BALANCEFLOW: CONTROLLER LOAD BALANCING FOR OPENFLOW NETWORKS

Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, Shiduan Cheng

State Key Laboratory of Networking and Switching Technology
Beijing University of Posts and Telecommunications, Beijing, 100876, China
{huyannan, wdwang, xygong, rongqx, chsd}@bupt.edu.cn

Abstract: In the discussion about Future Internet, Software-Defined Networking (SDN), enabled by OpenFlow, is currently seen as one of the most promising paradigm. While the availability and scalability concerns rises as a single controller could be alleviated by using replicate or distributed controllers, there lacks a flexible mechanism to allow controller load balancing. This paper proposes BalanceFlow, a controller load balancing architecture for OpenFlow networks. By utilizing CONTROLLER *X* action extension for OpenFlow switches and cross-controller communication, one of the controllers, called "super controller", can flexibly tune the flow-requests handled by each controller, without introducing unacceptable propagation latencies. Experiments based on real topology show that BalanceFlow can adjust the load of each controller dynamically.

Keywords: OpenFlow; Load balancing; Scalability; Software-defined networking; Future Internet

1 Introduction

After decades of development, the Internet has made great success. However, many challenges have emerged during this process, which call for new Internet architectures. In the discussion about Future Internet, Software-Defined Network (SDN) is currently seen as one of the most promising paradigm. Unlike traditional networks, SDN decouples the control and data planes of the network. The data plane is a packet-forwarding abstraction whereas the control plane is a centralized network-wide operating system providing open APIs for network applications and services. In SDN, the data plane is con-trolled by the control plane through a well-defined API.

OpenFlow [1] provides a mechanism for SDN. While OpenFlow was first proposed as a way to enable researchers to conduct experiments in campus networks, its advantages lead to its use beyond that. OpenFlow's central-control model can avoid the need to construct global policies from switch-by-switch configurations, and can also support near-optimal traffic management [2]. As such, OpenFlow is being deployed in various networks, such as wide-area networks, enterprise networks and data center networks, and has become the basis of many recent research papers [3-5].

978-1-4673-1857-0/12/$31.00 ©2012 IEEE

Since any given OpenFlow controller instance can support only a limited number of flow setups [6,7], relying on a single controller imposes a potential scalability problem [2,8]. To tackle this problem, researchers have proposed distributed OpenFlow controller implementations, such as HyperFlow [9] and Onix [10]. These implementations use multiple controllers to manage networks too large to be controlled by a single controller. Independently, the upcoming Internet2 OpenFlow production deployment [11] also suggests of using multiple controllers to manage the large wide-area network.

Although it is very clear that the use of multiple controllers improves the scalability of the control plane, it opens the question of how to maintain a map of forwarding devices and controllers [12], such that none of these controllers is overloaded. For example, when deploying multiple controllers, the assignments of switches to controllers need to be carefully considered, and the problem becomes extremely difficult with the dynamic changes of traffic condition. In this paper, instead of balancing the load between controllers by statically network planning, we argue that controller load balancing should be an indispensable primitive of the distributed control plane. Flow-requests should be dynamically distributed among controllers to achieve quick response. The load on an overloaded controller should be automatically transferred to appropriate under-loaded controllers to maximize the aggregate controller utilization.

This paper proposes BalanceFlow, a controller load balancing architecture for wide-area Open-Flow networks, which can partition control traffic load among different controller instances in a more flexible way. Unlike previous distributed control plane implementations, such as HyperFlow [7], which worked at the granularity of switches, BalanceFlow works at the granularity of flows. We embrace the multiple controllers feature of the latest OpenFlow standard and further propose a reasonable extension for OpenFlow switches: CONTROLLER X action. All controllers in Balance-Flow maintain their own load information and publish this information periodically through a cross-controller communication system. Upon traffic condition changes, one of the BalanceFlow controllers, called "super controller", partitions the traffic, and reallocates different flow setups to appropriate controllers, with the primary objective of having no

imbalance among controllers at run time. Besides, as propagation latency bounds the control reactions with a remote controller for WANs [12], allocating flow-requests to distant controllers should be forbidden as well to avoid unreasonable responses delay. We present an efficient traffic partition heuristic to achieve these goals. Our evaluation based on Abilene topology shows significant benefit of BalanceFlow on controller load balancing.

The rest of this paper is organized as follows. Section 2 discusses the requirements for OpenFlow switches to support controller load balancing. Section 3 describes BalanceFlow architecture. Section 4 evaluates our proposal and partition algorithm. Finally, we conclude the paper in Section 5.

2 Requirements for controller load balancing

2.1 Simultaneous multiple controller connections

The early version of OpenFlow standard only supports for single connection setup between each switch and its controller. When multiple controllers are deployed, one has to connect different subsets of switches to different controllers. However, it would be impossible to allow controller load balancing, since the flow setups cannot be directly shared among controllers[1]. Therefore, it seems that support for multiple simultaneous controllers is desirable. Fortunately, we are not the first to make this observation: the latest OpenFlow standard, such as OpenFlow 1.2 or OpenFlow 1.3, has already support for multiple simultaneous controller connections. This mechanism enables controller load balancing if the flow-requests can be carefully distributed.

2.2 Controller *X* action

There are many ways to spread switches' flow-requests over multiple controllers. A simple approach is using hashing. A switch could send each message to one of the controllers, which is determined by the hash of the message. In this way, the load of different controllers is totally determined by the hash distribution in the switches. However, this may lead to a complex switch design. Moreover, a key limitation of the hash method is that two or more large volumes of flow-requests may collide and end up on the same controller, creating potential controller overload. Instead, we argue that, while the switches should be responsible for distributing their flow-requests, how to distribute them should be entirely determined at the controller level. With that in mind, we propose an alternative flow-requests spread mechanism by introducing CONTROLLER *X* action extension to OpenFlow switches. By using this extension, flow-requests allocation can be dictated by controllers in a more flexible way: The controllers reactively or proactively install fine-grained or aggregated flow entries with CONTROLLER *X* action on each switch, and different flow-requests of each switch thus can be allocated to different controllers. In other words, whenever a new flow arrives at a switch, the first packet of that flow matches one of these entries, and is sent to corresponding controller according to the *X* in the action part for further process. Upon controller failure, active controllers could simply update the flow entries in the related switches to reallocate the flow-requests.

[1] Just to be clear, traditional control plane is not designed to allow sharing flow requests among multiple controllers (unless the network is "physically sliced" [5]).

3 BalanceFlow architecture

In this Section, we propose BalanceFlow, a controller load balancing architecture for wide-area OpenFlow networks. As shown in Figure 1, BalanceFlow network consists of OpenFlow switches, multiple controllers and a cross-controller communication system. The switches act as the forwarding elements, and connect to multiple controllers simultaneously. The controllers are all together function as a centralized control plane of the network. While they cover the whole network, each controller handles only a portion of the total flow installs. All controllers in BalanceFlow maintain their own flow-requests information and publish this information periodically through a cross-controller communication system to support load balancing. There are two types of controllers in BalanceFlow network, one "super contro-ller" and many normal controllers. The super controller is responsible for balancing the load of all controllers, and has a control loop of three basic steps. First, it collects flow-requests information from all controllers. Next, it partitions the traffic according to the flow-requests information when controller load imbalance is detected, and generates allocation information. Finally, allocation rules are installed on switches.

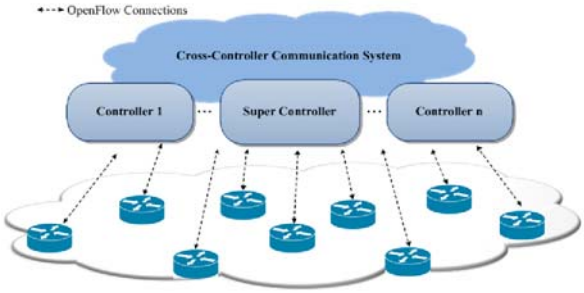

Figure 1 BalanceFlow architecture

In rest of this section, we first describe how each controller in BalanceFlow network constructs its flow-requests information which can be utilized by the super controller to tune the load of controllers. Next, we describe how the super controller detects imbalance and generates flow-requests allocation information, along with our partition algorithm that finds a tradeoff between controllers' load and propagation latency. Finally, we show that the allocation information can be expressed with allocation rules on OpenFlow switches.

3.1 Constructing flow-requests information

Besides handling flow setups, all controllers in BalanceFlow network should maintain their flow-requests information separately and publish the information through the cross-controller communication system periodically. On the surface, the simplest approach to represent the number of flow-requests received by each controller is using a single statistic. However, from the super controller's point of view, this information is meaningless, since a more detailed view of the flow-requests is needed in order to achieve load balancing. An alternative representation is the statistics of each microflow. However, this approach does not scale well in a large network. For example, a network with tens of thousands of nodes might involve billions of microflows; there would be too much data stored in the controllers, and it is also not feasible to exchange these data through the cross-controller communication system. Therefore, it seems that we need to find a tradeoff between informative and scalable. In this work, we represent the flow-requests information based on the flow statuses between switch pairs. This representation is viable since the total number of switch pairs is usually limited, and our evaluation in Section 4 also suggests that it is informative enough to achieve controller load balancing.

In BalanceFlow, each controller k maintains an $N \times N$ matrix M_k, where N is the number of switches in the network. The element in the *ith* row, *jth* column denotes the average number of flow-requests from switch i to switch j. Upon receiving a flow-request packet, the controller first knows from which switch the packet comes. By checking its destination address, the controller locates the corresponding egress switch for that flow, and the relevant element in the matrix is updated periodically. The average number of flow-requests from switch i to switch j is calculated using the following formula

$$R_{avg}(i,j) = (1 - w)\, R_{avg}(i,j) + wT(i,j) \quad (1)$$

Where w is a weighted coefficient, and $T_{(i,j)}$ is the number of flow-requests from switch i to switch j in a certain period of time.

3.2 Flow-requests partition and allocation

The super controller is responsible for collecting the flow-requests matrixes published by all controllers. The average number of flow-requests handled by each controller and the total number of flow-requests in the network are then calculated. Upon controller load imbalance is identified, the super controller runs flow-requests partition algorithm and reallocates different flow setups to different controllers.

Detecting imbalance: The super controller detects controller load imbalance when the average number of flow-requests handled by any controller contributes more than some threshold of the total flow-requests rate in the network. The threshold should be adjustable according to the performance of the super controller, the number of controllers and the network environment.

Generating allocation information: The super controller runs partition algorithm, which allocates flow-requests to multiple controllers and generates allocation information. In an acceptable allocation, the total number of flow-requests handled by any controller shall not exceed the threshold that triggers traffic reallocation. Meanwhile, one thing is important to mention: we should not allocate large volumes of flow-requests to the under-loaded controllers, which, however, could incur undesired node-to-controller delays, since the propagation latencies is the fundamental limits imposed on reaction delay, fault discovery, and event propagation efficiency for WANs [12]. In order to achieve this, we developed a heuristic algorithm, shown in Figure 2.

Algorithm 1: Flow-request partition algorithm

Data: set of average flow-requests number of all switch pairs M, k = number of controllers, set of node-to-controller latencies in the network

```
1  Sort M in descending order of average flow-requests number
2  foreach descending ordered switch pair (i,j) do
3      /* Allocate into a controller */
4      for s := 1 to k do
5          c_s := cost of adding switch pair (i,j) to controller s
6      end
7      Allocate switch pair (i,j) to the controller t = arg min c_t
8  end
```

Figure 2 Flow-request Partition Heuristic

The algorithm is in two parts. Firstly it sorts all switch pairs in descending order of their average number of flow-initiations. Then, it allocates each of the descending ordered switch pairs into one of the controllers according to a cost function. As a result, the generated allocation information is a mapping between k controllers and the set of switch pairs.

The essential part of the algorithm is the lines 3-6. It allocates the switch pairs *(i,j)* to controllers according to a cost function. The goal of the cost function is to allocate the each switch pair into controllers so that the maximum number of flow-requests handled by a controller and the average propagation delays are minimized. We thus define the cost function as

$$C_s = Q_s + P_{(i,j)} + \alpha\, v_M L_{(i,s)} \quad (2)$$

Where Q_s is the number of flow-requests already handled by controller s at the moment, $P_{(i,j)}$ is the flow-requests that is about to be handled by that controller, and $L_{(i,s)}$ is the propagation delay from switch i to controller s. v_M is a moderator variable that makes controllers' load and propagation latencies comparable. We define $v_M = R_{total} / L_{avg}$, where R_{total} is the total number of flow-requests in the network, and L_{avg} is the minimum average node-to-controller latencies (measured in milliseconds) given a placement of k controllers. Parameter α adjusts the weights between controllers' load and propagation latencies in the cost function. When setting α too small, we ignore the benefit of reducing latencies. When setting α too large, however,

we do not require the load to be balanced. We empirically found that α between 0.075 and 0.15 gives a good result. We set $\alpha = 0.1$ in our experiments but the wide range of appropriate values for α suggests that it is not very sensitive.

3.3 Representing allocation information as allocation rules

Once we have the allocation information, the immediate questions are:

1) How to represent this information as low-level rules in switches;

2) How these rules co-exist with other OpenFlow rules.

For the representation problem, given a controller t, the relevant low-level rules can be defined as CONTROLLER t action (described in Section 2) on a set of points P_t in a flow space. Assume t handles flow-requests from a set of switch pairs S_t, then we need a mapping between P_t and S_t. Thanks to the central-control model of OpenFlow, a controller can know about all the potential flows and their ingress-egress switch pairs. We named the low-level representation of allocation information "allocation rules". In Balance-Flow, these allocation rules are host-pair-based, and all flows from host pairs attached to the same sets of switch pairs match allocation rules with the same CONTROLLER X action. For example, considering a simple BalanceFlow network with two controller (A and B), three switches and five hosts, the host-switch relationship and the addresses of hosts are shown in Figure 3. If all the flow-requests from switch 1 to switch 3 are assigned to controller A, and the first allocation rule in Figure 4 could represent this allocation.

Switch Number	Host address
1	1100
2	0101
	0110
	1001
3	0011

Figure 3 An illustration of host-switch relationship

Type	Priority	Matching Field	Action	Timeout	Note
Normal rules	220	IP dst = 0011	Output: port 2	10s	Installed by A
	219	IP src = 1100, IP dst = 0101	Drop	10s	Installed by B
	...	...	...	...	
Allocation rules	14	IP src = 1100, IP dst = 0011	Encap, forward to controller A	Permanent	Switch 1 to Switch 3
	14	IP src = 1100, IP dst = 01**	Encap, forward to controller B	Permanent	Switch 1 to Switch 2
	14	IP src = 1100, IP dst = 1001	Encap, forward to controller B	Permanent	
	...	...	...	...	...
	1	*	Encap, forward to controller A	Permanent	Default

Figure 4 Rules in switch 1 (Controller A is the super controller)

However, caching separate allocation rules for each host pair under the same switch pairs, while conceptually simple, consumes more data-plane memory on the switches. Therefore, we use a simple approach to overcome this problem, where we try to allocate the hosts attached to the same switch aggregatable addresses as long as possible. As such, the allocation information of a switch pair can be represented by an aggregatable allocation rule (e.g. the second allocation rule in Figure 4) and a few unaggregatable allocation rules (e.g. the third allocation rule in Figure 4).

We now consider the coexistence problem. All Balance-Flow switches contain two sets of rules.

Normal rules: Normal rules are standard OpenFlow flow entries with standardized actions, such as drop or forward. These rules are installed by different controllers, so that most of the data traffic can be processed by the switches themselves.

Allocation rules: Allocation rules are the expression of allocation information and installed by the super controller. In contrary to normal rules which have a limited timeout, the timeout of the allocation rules is set to permanent to make sure a flow-request can always be answered by controllers. The two sets of rules are of different priorities. Priorities are supported by OpenFlow. Packets matching multiple rules are processed based on the rule which has the highest priority. The normal rules have highest priority because packets matching these rules do not need to be sent to controllers. Allocation rules have lowest priority, and the first packets of new flows always match allocation rules and are sent to different controllers to achieve controller load balancing.

4 Evaluation

In order to provide some intuition for how BalanceFlow would achieve controller load balancing, we first implemented a simulator. The simulator is event-based, and whenever a flow is started, the relevant controller receives the flow-request immediately. The corresponding element of its matrix is recomputed using formula (1) every T_m seconds, and we set the weighted coefficient w to 0.002 empirically. The flow-request matrixes are analyzed every T_a by the super controller to detect imbalance. We also model the load allocation information generated by the super controller as mappings between different source-destination pairs and different controllers. Upon controller load imbalance the simulator reallocates the flow-requests among controllers using the partition algorithm present in Section 3.2.

We evaluate our Balance architecture on Abilene topology [13], which contains 10 nodes and 13 links. In our simulations, ten hosts are attached to each switch, and three controllers (denoted as A, B and C for simplicity in illustrating) are deployed in the network. We adopt the controller placement strategy in Ref. [12] that minimum the average node-to-controller latency of the whole network. As a result, the default behavior of each switch is to forward its flow-requests to its nearest controller to achieve quick response, and controller A, B

and C will receive the flow-requests from 4, 3, and 3 switches, respectively.

In our experiments, we set T_m to 10ms, T_a to 1s, and the controller load imbalance threshold be 45%. The flow-request reallocation function is disabled at first. At time 0 second, each host starts to send a one packet flow to a randomly-chosen host. After that, a host randomly waits between 0 and 10ms before sending a new flow. At time 70 seconds, we intentionally increase the load of controller A (by raising the rate of new flows between a randomly chosen host pair, whose flow-requests are typically handled by controller A), since controller load imbalance is more likely to happen in this circumstance. At time 120 second, we "turn on" the super controller, and it reallocates the traffic if imbalance is detected. We repeated the test 20 times, and Figure 5 plots the average load of the three controllers over time. As the hosts start sending traffic, the average flow-requests received by each controller ramp up, and the division of load is relatively close to the 4:3:3. After 70 seconds (indicated by the first vertical dashed line), as expected, the load on controller A increases dramatically, and it contributes roughly 50% of the total flow-requests rate in the network. Fifty seconds later (indicated by the second vertical dashed line), the flow-requests are reallocated, and the load of each controller is well under the imbalance triggering threshold.

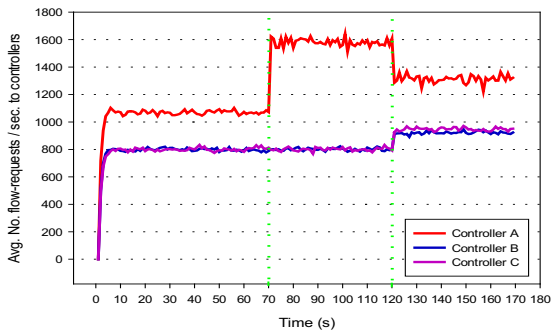

Figure 5 The average number of flow-request handled by three controllers

Recall that, the allocation of a flow-request to a distant controller should be forbidden to avoid unacceptable responses delay. To evaluate the propagation latencies introduced by traffic transitions, we compare the average node-to-controller delay of the whole network after reallocation with the optimal latency and the most rigorous latency deadline discussed in Ref. [12].

Table I shows latency-to-deadline ratios in different situations. A ratio of 1.0 indicates that the latency is equal to the most rigorous latency deadline. The data shows that the mean value of the average delay is only about 8.7% larger than the optimal value. Moreover, we can see that our algorithm can succeed in tuning the controllers' load without sacrificing propagation delays, since even the worst case average delay after traffic reallocation is still well below the target deadline.

Table I Latency-to-deadline ratios in different situations, the lower the better

After reallocation			optimal
Min	Mean	Max	
0.7028	0.7213	0.7432	0.6632

5 Conclusions

This paper proposes BalanceFlow, which can handle controller load balancing entirely at the controller level. We introduce an extension for OpenFlow switches: CONTROLLER *X* action. Upon controller load imbalance, the super controller runs partition algorithm and reallocates the load of different controller by distributing allocation rules to switches. Based on the evaluation of our BalanceFlow architecture, we show that BalanceFlow can flexibly adjust the workload of each controller, and succeed in finding a balance between controllers' load and average propagation latencies of the whole network.

Acknowledgements

This work was supported in part by the National High-tech Research and Development Program (863 Program) of China under Grant No. 2011AA01A101, National Basic Research Program of China (973 Program) under Grant No. 2009CB320504, National Natural Science Foundation of China under Grant No. 61271041 and Fundamental Research Funds for the Central Universities.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, OpenFlow: Enabling innovation in campus networks, ACM Computer Communication Review, Apr. 2008.

[2] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In Proc. SIGCOMM, 2010.

[3] R. Bennesby, P. Fonseca, E. Mota, and A. Passito. An Inter-AS Routing Component for Software-Defined Networks. In Proc. NOMS, Apr. 2012.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proc. NSDI, Apr. 2010.

[5] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In Proc. OSDI, Oct. 2010.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfa, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In SIGCOMM CCR, July 2008.

[7] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In HotNets, 2009.

[8] A. R. Curtis, J. C. Mogul, J. Tourrihes, and P. Yalagandula. DevoFlow: Scaling Flow Management for High-Performance Networks. In Proc. SIGCOMM, 2011.

[9] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In Proc. INM/WREN, San Jose, CA, Apr. 2010.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In Proc. OSDI, Oct. 2010.

[11] Internet2 open science, scholarship and services exchange. http://www.internet2.edu/-network/ose/.

[12] B. Heller, R. Sherwood, and N. McKeown. The Controller Placement Problem. In Proc. HotSDN, Aug. 2012.

[13] http://www.internet2.edu/network.