffic, respectively, while Duration(t_1, t_2) represents the duration time of burst traffic, which is set at 5 s, 10 s, 15 s and ∞, respectively, in our test. In Figure 6, the dashed line indicates the maximum throughput of the controller. Without LB indicates that no load balancing occurs between controllers.

From Figure 6, we observe that the advantage of flow stealing is more evident when the duration is shorter, as in Figure 6(a), in which the duration is 5 s, because after detecting burst traffic, the flows of controller A are transferred to controller B immediately via flow stealing. By contrast, the workload of A decreases slowly on the switch-migration approach, since that switch requires a much longer time, when controller A is overloaded, as shown in Figure 1. With increasing Duration(t_1, t_2), the load-balancing effect of switch migration gradually becomes close to that of flow stealing. As shown in Figure 6(c), switch migration and flow stealing have a similar balancing effect after 15 s of burst traffic.

In Figure 6(d), we set t_2 as ∞, which means that the burst traffic remains stable once it occurs, or more accurately, that the traffic at this time is not "burst" but "stable". We set the detection period H at 20 s. As shown in Figure 6(d), after this detection period, switch migration occurs in the flow stealing, because controller A detects that $U(H)$ is greater than 0.8. The switch migration finishes immediately, because flow stealing and switch migration are executed concurrently, and the workload of A is lower. Figure 7 shows the average throughput of control plane during the test, from which we can observe that the throughput under flow stealing is larger, since more flow events are handled by idle controller B during the test.

5.2.2 *Rebalancing under Pareto distribution*

We also evaluate the benefit of flow stealing in the case in which multiple hotspots appear randomly and are following a Pareto distribution. In the test, controllers A and B manage a fat-tree network containing 20 switches, with each controller initially assigned 10 switches randomly. The network generates 10000 Packet_in messages per second, and the flow arrival rate is distributed across all of the switches in the network according to the Pareto distribution. We repeat the traffic pattern with five different seeds. Figure 8 shows the control plane throughput with and without flow stealing (FS).

From Figure 8, we can observe that throughput improvement varies widely depending on the seeds. If the distribution generated with a seed is more skewed, then flow stealing can achieve better throughput (Seeds 2 and 4). However, if the Pareto distribution distributes traffic across the switches on average, the performance improvement of flow stealing is not significant (Seeds 3 and 5).

Table 1 shows detailed information regarding the Pareto distribution test. From the table, we can see that the stealer can adjust the number of stolen flows based on its workload. In the case in which the distribution is highly skewed, as in Seeds 2 and 4, more flow events are transferred to the stealer.

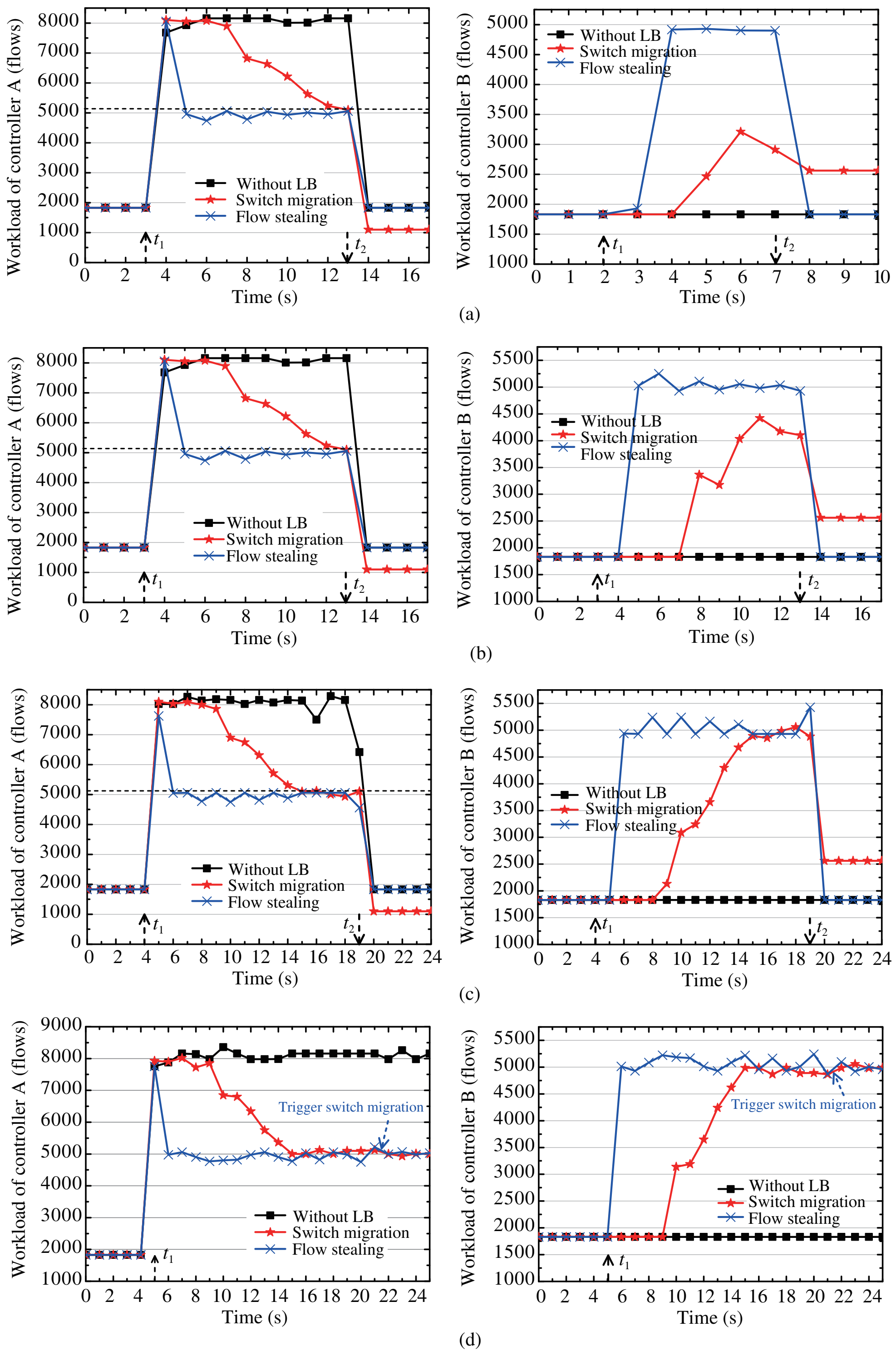

Figure 6 (Color online) Workloads of controllers A and B under burst traffic. (a) Duration$(t_1, t_2) = 5$ s; (b) Duration$(t_1, t_2) = 10$ s; (c) Duration$(t_1, t_2) = 15$ s; (d) Duration$(t_1, t_2) = \infty$.

5.2.3 *Scalability of flow stealing*

We test the throughput of control plane with different stealing frequencies and stolen workloads. In this experiment, each controller manages five switches with different flow arrival rates. We manually adjust the number of Packet_in messages sent by switches to overload half of the controllers. The results are shown in Figure 9.

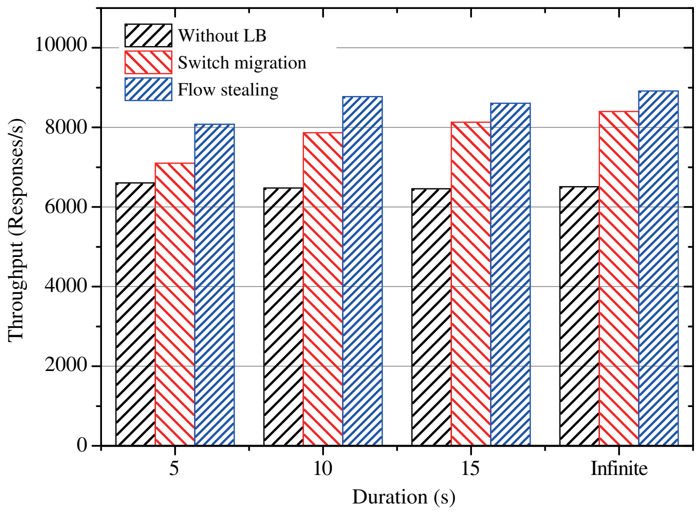

Figure 7 (Color online) Control-plane throughput under different burst-traffic durations.

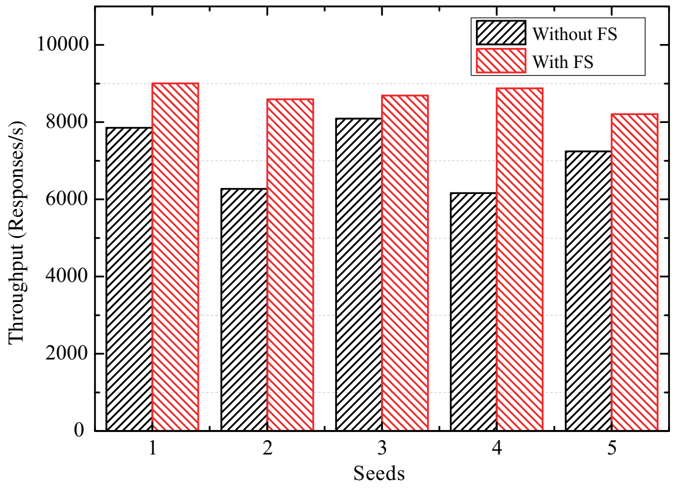

Figure 8 (Color online) Throughput improvement brought about flow stealing under Pareto distribution.

Table 1 Detailed information regarding flow stealing under a Pareto-distributed traffic pattern

Seed	Flow arrive rate of controller A (flows/s)	Flow arrive rate of controller B (flows/s)	Stealer	Remain resource proportion of stealer (%)	Stolen workload (flows/s)
1	7688	2312	B	53	2688
2	9029	971	B	80	4029
3	2446	7554	A	51	2554
4	9608	392	B	91	4608
5	1174	8826	A	76	3826

In Figure 9, Q_m is the time interval between steals; therefore, the bigger the Q_m, the more frequently the idle controller sends a stealing request. “Without FS” means “without flow stealing”. FS-i% is the percentage of workload to steal in overloaded controllers.

As Figure 9 shows, flow stealing can improve the throughput of control plane. As the number of controllers increases, the throughput improvement is more significant, since in such cases, more resources of idle controllers are used to steal and handle more flow events from overloaded controllers. Another conclusion is that stealing more flows can improve the throughput more significantly.

In Figure 9(a), when the number of controllers is 2, the throughputs of FS-20% and FS-60% are improved approximately 7% and 29%, respectively, compared to the throughput of “Without FS”. When

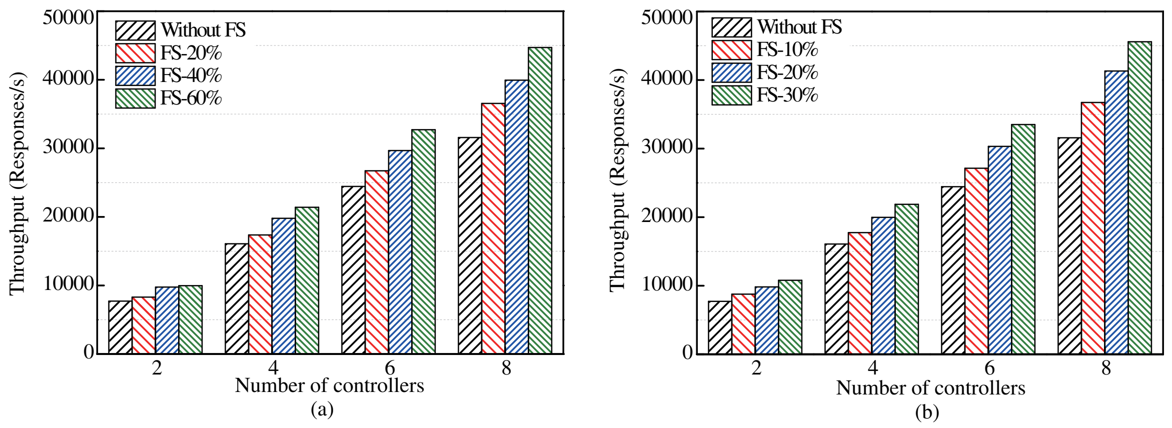

Figure 9 (Color online) Throughput under different stealing frequencies and stolen workload. (a) Length of Q_m is 1 s; (b) length of Q_m is 0.5 s.

Table 2 Average overhead of flow stealing

	Stolen flows per second		
	1000	2000	3000
Average communication delay (ms)	11.3	30.8	49.9
Number of additional messages exchanged between controllers	13	22	29
Average number of additional flows handled by stealer (flows/s)	846	1732	2474

the number of controllers is 8, the throughputs of FS-20% and FS-60% are improved approximately 16% and 42%, respectively.

Comparing Figure 9(a) and (b), we can see that higher stealing frequency brings about larger throughput when controllers steal the same workload every time. For example, when there are eight controllers in the control plane, the throughput of FS-20% is improved approximately 16% in Figure 9(a), while the throughput of FS-20% is improved approximately 31% in Figure 9(b).

5.2.4 *Overhead of flow stealing*

In flow stealing, there is overhead resulting from queue operation and transmission delay. We test the average overhead of flow stealing when stealing different numbers of flows. In this experiment, two controllers manage different numbers of switches with the same flow arrival rate (1000 flows/s), with one controller overloaded and the other idle. Each test lasts 10 s, and Q_m is set as 0.5 s. The results are shown in Table 2.

From Table 2, we observe that the average communication delay of flow stealing is quite low, and it increases gradually with the increasing number of stolen flows, since the number of additional messages exchanged between controllers increases when more flows are stolen.

As the network scales up, more controllers will be used to manage more switches, and burst traffic will occur frequently with higher flow arrival rate. In this situation, more overloaded controllers need to be stolen, and flow stealing happens more frequently among controllers, which results in the growth of SDN control-plane overhead. However, the time cost of flow stealing is quite low, meanwhile, higher stealing frequency brings about larger throughput when controllers steal the same workload every time (as shown in Subsection 5.2.3).

If the flow arrival rate is high when burst traffic occurs, idle controllers need to steal more flow events from the overloaded controllers, which results in a less growth of flow stealing time cost. Nevertheless, stealing more flows can also bring about a higher control-plane throughput according to the test results in Subsection 5.2.3.

As mentioned in Section 2, the switch-migration overhead increases dramatically when the flow arrival rate is high. If the load imbalance among controllers lasts for a long time, Flow Stealer still uses switch migration to adjust load distribution. In this situation, flow stealing and switch migration are executed

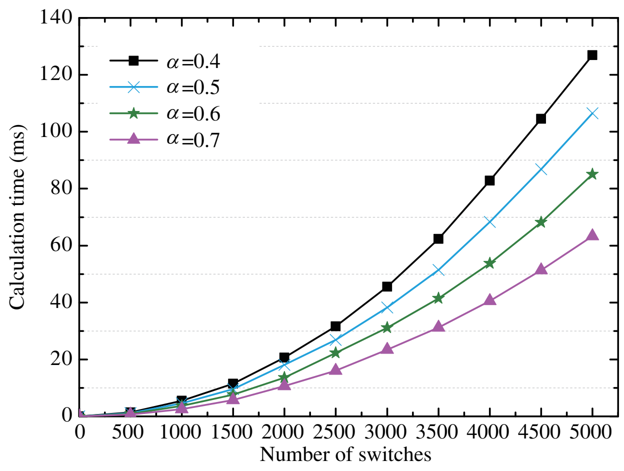

Figure 10 (Color online) Calculation time of switch-reassignment algorithm.

concurrently, which brings about a low switch-migration overhead (as shown in Subsection 5.2.1).

5.3 Switch reassignment test results

We test the overhead of the switch-reassignment algorithm proposed in this paper. In this experiment, we use three controllers to manage a network with 0–5000 switches in a random topology. The flow arrival rate of each switch and the distance between a switch and a controller are also set randomly. In the beginning, each controller manages a similar number of switches. During the test, one of the controllers is overloaded, while the others have lower loads. We test the calculation time of the algorithm with different network scales. In addition, since α in (4) can affect the number of migrated switches in the switch-reassignment algorithm, we set α at different values to observe the effect of α on the calculation time.

As shown in Figure 10, the calculation time of the switch-reassignment algorithm grows along with the number of switches. Under the same network scale, a smaller α results in a longer calculation time. Furthermore, the calculation time grows more quickly when α is smaller, primarily because a smaller α causes more switches to be migrated, generally implying a longer calculation time.

6 Related work

Researchers have proposed a series of load-balancing approaches for distributed SDN controllers. ElastiCon [7] is an elastic distributed controller architecture in which the controller pool is dynamically grown or shrunk according to traffic conditions. It also involves a Load Adaptation algorithm to detect and adjust workload among controllers periodically. Pratyaastha [8] is also an elastically distributed controller architecture. Differently from ElastiCon, it considers the memory resource allocation during the switch assignment to minimize the cost of accessing the network status. BalanceFlow [10] uses a centralized super controller to partition all switch pairs in a network dynamically for the sake of realizing load balancing. To solve the problem of BalanceFlow, DALB [9] proposes a new load-balancing method in which every controller in the control plane makes decisions independently, and load balancing is realized by negotiation among controllers. Bari et al. [11] synthesize the costs of statistics collection, flow setup, synchronization, and switch reassignment to adjust the mapping between switches and controllers using the GK and simulated annealing algorithms.

All of the above solutions use switch migration to balance workloads among controllers, that is, to adjust the map between controllers and switches dynamically. In contrast, this paper uses flow stealing, a lightweight load-balancing method, to balance workloads among controllers. Furthermore, our Flow Stealer incorporates both flow stealing and switch migration to adapt to burst traffic as well as long-term traffic change.

In task scheduling for multi- or many-core processors, researchers have proposed the work-stealing algorithm [14] in which an idle processor can steal tasks from other processors to balance loads. Compared to traditional thread-migration-based task scheduling approaches, work stealing reduces the frequency of thread migration and has lower cost. Currently, work stealing has been used in runtime systems of programming language extensions to achieve fine-grained parallelism, e.g., Cilk [16] and TBB [17].

7 Conclusion

Distributed SDN controllers are generally used in large-scale data center networks to provide higher throughput and scalability. Statically assigning switches to controllers can easily cause load imbalance among controllers owing to dynamic traffic changes. However, current load-balancing approaches for distributed SDN controllers generally use switch migration to balance workloads among controllers, but this is not suitable for burst traffic owing to its overhead and longer detection periods.

This paper proposes Flow Stealer, a lightweight load-balancing approach for distributed SDN controllers. To improve load-balancing efficiency under burst traffic, Flow Stealer uses the flow-stealing method, in which idle controllers share workloads with overloaded controllers temporarily by stealing flow events from them. Experimental results show that Flow Stealer can balance the workloads among controllers more efficiently than other approaches.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 91530324) and National High Technology Research and Development Program of China (863) (Grant No. 2015AA01A301).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Voellmy A, Wang J. Scalable software defined network controllers. In: Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Helsinki, 2012. 289–290
2 Tootoonchian A, Ganjali Y. Hyperflow: a distributed control plane for openflow. In: Proceedings of the Internet Network Management Conference on Research on Enterprise Networking, Berkeley, 2010. 3
3 Hassas Yeganeh S, Ganjali Y. Kandoo: a framework for efficient and scalable offloading of control applications. In: Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks, Helsinki, 2012. 19–24
4 Koponen T, Casado M, Gude N, et al. Onix: a distributed control platform for large-scale production networks. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, Vancouver, 2010. 1–14
5 Tam A, Xi K, Chao H J. Use of devolved controllers in data center networks. In: Proceedings of the IEEE INFOCOM Workshop on Cloud Computing, Shanghai, 2011. 596–601
6 Phemius K, Bouet M, Leguay J. DISCO: distributed multi-domain sdn controllers. In: Proceedings of the IEEE/IFIP Network Operations and Management Symposium, Krakow, 2014. 1–4
7 Dixit A, Hao F, Mukherjee S, et al. Towards an elastic distributed sdn controller. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, 2013. 7–12
8 Krishnamurthy A, Chandrabose S, Gember-Jacobson A. Pratyaastha: an efficient elastic distributed SDN control plane. In: Proceedings of the 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Chicago, 2014. 133–138
9 Zhou Y, Zhu M, Xiao L, et al. A load balancing strategy for SDN controller based on distributed decision. In: Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, Beijing, 2014. 851–856
10 Hu Y, Wang W, Gong X, et al. Balanceflow: controller load balancing for Openflow networks. In: Proceedings of the IEEE 2nd International Conference on Cloud Computing and Intelligence Systems, Hangzhou, 2012. 2: 780–785
11 Bari M, Roy A, Chowdhury S, et al. Dynamic controller provisioning in software defined networks. In: Proceedings of the 9th International Conference on Network and Service Management, New York, 2013. 18–25
12 Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM, 2008, 51: 107–113
13 Benson T, Akella A, Maltz D. Network traffic characteristics of data centers in the wild. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, Melbourne, 2010. 267–280
14 Blumofe R D, Leiserson C E. Scheduling multithreaded computations by work stealing. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, 1994. 356–368
15 Curtis A R, Mogul J C, Tourrilhes J, et al. Devoflow: scaling flow management for high-performance enterprise networks. ACM SIGCOMM Comput Commun Rev, 2011, 41: 254–265
16 Joerg C, Blumofe R. Cilk: an efficient multithreaded runtime system. J Parall Distrib Comput, 1996, 37: 55–69
17 Pheatt C. Intel threading building blocks. J Comput Sci Coll, 2008, 23: 298