OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support

Yu Li and Deng Pan
Florida International University
Miami, FL

***Abstract*—Data center networks are designed for satisfying the data transmission demand of densely interconnected hosts in the data center. The network topology and routing mechanism can affect the performance and latency significantly. Nowadays, the fat-tree network is one of the most widely used topologies for data center networks. Network engineers also adopt load balancing methods in the design of routing algorithms. However, the requirement of load balancing routing in fat-tree networks cannot be fully satisfied by traditional approaches. The main reason is the lack of efficient ways to obtain network traffic statistics from each network device. As a solution, the OpenFlow protocol enables monitoring traffic statistics by a centralized controller.**

To achieve high performance and low latency, we present a load balancer for OpenFlow based data center networks. We implement a dynamic routing algorithm in the load balancer. The task of the algorithm is to distribute traffic of upcoming network flows and make each alternative path receive equal amounts of traffic load. It can apply to large scale networks and schedule data flows dynamically. Our implementation uses the OpenFlow controller Beacon and network emulator Mininet. The evaluation results demonstrate that our dynamic load balancing routing algorithm is superior over not only the none load balancing routing algorithm but also the static load balancing algorithm.

I. Introduction

We have witnessed the growing needs for the capability of the network data processing. For the newly emerging data center networks, maximizing system throughput and minimizing network latency are two important objectives [1]. In order to accomplish them, there exist both hardware and software approaches.

The hardware approach makes network infrastructures accommodate to novel network topologies. Recently, several topologies have been proposed, including the fat-tree [2] and Portland [8]. They build up networks in certain manners with hierarchical and scalable multi-layer structures. These types of structures usually contain dense interconnections among different layers of switches or routers. The data flows transmitting between hosts can go through multiple paths. These multiple paths provide more available bandwidth than a single path. Further, the whole infrastructure becomes more fault tolerant. Although this approach can achieve high throughput and low latency in networks, it may not be viable in some cases because the cost of modifying network topologies can be expensive.

On the other hand, the software approach applies load balancing methods for higher bandwidth utilization in existing networks. It requires improving current approaches of network traffic flow scheduling. Load balancing methods can be divided into two categories: static load balancing and dynamic load balancing [11]. By applying static load balancing, flows among hosts are allocated with calculated routes before data transmission. The routes cannot be changed during the transmission. As we will show in the evaluation section, however, static load balancing has limits to schedule large numbers of randomly generated flows in data center networks. By comparison, dynamic load balancing routing can schedule network traffic according to updated traffic statistics on each network device. Existing load balancing routing techniques include equal-cost multipath routing (ECMP) [4] and valiant load balancing (VLB) [5]. In ECMP, next-hop packet forwarding to a destination can occur over multiple paths with the same minimum performance metric. VLB, also known as randomized load balancing, is a scheme of routing through a randomly picked intermediate node. It helps eliminate congestion in the network. ECMP is used mainly on the flow level while VLB can be applied on both the packet and flow levels.

Although dynamic load balancing is flexible and adaptive to real-time network statistics, it brings extra overheads for monitoring network statistics and scheduling flows. Moreover, common commodity network devices such as switches and routers maintain their own forwarding or routing tables. In order to dynamically schedule flows, one way is to deploy a centralized controller to gather information from each network device. However, the communication among these devices requires time and bandwidth with the increasing scale of networks [8].

OpenFlow [7] is an innovative networking technology that offers high interoperability and economical ways of user control in data center networks. It has been used for load balancing in various systems [13]–[15]. A typical OpenFlow network consists of three components: the OpenFlow controller, OpenFlow switches, and hosts. Each of the switches maintains a flow table that contains forwarding information. The controller and switches communicate via OpenFlow messages. There are a series of actions that the OpenFlow controller can perform by sending messages to switches, such as updating flow tables or probing switch statistics. By analysing replying messages from switches, the OpenFlow controller can schedule data flows efficiently.

As a popular data center network topology, the fat-tree topology [2] contains multiple paths among hosts so it can provide higher available bandwidth than a single-path tree with

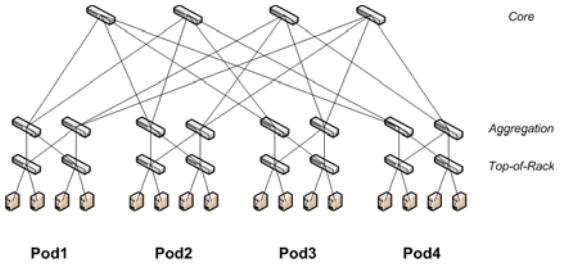

Fig. 1. A k=4 3-layer fat-tree network

the same number of nodes. It is typically a 3-layer hierarchical tree that consists of switches on the core, aggregation and top-of-rack(ToR) layers. The hosts connect to the switches on the ToR layer. The multipath feature of fat-tree networks enables chances to distribute data traffic on different network components. It is a practical task to achieve load balancing to help schedule traffic in fat-tree networks.

In this paper, we present a load balancer for the fat-tree network with multipath support. We implement a dynamic load balancing routing algorithm in the load balancer. This algorithm is adaptive to network traffic, and schedules flows by examining current available bandwidth on all alternative links. The load balancer works as a module of the OpenFlow controlling program called Beacon [9], which runs on the controller host. The dynamic flow scheduling algorithm in the load balancer module determines the path for each flow.

In the evaluation, we test our load balancer using the Mininet network emulator [10]. We compare the results of three flow scheduling algorithms: none load balancing (NLB) algorithm, static load balancing (SLB) algorithm and our dynamic load balancing (DLB) algorithm. The results show that our DLB algorithm works effectively for fat-tree networks. It provides network flows more bandwidth and shorter packet delay on average than the other two algorithms under same circumstances.

The remainder of the paper is organized as follows. Section II presents our network architecture, DLB algorithm, and its implementation. Section III explains the evaluation setup and provides the results. Finally, Section IV concludes the paper.

II. Algorithm Design

In this section, we propose our DLB algorithm. We also introduce the design of our OpenFlow based load balancer with two of its main functionalities.

A. Architecture

Figure 1 shows a 4-ary fat-tree. A typical k-ary fat-tree network [1] has three layers and consists of $(k/2)^2$ core layer switches and k pods that contain same number of aggregation and Top-of-Rack(ToR) layer switches. Each pod has k number of k-port switches. The switches in each pod have two types: top-of-rack (ToR) switches on the bottom, aggregation switches on the medium. In one pod, each ToR switch is connected to every aggregation switch and $(k/2)$ hosts. Each aggregation switch connects to $(k/2)^2$ switches on the core layer. In the fat-tree, there are more interconnections among different layers than many traditional tree networks. Those redundant links provide alternative paths for data transmission.

In our design, the fat-tree network has one OpenFlow controller host. The controller program running on the host includes the load balancer as one of its modules. The load balancer is responsible for scheduling flows over the OpenFlow switches by applying its load balancing routing algorithm.

B. Algorithm Description

In general, our DLB algorithm performs on flow level and follows a depth-first scheme. All traffic in the network from their source hosts will transmit upward until reaching highest layers that they need access, then go downward to destination hosts. For a new network flow, DLB firstly determines its source and destination hosts, then decides which layer the flow needs to access. Specifically, the flow between hosts connected to the same ToR switch needs access the ToR layer; the flow between hosts located in the same pod but connected to different ToR switches needs access the aggregation layer; the flow between hosts in different pods needs access the core layer.

One characteristic of routing in fat-tree network is: once a flow reaches the highest layer that it accesses, the path from that switch to the destination host is deterministic. In another word, only the upward traffic will be handled by our DLB algorithm. The downward traffic will be automatically determined corresponding to highest layer switch and destination.

Algorithm 1 DLB routing algorithm

Require: flow, fatTree
1: srcHost = locateSrc(flow);
2: dstHost = locateDst(flow);
3: layer = setTopLayer();
4: curSw = locateCurrentSwitch();
5: direction = 1; //search upward
6: path = null; //list of switches
7: **return** search();

The DLB algorithm schedules flows based on current network statistics. Algorithm 1 shows the pseudo code of our DLB algorithm. This algorithm works as follows. When the OpenFlow controller receives a packet from a switch, it switches the control to the load balancer. Line 1 to 6 introduces the initialization for necessary variables. The load balancer firstly analyses the packet's match information including the input port on the switch that receives the packet as well as the packet's source address and destination address. Then it looks up those addresses using its knowledge about the network topology. Once the source and destination hosts are located, the load balancer calculates the top layer that the flow needs to access. We use the search direction flag. The flag has two values: 1 for upward and 0 for downward. It is initialized to 1. A path is created for saving a route grouped by a list of switches later. Line 7 calls search() that performs the search for paths recursively.

Algorithm 2 describes the method search(). It firstly adds current switch into path. It returns the path if current search reaches the bottom layer. It reverses the search direction if current search reaches the top layer. Then it calls a method

Algorithm 2 Recursive search for paths

```
1: search() {
2: path.add(curSw);
3: if isBottomLayer(curSw) then
4:    return path;
5: end if
6: if curSw.getLayer()==layer then
7:    direction = 0; //reverse
8: end if
9: links = findLinks(curSw, direction);
10: link = findWorstFitLink(links);
11: curSw = findNextSwitch(link);
12: return search();
13: }
```

that returns all links on current switch that are towards current search direction. Only one link is chosen by picking up the worst-fit link with maximum available bandwidth. And then the current switch object is updated. The method search() is called recursively layer by layer from the source to destination. At last the path will be return to the load balancer. The path information will be used for updating flow tables of those switches in the path.

C. OpenFlow based Load Balancer Implementation

As a module of the OpenFlow controller program, the load balancer has two important functionalities: monitoring ports statistics and scheduling new flows. It maintains updated port statistics and schedules flows using our DLB algorithm. Next we illustrate them one by one.

1) Ports Statistics Monitoring: As shown above, the worst-fit link selection is based on finding the link with maximum available bandwidth. But the important question is how to measure the available bandwidth on each link. It is difficult to measure them in direct ways from the controller or switches.

In our design, the controller queries the transmitted bytes on each port from switches periodically, calculates their increments and uses the increments as the selecting criteria. It works as follows. We initially define a monitoring cycle as time T. At the beginning of each monitor cycle T, the controller will send out a STATS_REQUEST message assigned "PORT" as its statistic type to every switch in the network. This message requires replies from the switches. Meanwhile, the controller maintains a set of ports S and the number of bytes they have transmitted in the last monitor cycle as Tx. Each tuple of S consists of three elements as $\{switchID, portnumber, Tx\}$. Once a switch replies to the controller with a STATS_REPLY message, the controller will update Tx for all of its ports in S according to corresponding $\{switchID, portnumber\}$. The load balancer can find the worst-fit link by comparing Tx of all links.

The overhead of monitoring port statistics is minimal. The sizes of messages for requesting and replying port statistics on each port are 8 bytes and 104 bytes respectively [12].

2) Flow Scheduling: The Flow scheduling functionality works as follows. Each OpenFlow switch maintains its own flow table. Whenever any packet comes in, the switch checks the packet's match information with the entries in its flow table. The packet's match information includes $ingressPort$, $etherType$, $srcMac$, $dstMac$, $vlanID$, $srcIP$, $dstIP$, $IPprotocol$, $TCP/UDPsrcPort$, $TCP/UDPdstPort$ [12]. If it finds a match, it will send out the packet to the corresponding port. Otherwise it will encapsulate the packet in a PACKET_IN message and send the message to the controller. As a module of the OpenFlow controller, the load balancer will handle the PACKET_IN message. It finds a proper path by executing a search with the DLB algorithm described in Algorithm 1. The path is a list of switches from source to destination of the packet. Then the load balancer creates one FLOW_MOD message for each switch in the path and send it to the switch. This message will have the packet's match information as well as a output port number on that switch. The output port number is directly calculated by the path and network topology. If one switch receives a FLOW_MOD message, it will use it to update its flow table accordingly. Those packets buffered on ports of that switch may find their matches in the updated flow table and be sent out. Otherwise the switch will repeat this process.

III. Evaluation

In this section, we describe the evaluation environment, traffic design and measurement approaches. We conduct tests with Mininet on Linux host with the Beacon OpenFlow controller. Then we measure the network throughput and latency under different types of traffic patterns.

A. Environment

1) Network Emulator: We use Mininet [10] to model data center network behaviour. To evaluate our DLB algorithm's scalability, we build up a k=8 fat-tree network with 80 switches and 128 hosts.

2) Monitoring Cycle Length: In Ports Statistics Monitoring of Section 3, we defined the monitoring cycle length as T. In our implementation, we set T as 5 seconds.

B. Traffic Design and Measurement

1) Traffic Design: One main challenge of the network performance evaluation is how to generate real network traffic since most current data centers are confidential for commercial uses. Research papers [3], [6] showed that by creating artificial traffic flows by programs, we can build up the testing environment similar to real ones. We use two types of traffic patterns. They are derived from the designs in [1]:

1) Random: A host sends packets to any other host in the network with uniform probability;
2) Probability Stride(i, j, k) (P_t, P_a, P_c): A host of index m sends packets to another host of index $(m+i), (m+j)$ or $(m+k)$ with probabilities P_t, P_a or P_c respectively, while $P_c = 1 - P_a - P_t$.

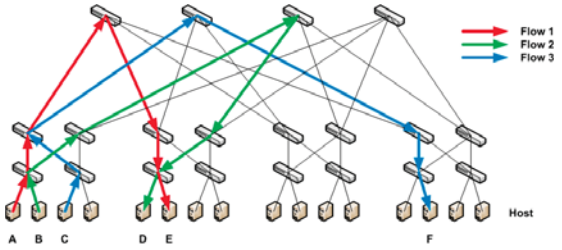

Fig. 2. Static load balancing(SLB) in a fat-tree network

2) Measurement: We use Iperf to generate UDP traffic in the network. The Iperf clients transmit data flows to Iperf servers. The traffic rate of each flow is in the range 8% to 12% of the overall link bandwidth. We simulate the traffic in real fat-tree data center networks according to the two traffic patterns above. Then we measure the results using two criteria:

1) Average ratio of flows' actual bandwidth / designed bandwidth: the real bandwidth that one flow can achieve will be less than or equal to the bandwidth it has been designed on Iperf client host. The higher this ratio is, the better throughput the network can get. This ratio can be obtained by Iperf server's report.
2) Average UDP packet transfer delay: the time consumed from packet sent out of the Iperf client to received by the Iperf server. It is measured by the server. The smaller the delay is, the less chances of network congestion exist. Iperf natively does not support measuring average delay and we installed a patch for measuring it.

The traffic load is defined as the ratio of average occupied bandwidth on links to hosts divided by the link capacity. It ranges from 0 to 1. We tested 9 groups with the load from 0.1 to 0.9.

C. Benchmark Algorithms

To test the performance of our dynamic load balancing (DLB) algorithm, we have used two other algorithms as comparisons. They are similar to DLB as described in Algorithm 1 except for strategies of selecting the link. Notice that both of them only affect link selections with packet going upward rather than downward just like DLB.

The first is the none load balancing(NLB) algorithm. NLB is not aware of traffic load information on links. Instead of choosing the worst-fit link, it randomly selects one link by calling a hash function.

The second is the static load balancing(SLB) algorithm. It selects links exclusively based on the location of the Iperf client. For each of the aggregation and ToR layer switch, we add indexes on its ports firstly from upside to downside, then from left to right on each side. Then a packet received by a downside port with index i will transmit via the output port with index $(i - k/2)$. One example is shown in Figure 2. In this tree, one ToR/aggregation switch has 4 ports with indexes from 1 to 4. And flow 1, 2 and 3 from source host A, B and C will take different routes to their destination hosts.

D. Results

We take the following parameters for the probability stride traffic test: Probability Stride(1, 4, 64) (50%, 30%, 20%).

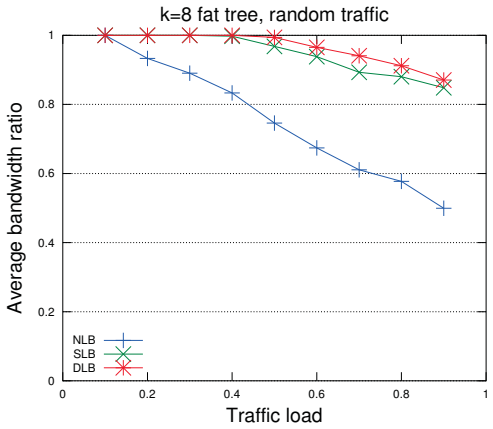

Fig. 3. Average bandwidth ratio in random traffic

Figure 3 shows the average ratio of flow's actual bandwidth versus designed bandwidth in random traffic. When the traffic load is 0.1, all three algorithms can schedule flows in a way that provides the maximum bandwidth they need. Then the ratios with all three algorithms decrease while the network traffic load increases. DLB keeps the highest average bandwidth ratio under each type of traffic load. The performance of SLB is between DLB and NLB. NLB has the most performance degradation with the increase of the traffic load. As we demonstrated above, NLB does not load balance the traffic. This can easily cause congestion when the load on one link exceeds its capacity. On the contrary, DLB and SLB take multi-paths and load balancing into consideration in scheduling and can achieve more available bandwidth than NLB under same level of traffic load. On the other hand, DLB outperforms SLB since it can make adaptive changes for scheduling paths based on real-time network traffic statistics. DLB makes more accurate decisions than SLB.

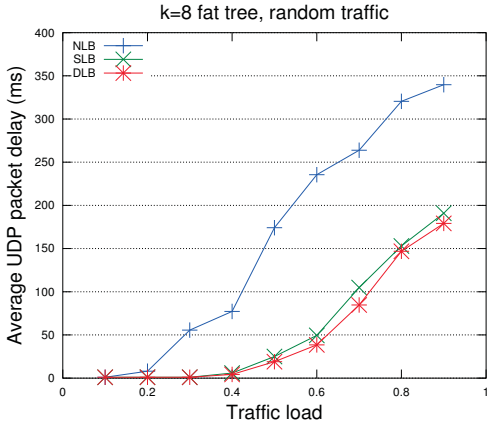

Fig. 4. Average packet delay in random traffic

Figure 4 shows the average UDP packet delay in random traffic. In general the delay moves higher with the traffic load increases. DLB and SLB have generally less than a half of the average UDP packet delay of NLB. DLB has a little advantage than SLB. When the load reaches 0.9, the average delays of all algorithms are over 170 milliseconds. This fact implies that when hosts in the network send flows with close to their

maximum capability, the traffic load becomes so high that the latency increases significantly. We also find that the increase of delay changes more rapidly than the decrease of bandwidth ratio.

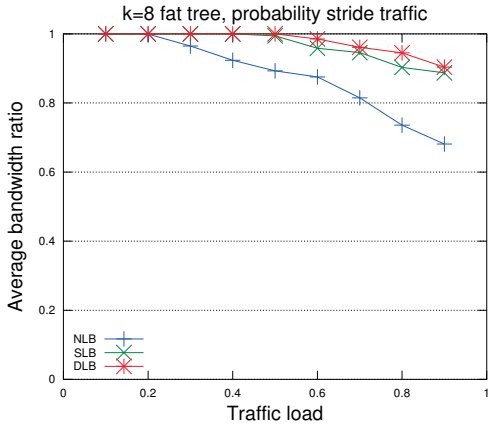

Fig. 5. Average bandwidth ratio in probability stride traffic

Figure 5 shows the average ratio of flow's actual bandwidth versus designed bandwidth in probability stride traffic. The results are similar to Figure 3 with random traffic. DLB can provide the largest amount of available bandwidth for flows on average under each level of traffic load. The performance of DLB and SLB under these two traffic patterns is close. On the other hand, although NLB still has the worst performance, the ratio curve of NLB shows improvement in probability stride traffic than in random traffic. This is because there are a lot of flows transmitted inside pods locally in probability stride traffic pattern as the probability of stride 1 and 4 traffic is 80% in total. The local traffic within one pod needs less number of links to transmit and has less chance of being scheduled on the same link. This reduces the chance of congestion.

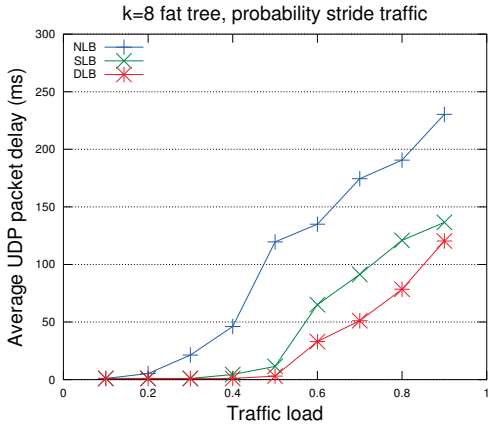

Fig. 6. Average packet delay in probability stride traffic

Figure 6 shows the average UDP packet delay in probability stride traffic. As we can see DLB maintains the average packet delay within milliseconds even under the load 0.5. The delays with all three algorithms increase more rapidly after the load exceeds 0.5. DLB leads this test again while NLB finishes the worst. Comparing to tests under the same load in random traffic, all three algorithms achieve lower delay in probability stride traffic. This improvement is also one consequence of the frequently generated local flows within pods by the probability stride traffic pattern.

IV. Conclusion

In this paper, we present a dynamic load balancing algorithm to efficiently schedule flows for fat-tree networks, which provide multiple alternative paths among a single pair of hosts. The algorithm utilizes the hierarchical feature of fat-tree networks to recursively search for a path, and makes decisions based on real-time traffic statistics obtained via the OpenFlow protocol. We have implemented the algorithm as a module of the Beacon OpenFlow controller program, with two main functions: monitoring traffic statistics and scheduling flows. In conjunction with the Beacon controller, we use the Mininet network emulator to evaluate the dynamic load balancing algorithm, by comparing it with the none load balancing and static load balancing algorithms. The results show that our algorithm is superior over the other two in maintaining high rate data transmission and avoiding network latency under various types of network traffic.

In our future work, we plan to extend the dynamic load balancing algorithm to traditional networks with only regular switches, or hybrid networks with both OpenFlow and regular switches.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. *A Scalable, Commodity Data Center Network Architecture*. ACM SIGCOMM, 2008.

[2] C. E. Leiserson. *Fat-trees: Universal networks for hardware-efficient supercomputing*. IEEE Transactions on Computers, 1985.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang. *Understanding Data-center Traffic Characteristics*. SIGCOMM WREN workshop, 2009.

[4] HOPPS, C. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992, IETF, 2000.

[5] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publisher, 2004.

[6] S. Kandula, S. Sengupta, A. Greenberg, P. Patel and R. Chaiken. *The Nature of Data Center Traffic: Measurements & Analysis*. ACM IMC 2009.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. *OpenFlow: Enabling Innovation in Campus Networks*. ACM SIGCOMM CCR, 2008.

[8] R. N. Mysore, A. Pamporis, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. *PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric*. ACM SIGCOMM, 2009.

[9] Beacon OpenFlow Controller. *https://OpenFlow.stanford.edu/display/Beacon/Home*.

[10] B. Lantz, B. Heller, and N. McKeown. *A Network in a Laptop: Rapid Prototyping for Software-Definded Networks*. ACM SIGCOMM, 2010.

[11] Y. Zhang, H. Kameda, S. L. Hung. *Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems*. Computers and Digital Techniques, IEE, 1997.

[12] OpenFlow Switch Specification, Version 1.0.0. *http://www.OpenFlow.org/documents/OpenFlow-spec-v1.0.0.pdf*.

[13] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. *Plug-n-Serve: Load-balancing web traffic using OpenFlow*. ACM SIGCOMM Demo, 2009.

[14] R. Wang, D. Butnariu, J. Rexford. *OpenFlow-Based Server Load Balancing Gone Wild*. Hot ICE, 2011.

[15] M. Koerner, O. Kao. *Multiple service load-balancing with OpenFlow*. IEEE HPSR, 2012.