VMS: Traffic Balancing Based on Virtual Switches in Datacenter Networks

Zhaogeng Li, Jun Bi, Yiran Zhang, Abdul Basit Dogar, Chengwei Qin
Institute for Network Sciences and Cyberspace, Tsinghua University
Department of Computer Science, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology (TNList)
li-zg07@mails.tsinghua.edu.cn, junbi@tsinghua.edu.cn, zyr17@mails.tsinghua.edu.cn,
bas15@mails.tsinghua.edu.cn, qcw14@mails.tsinghua.edu.cn,

***Abstract*—There have been many traffic balancing solutions for datacenter networks. All of them require modifications to the network fabric or/and virtual machines. In this paper, we propose Virtual Multi-channel Scatter (VMS), a new traffic balancing solution in datacenter networks. VMS works in the virtual switches between the network fabric and virtual machines. It can be deployed by datacenter operators at a relatively low cost without extra restrictions to virtual machine users. VMS scatters packets in one TCP flow to several different forwarding paths. It employs an adaptive path selection based on the virtual window size of different paths. We implemented VMS based on OVS. Our evaluation demonstrates that VMS improves traffic balancing very well, and the performance of VMS is approximate to MPTCP in almost all the cases, while only modifies virtual switches. Further, the overhead of VMS is tolerable.**

***Index Terms*—Datacenter Network, Traffic Balancing, ECMP, MPTCP, VMS**

I. Introduction

Modern datacenter networks are usually constructed in a multi-tier Clos way. Many available paths exist between any server pair, which provide sufficient server-to-server bandwidth for east-west traffic [1], and quite a good failure tolerance. ECMP, which uses the hash value of 5-tuple for next hop selection, is often used for distributing traffic among these available paths to reduce the probability of congestion [2], [3], [4]. However, the effectiveness of ECMP is limited because it is totally random and has no global information. To balance traffic evenly, which is known as traffic balancing, or traffic load balancing, is a challenging problem.

In the recent few years, many advanced traffic balancing solutions have been proposed. These solutions can be categorized into four types: centralized scheduling [5], [6], adaptive forwarding [7], [8], spraying [9], [10], [11], [12], [13] and transport protocol modifying [14], [15]. However, none of them is widely used. The main reason is that these solutions require modifications to the network fabric or/and end servers. For the datacenter operators, they have to suffer from much more cost and extra user restrictions to deploy a traffic balancing solution. Therefore, a new traffic balancing solution with fewer modifications is needed.

Virtualization has become dominant in modern datacenters. In the context of virtualization, resources are provisioned to the users in the form of virtual machines (VM) or even containers. To provide network service for these virtual machines and containers, virtual switches (e.g. OVS[16]) are widely used as a software in servers. Generally, virtual switches act as the edge switches from the view of VM users. It is the datacenter operators who take control of these virtual switches. And it is relatively easier to modify virtual switches.

In this paper, we propose a traffic balancing solution based on virtual switches, named Virtual Multi-channel Scatter (VMS). VMS requires no modification to the network fabric or VM users. It can be deployed when the datacenter operators do not attempt to change the network fabric or cannot control the transport protocol of virtual machines. VMS scatters packets in one TCP flow to several different forwarding paths (channels) by changing the 5-tuple on the send side. The receive side changes the 5-tuple back and re-sequence the packets to avoid out-of-order. For each forwarding path, VMS estimates the available bandwidth of it, represented by the virtual window size. With virtual window size, VMS enforces the total flow rate by setting the window size in the TCP header. And VMS adaptively selects the forwarding path for a packet based on the virtual window size of different channels. We implemented VMS based on OVS.

We evaluate VMS with simulations. The simulation results suggest that 8-channel VMS is better than VMS with fewer channels. And the performance of VMS is approximate to MPTCP (while only modifies virtual switches), and it is hardly affected by the topology asymmetry. Moreover, VMS guarantees robustness to packet drop and fairness very well. Besides, we also evaluate the overhead of VMS by the prototype implementation. The experiment results demonstrate VMS introduces tolerable overhead.

We have three main contributions in this paper. First, we summarize the previous traffic balancing solutions and point out their drawbacks. Second, we propose VMS to show that it is possible to achieve traffic balancing with no modification to the network fabric or virtual machines, unlike existing solutions. Third, we perform simulations and implement a prototype to show the good performance and the slight overhead of VMS.

The rest of this paper is organized as follows: In Section II, we introduce the background of traffic balancing and the

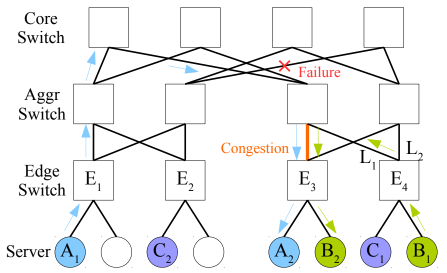


Fig. 1. Examples of the problem of ECMP and the path selection problem of spraying-based traffic balancing. This network is a 3-tier Clos network and all links have the same capacity.

problems of the previous solutions. In Section III, we describe VMS from the overview to details step by step. Section IV is some notes on the prototype implementation. We show the evaluation of VMS of in Section V. Section VI is the related works. We conclude this paper in Section VII.

II. Background

ECMP may result in congestion on the selected path while other paths are not fully utilized. For example, in Fig. 1, when A_1 sends data to A_2 and B_1 sends data to B_2 at the same time, there is a possibility that the two flows both get through the same link and encounter congestion (called *ECMP collision*). However, if the flow of user B chooses link L_2 instead of link L_1, the congestion disappears. That means the available bandwidth between two servers is uncertain. And the network loses the characteristic of non-blocking even though it has no oversubscription. To cope with this problem, there are four types of traffic balancing solutions so far.

Centralized Scheduling When the congestion is mainly caused by collisions among large flows, centralized flow scheduling is a good choice [5], [6]. A centralized controller gathers information of flows (especially the bandwidth requirements) and tries to calculate the optimal (or sub-optimal) flow routing. After that, switches get the routing rules from the controller to reroute the flows to avoid congestion. The critical drawback of this approach is that it can hardly handle highly dynamic traffic, since the flow bandwidth requirements may have changed a lot when the rerouting rules are distributed to the switches. □

Adaptive Forwarding Specialized switches which realize adaptive forwarding can also be used for traffic balancing [7], [8], [17]. Here "adaptive" means that the switches can detect the utilization of each path with a specialized detection protocol and always forward a flowlet to the path with the lowest utilization. This is a promising approach, especially when the emerging technique of programmable data plane like P4 [18] is mature enough. However, this kind of devices has not been proved viable in a production datacenter. And the flowlet splitting does not work for all traffic patterns [12]. □

Spraying Another type of traffic balancing is to select paths for different packets, or different packet sets, instead of flows [9], [10], [11], [12], [13]. This is called spraying-based traffic balancing. It can be proved that this approach can achieve ideally even traffic distribution over different paths for a symmetric topology [19]. Spraying-based approaches mainly have two problems. First, it introduces out-of-order and requires reordering in the receivers. That means all the servers have to make changes accordingly. What is worse, for UDP traffic, out-of-order cannot be handled by the servers.

The second problem is topology asymmetry, which is often caused by failures and has a significant impact on the performance [19]. The switches must be able to detect the failures (through a centralized controller or a distributed protocol) as soon as possible, in order to adjust the path selection sequence/weight like [20]. Unfortunately, the adjustment can never lead to strict balance. To explain the path selection problem of spraying-based approaches, we use Fig. 1 as an illustration. Assuming that a link is failed in the topology (denoted by the red cross), the path selection weights in switch E_4 must be adjusted. However, optimal path selection weights depend on the traffic on the present moment. Suppose B_1 sends data to B_2 and C_1 sends data to C_2 at the same rate. For destinations under E_2, the optimal path selection weights ratio of L_1 to L_2, which is denoted by $\omega_{E_2}^{E_4}$ ($\omega_{switch_{destination}}^{switch_{current}}$), is 2:1. And $\omega_{E_3}^{E_4} = 1:2$. If we let A_1 send data to A_2 at the same time and $\omega_{E_3}^{E_1} = 2:1$, it is obvious that $\omega_{E_2}^{E_4} = 2:1$ and $\omega_{E_3}^{E_4} = 1:2$ are not optimal any more. □

Transport Protocol Modifying The traffic balancing problem can also be solved by modifying the transport protocol, namely TCP, of users. MPTCP is a multipath TCP modification which uses different paths to send data at the same time [14]. For one TCP connection, MPTCP employs several subflows, each of which behaves like an independent TCP flow. These different subflows traverse different paths because they have different 5-tuples whereas the network uses ECMP. Since almost all traffic in a datacenter is TCP traffic, MPTCP solves the traffic balancing problem at a low cost. However, it still has many drawbacks, including high CPU overhead and potential failures [21]. In addition, the network operators cannot guarantee that all the users use MPTCP. Other TCP modifications (e.g. [15]) share the similar problem. □

Although all the above four types of traffic balancing have good performance in specific scenarios, we found none of them completely suits current datacenters. Centralized flow scheduling and adaptive forwarding both require changes to network devices, which will increase the cost significantly. MPTCP is not transparent to users, and it is unlikely to be deployed by users due to the problems mentioned above. Spraying is not transparent to users either (requires reordering), and it is not always efficient. To the best of our knowledge, none of these solutions are widely used in current datacenters. In order to meet the requirements of low cost, transparency to users and efficiency simultaneously, we design Virtual Multi-channel Scatter (VMS).

III. Virtual Multi-channel Scatter (VMS)

A. Overview

Modern datacenters often provision virtual machines or even containers. Usually, these units connect to the network

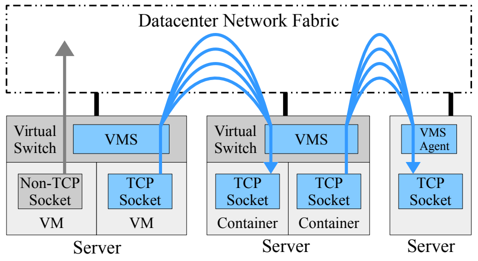


Fig. 2. Traffic balancing with VMS. Only TCP traffic is scattered by VMS.

via virtual switches. Virtual switches are controlled by the datacenter operators. Modifications to the virtual switches are transparent to VM users. Modifying them will not introduce too much cost since they are softwares[1]. Therefore, virtual switches can be used to deploy a traffic balancing solution.

VMS runs in the virtual switches (see in Fig. 2). Generally, it scatters TCP traffic into different paths on the send side to make traffic distribute more evenly. VMS works only when the servers on the both sides support VMS at the same time because the receive side must handle out-of-order and return feedbacks to the send side. The ultimate aim of VMS is to approximate the behavior of MPTCP in a switching-based way. The only premise of VMS is that the fabric enables ECMP (with 5-tuple hashing). Note that VMS will not scatter non-TCP traffic.

For a non-virtualized server, it can also deploy a VMS agent in its protocol stack to realize scattering. However, VMS is not mandatory. VMS traffic (using multiple paths) can work along with non-VMS traffic (using only one path) very well. This is different from spraying-based solutions mentioned in Section II. The network operator can decide which servers support scattering and which servers do not. A VMS-enable table can be configured centrally and distributed to all the servers. Only when the destination appears in this table, scattering is applied. For example, datacenter-internet traffic can pass virtual switches without scattering.

In the data plane, VMS handles packets in a way similar to VCC [22] and AC/DC TCP [23]. VMS maintains a set of state information for every active TCP flow (see in Table I). For each passing packet, VMS will change the header of this packet and the state information accordingly. In the following, we describe the details of VMS step by step. For convenience, we use sVMS to denote VMS on the send side and rVMS to denote the VMS on the receive side. And we use sEnd and rEnd to denote the VM/container on the send side and the receive side respectively. Note that the send side and the receive side are defined in the direction of data transmission (not the direction of ACK).

B. Scattering among Different Channels

Since the network employs ECMP, the forwarding path of a packet depends on the 5-tuple: (source address, destination address, source port number, destination port number, protocol number). To scatter TCP traffic into different paths, we need to use different 5-tuples. sVMS changes the source port number to realize the scattering. In VMS, we use the term *channel* to refer to the path corresponding to a specific source port number. VMS has a fixed number of channels. We use N to denote the number of channels. For each packet, it chooses one of the N channels as its forwarding path.

Source port number is crucial for the TCP endpoint lookup in rEnd. Therefore, rVMS has to change it back. In order to make the change reversible, sVMS cannot change source port number arbitrarily. Assume that the original source port number of a packet is p and the currently used channel ID in sVMS is c ($0 \le c \le 7$). Then the source port number p' will be changed to

$$p' = ((p - c) \ \& \ 7) + (p \ \& \ (\sim 7)) \tag{1}$$

Here $\&$ and $\sim$ are bitwise operations "*and*" and "*not*" respectively. sVMS must put c in the *Channel ID* field in the TCP header (show in Fig. 3) at the same time. Then in rVMS, the original source port number p can be calculated from p' and c by

$$p = ((p' + c) \ \& \ 7) + (p' \ \& \ (\sim 7)) \tag{2}$$

We use $N \le 8$ due to the limited space in the TCP header[2]. The performance of VMS increases with N (unless N is already too large) because more channels introduce more even traffic distribution statistically. This is similar to the impact of the subflow number in MPTCP [14]. In the rest of this paper, we use $N = 8$ if there is no specific indication.

TABLE I
SOME IMPORTANT FLOW STATES USED IN VMS.

State	Meaning
NextSeq	next sequence number
LastACK	last acknowledgement number
MileStone	sequence number used to identify RTT
WindowSize	total window size of a flow
WindowScale	window scale factor of a flow
Timestamp	timestamp used for identify expired entry
CurrentChannel	current channel used for send packet
FeedbackChannel	current channel used for send feedback
Flags	some flags, like CLS
WindowSize[i]	window size of channel i
ReceivedCount[i]	received data of channel i in current interval
RttSize[i]	feedbacked data of channel i in current RTT
RttCESize[i]	feedbacked (CE) data of channel i in current RTT
LocalSendSeq[i]	local send sequence number of channel i
LocalRecvSeq[i]	local receive sequence of channel i
LocalFBKSeq[i]	local feedback number size of channel i
Alpha[i]	α of channel i
SsThresh[i]	slow start threshold of channel i
Flags[i]	some flags of channel i, like RCE

[1]Virtual switches can also be offloaded to NICs. Currently, we do not consider this case in this paper.

[2]If we use $N > 8$, *Channel ID* should not be put in the reserved bits of the TCP header. Instead, it should be put in an extra option field. Since $N = 8$ can achieve very good performance, we do not consider this case.

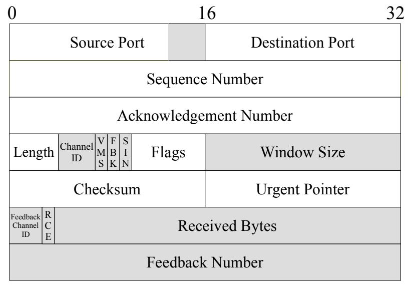


Fig. 3. TCP header format used in VMS. The gray fields are changed/inserted by VMS. Only the packet with the VMS Flag should be handled by VMS.

C. Re-sequencing in rVMS

Scattering traffic into different channels introduces out-of-order. rVMS solves this problem with a re-sequencing mechanism. Specifically, rVMS puts an incoming packet into the re-sequence buffer associated with the current flow. If the expected packet is already in the re-sequence buffer, it will be delivered to rEnd from the re-sequence buffer. Each re-sequence buffer records the time of recent delivering (e.g. by *jiffies*) to detect a timeout. If no packet is delivered to rEnd after timeout (detected by an independent thread), all the packets in the re-sequence buffer should be delivered to rEnd in sequence.

The most important parameter of the re-sequencing mechanism is the timeout T. If T is too large, the fast retransmission caused by packet loss would be delayed. In order to avoid the "silly wait" when there is a packet loss, T should not be larger than the maximum arrival time gap T' between any two consecutive packets P_1 and P_2 when there is no packet loss. The maximum gap happens when the packet P_1 traverses a path with all buffers full while packet P_2 traverses a path with all buffers empty. Assume that the capacity of the links in the fabric is C, the buffer size for one port in one switch is B, and there are at most k buffers in the fabric that a packet traverses. Then we have:

$$T \leq T' = kB/C \tag{3}$$

For example, if $C = 10Gbps$, $k = 4$ (for 3-tier Clos datacenter networks like Fig. 1) and $B =$1MB, $T \leq 3.2ms$. That means that a $4ms$ timeout (assuming that the resolution of time is $1ms$) is enough to eliminate out-of-order. Therefore, the maximum size of re-sequence buffer S can be figured out. Assume that the capacity of the access links is C'. Then we have:

$$S = C'T \tag{4}$$

For example, if $T = 4ms$ and $C' = 10Gbps$, $S \leq$ 5MB. That means a re-sequence buffer of 4MB can avoid buffer overflow. Note that 4MB is not for one flow, but for all flows in one virtual switch. Therefore, all the buffered packets can be kept in the CPU L3 cache (mostly larger than 10MB) before they are delivered to rEnd. This guarantees the performance of VMS.

$T = T'$ is for the worst case. In fact, VMS can use much smaller T. Empirically, $T = T'/2$ can make VMS work very well. For example, under the above conditions, the re-sequence buffer of 2.5MB with a timeout of $2ms$ is a good configuration.

D. Window-based Channel Choosing

For sVMS, the most important task is to choose a channel to send the current packet. Basically, sVMS can choose the current channel in a round-robin way. That means that the traffic is distributed to all the channels evenly. However, this will still result in traffic unbalance, because the forwarding path of each channel is random. For example, assuming there are two TCP flows f_1, f_2 and two paths p_1, p_2 with capacity c, when x channels of f_1 and y channels of f_2 are put on p_1 $(x + y > 8)$, there will be a congestion in p_1 while p_2 is not fully utilized. We use r_1 and r_2 to denote the flow rate of f_1 and f_2 respectively. Then we have $r_1 = r_2$ (fairness guaranteed by congestion control) and $xr_1/8 + yr_2/8 = c$. Therefore, $r_1 = r_2 = 8c/(x + y)$, which is lower than the expected rate c.

To cope with this problem, sVMS does not distribute traffic evenly to all the channels. Instead, sVMS chooses the current channel in a window-based way. In order to realize the window-based choosing, VMS estimates the virtual window size (stored as *WindowSize* in the flow states) of different channels. The virtual window size of one channel is similar to the window size of one subflow in MPTCP. It represents the available bandwidth of this channel. sVMS distributes traffic to all the channels according to the virtual window size of them.

We will describe the virtual window size estimation in Section III-F. Now we just suppose that the virtual window size is already known. Algorithm 1 shows the procedure of window-based channel choosing. *LocalSendSeq* and *LocalFBKSeq* are both in the fake local sequence space, which represent the next local sequence number which is going to send, and the maximum local sequence number which is already returned in the feedback respectively. More details about the fake local sequence number are in Section III-E. Here we only need to know *LocalSendSeq*−*LocalFBKSeq* equals to the total size of on-the-fly packets in one channel, denoted by $onfly$.

According to Algorithm 1, sVMS will send a packet to the channel which has an enough available window to contain this packet (line 8-9). If there is no channel that has an enough available window, the packet should be sent to the channel with the largest available window (line 11-13). Since sVMS can control the total size of packets sent by sEnd with receive window size adjustment (see in Section III-F), almost all the packets can find a channel which has an enough available window. With the help of the window-based channel choosing, the traffic will be distributed much more evenly.

E. Fake Local Sequence Number and Feedback

VMS maintains a fake local sequence space for each channel. We call it "fake" because the local sequence number is

Algorithm 1 Window-based Channel Choosing

```
1: procedure ChooseChannel(packet $p$, flow states $f$)
2:     $c \leftarrow f.\textit{CurrentChannel}$
3:     $max \leftarrow 0$
4:     for $i$ in $[1, 8]$ do
5:         $c \leftarrow (c+1)$ & 7
6:         $ch \leftarrow f.\textit{Channels}[c]$
7:         $onfly \leftarrow ch.\textit{LocalSendSeq} - ch.\textit{LocalFBKSeq}$
8:         if $onfly + p.size <= ch.\textit{WindowSize}$ then
9:             $c' \leftarrow c$
10:            break
11:        if $max < ch.\textit{WindowSize} - onfly$ then
12:            $max \leftarrow ch.\textit{WindowSize} - onfly$
13:            $c' \leftarrow c$
14:    $ch' \leftarrow f.\textit{Channels}[c']$
15:    $ch'.\textit{LocalSendSeq} \leftarrow ch'.\textit{LocalSendSeq} + p.size$
16:    $f.\textit{CurrentChannel} \leftarrow c'$
17:    return $c'$
```

not really used to identify the sequence of packets. sVMS and rVMS do not exchange the local sequence number explicitly. Instead, they update the local sequence number independently (both start from 0). For sVMS, the *LocalSendSeq* is increased once a packet is sent (line 15 in Algorithm 1). For rVMS, the *LocalRecvSeq* is increased after a packet is received accordingly.

VMS uses fake local sequence space rather than real local sequence space like MPTCP because sVMS has no send buffer in it. The sequence number without a buffer introduces one problem, that it is very difficult to be acknowledged. Besides, real local sequence space is meaningless because sVMS does not execute retransmission. And the retransmission started from sEnd will even not use the same channel. The only use of the fake sequence number here is to help sVMS to calculate the value of $onfly$ (see in Section III-D). Essentially, fake local sequence number is only a counter starting from the beginning of a flow.

rVMS returns feedbacks of its fake local sequence numbers (*LocalRecvSeq*) of each channel in a piggyback way. It inserts a special TCP option as shown in Fig. 3. And the flag *FBK* in the TCP header should be set. The feedback also includes *Received Count*, which is the total size received in this channel since last feedback (*ReceivedCount* in flow states). After received a feedback, sVMS updates the *LocalFBKSeq* according to the *Feedback Number* and increases the window size according to *Received Count*. Then sVMS removes this option before it delivers the packet to sEnd.

Since the fake local sequence number is updated by sVMS and rVMS independently, there will be an error if a data packet is dropped. To cope with this problem, we let VMS degrade to single-path transmission when it detects a data packet drop. More details about the degradation are in Section III-G. Although fake local sequence numbers are fragile, we argue that the fake local sequence number is necessary because it can eliminate the impact of ACK packet drop, which cannot be detected easily. If VMS only used *Received Count*, the estimated value of $onfly$ would be larger than the correct value if ACK packets with feedbacks are dropped. This will force some data migrating to other channels, which may incur severe congestion on those channels.

F. Window Control

sVMS must estimate the virtual window size of different channels as mentioned in Section III-D. VMS updates virtual window size in the way similar to [22] and [23]. Specifically, VMS takes advantages of explicit congestion notification. After a packet with CE codepoint in IP ToS field is received, rVMS will set the RCE flag of the corresponding channel in the flow states (see in Table. I). When an ACK is returned to sVMS, the RCE flag is also packed in the feedback option (see in Fig. 3). The use of *RCE* flag at sVMS is similar to ECE flag in DCTCP [24].

Algorithm 2 shows the procedure of virtual window size update in sVMS. First, the window size can be increased on each feedback if *RCE* flag is unset(line 9-14). The increasing is multiplicative when the window size is smaller than the slow start threshold (*SsThresh*), while it is additive on the contrary. *MSS* in line 12 refers to the maximum segment size of TCP. Note that there is a divisor of 2 in line 10 and line 12. It makes the window increasing keep pace with TCP implementation when delayed ACK threshold is 2 (default configuration on almost all operating systems). The divisor of 8 in line 12 is to guarantee fairness with the TCP flow not using multi-channel scatter.

The window size should be decreased every RTT. VMS uses the method based on *MileStone* to realize RTT estimation, which is similar to DCTCP implementation. If a new RTT starts, the *MileStone* should be set to a number larger than *NextSeq* (line 29). When the acknowledgement number (AN) in the received ACK exceeds *MileStone*, VMS considers it as the end of this RTT (line 18). At the end of one RTT, sVMS needs to calculate the fraction F of sent packets which have been tagged as CE in each channel in this RTT, by $F =$ *RttCESize* / *RttSize* (line 22). Here *RttCESize* is the total size of packets in feedbacks with *RCE* flag in this channel during this RTT (line 16). And *RttSize* is the total size of packets in feedbacks in this channel during this RTT (line 7). Then α for each channel (*Alpha*) is updated by: $\alpha \leftarrow \alpha \times (1-g) + F \times g$ (line 24). And the window size w is updated by $w \leftarrow w \times (1-\alpha/2)$ (line 25-26).

sVMS views the sum of the virtual window size of all channels as the total window size of this flow (line 27, 28). sVMS puts the total window size (right shift by *WindowScale* which is specified in the TCP *Window Scale* Option) in the *Window Size* field of the TCP header (line 30) and delivers it to sEnd. Then sEnd will limit the flow rate according to this value. This procedure is same with VCC and AC/DC TCP.

G. VMS Reset

VMS controls the flow rate on each channel based on the fake local sequence number. As mentioned in Section III-E,

Algorithm 2 *WindowSize* update in sVMS

```
1:  procedure OnFeedback(packet $p$, flow states $f$)
2:      $c \leftarrow p.\mathit{FeedbackChannelID}$
3:      $ch \leftarrow f.\mathit{Channels}[c]$
4:      $size \leftarrow p.\mathit{ReceivedCount}$
5:      if $size > 0$ then
6:          $ch.\mathit{LocalFBKSeq} \leftarrow p.\mathit{FeedbackNumber}$
7:          $ch.\mathit{RttSize} \leftarrow ch.\mathit{RttSize}+size$
8:          if $p.\mathit{RCE} = 0$ then
9:              if $ch.\mathit{WindowSize} < ch.\mathit{SsThresh}$ then
10:                 $adder \leftarrow size/2$
11:             else
12:                 $adder \leftarrow \mathit{MSS}\times size/2/8/ch.\mathit{WindowSize}$
13:             $ch.\mathit{WindowSize} \leftarrow ch.\mathit{WindowSize}+adder$
14:             $f.\mathit{WindowSize} \leftarrow f.\mathit{WindowSize}+adder$
15:         else
16:             $ch.\mathit{RttCESize} \leftarrow ch.\mathit{RttCESize}+size$
17:             $ch.\mathit{SsThresh} \leftarrow ch.\mathit{WindowSize}/2$
18:     if $f.\mathit{MileStone} \leq AN$ then
19:         $sum \leftarrow 0$
20:         for each channel $c$ do
21:             $ch \leftarrow r.\mathit{Channels}[c]$
22:             $F \leftarrow ch.\mathit{RttCESize}/ch.\mathit{RttSize}$
23:             reset $ch.\mathit{RttSize}$ and $ch.\mathit{RttCESize}$
24:             $ch.\mathit{Alpha} \leftarrow ch.\mathit{Alpha}\times(1-g) + F \times g$
25:             $m \leftarrow 1 - ch.\mathit{Alpha}/2$
26:             $ch.\mathit{WindowSize} \leftarrow ch.\mathit{WindowSize}\times m$
27:             $sum \leftarrow sum + ch.\mathit{WindowSize}$
28:         $f.\mathit{WindowSize} \leftarrow sum$
29:         $f.\mathit{MileStone} \leftarrow f.\mathit{NextSeq}+1$
30:     $p.\mathit{WindowSize} \leftarrow f.\mathit{WindowSize} \gg f.\mathit{WindowScale}$
```

fake local sequence number will have errors when there is a data packet drop. In order to avoid further loss, we let VMS degrade to single path when it detects a data packet drop. We call this degradation "VMS Reset". The data packet drop is detected by checking *LastACK* and *NextSeq*. If the acknowledgement number of an ACK equals to *LastACK* (Duplicated ACK) in rVMS, or the sequence number is smaller than *NextSeq* (timeout) in sVMS, VMS considers that a data packet drop has happened. Then the *SIN* flag in the TCP header is set to inform the peer side. After that, only channel 0 will be used. In order to maintain fairness, the divisor of 8 in line 12 of Algorithm 2 is removed in this case.

H. GSO and GRO

NIC offloading like GSO and GRO are widely used in datacenter networks. GSO segments large TCP data packets into small ones. And GRO assembles small TCP data packets to a large one. Note that GSO and GRO only happen in the driver of physical NIC. Virtual NICs used by virtual machines do not actually invoke GSO and GRO although the virtual machines enable them. That means: 1) On the send side, segmentation and checksum calculation are both executed after VMS handling finishes; 2) On the receive side, assembling and checksum verification are both executed before VMS handling starts. This feature reduces the overhead of VMS because the number of packets handled by VMS is reduced.

GSO introduces a problem for VMS. Recalling the window control in Section III-F, the total window size W is the sum of the window size of all channels: $w_0 + w_1 + \cdots + w_7 = W$. That means that sEnd can generate new data packet with size s according to W: $s \leq W$. It is possible that $s > w_i (i = 0, 1, \cdots, 7)$, which means that no channel has enough window size to contain this new data packet. Putting this packet in any channel will exceed the expected value. However, since we always choose the channel with the largest available window, the impact of this problem is limited (in Algorithm 1, $onfly$ can be larger than $ch.\mathit{WindowSize}$). Therefore, we do not cope with this problem in order not to increase extra overhead.

VMS has little impact on GRO. Since different channels have different 5-tuples, GRO will not be interrupted by out-of-order (unlike Presto [12]). Packets segmented from the same packet by GSO can be normally assembled by GRO. Although sometimes packets from different connections in the same sEnd may use the same 5-tuple because of source port number changing, they will not be incorrectly assembled by GRO because the TCP headers of these packets are not identical (at least, *Channel ID* is different).

I. Flow States Table

VMS maintains a set of state information for each flow, as shown in Table I. All the flow states are stored in the flow states table, which is organized as a hash table. The flow states of one flow consume 316 bytes. Assume that there are hundreds to thousands concurrent connections in one server [1]. Then the total size of flow states table is about 70KB-7MB (note that one TCP connection is composed of two flows). A modern CPU can easily keep the whole table in its L3 or even L2 cache.

VMS should remove the flow states after a connection is disconnected. A connection is disconnected after packets with FIN flag are received by both sides. Note that there is a LAST ACK packet to acknowledge the second FIN packet. VMS keeps a *CLS* flag in the flow states. If a packet with FIN flag is received, the *CLS* flag of the corresponding flow is set to 1. When an ACK packet is received, if the *CLS* flags in the corresponding flow and the reverse flow, both are 1, the flow states should be removed.

Sometimes, the connection does not disconnect normally. An entry in the flow states table should be removed if it has not been accessed for some time (e.g. 1 second), with the help of *Timestamp*. This mechanism may remove the states of some ON/OFF flows during the OFF period. Therefore, we allow a flow initializing its flow states by any non-FIN packet (not only by SYN packet). A flow whose states are removed by timeout can be regenerated in this way. However, since the fake local sequence number cannot be set properly, VMS does not scatter such flows (only use single-channel, equivalent to AC/DC TCP).

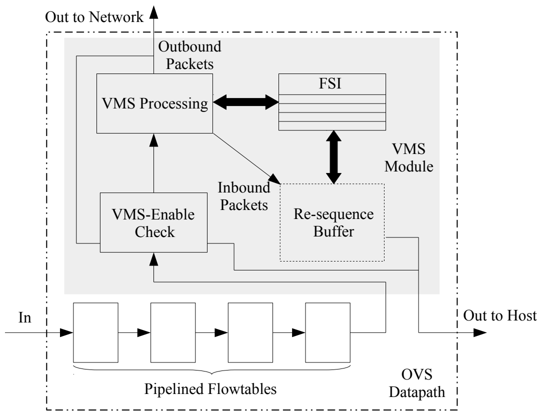


Fig. 4. Implementation of VMS.

IV. Notes on Implementation

We implement a prototype of VMS based on OVS 2.4.0. We mainly modify the datapath processing logic. Fig. 4 depicts the implementation. Before OVS executes the final forwarding action for a TCP packet, it sends the packet to *VMS Module*. *VMS module* first executes *VMS-Enable Check* to decide whether the packet should pass *VMS Processing* or not. For outbound packets, this check is realized by searching in *VMS-Enable Table* (also a flow table). For inbound packets, this check is realized by matching *VMS* flag in the TCP header. If the packet should pass *VMS Processing*, it will follow the rules described in Section III.

Note that in *VMS module* there is a Flow States Index (*FSI*), which is the hash table to store flow states (see in Section III-I). Each TCP connection is associated with two entries of flow states in FSI. The two entries are paired with each other to avoid two searches for one packet. For outbound packets, the packet can be forwarded after the VMS processing. For inbound packets, the packet should be sent to *Re-sequence Buffer* first. VMS uses a binary tree to organize the *Re-sequence Buffer* for each flow. Each node in the binary tree is a set of ordered packets (pointers to *skbuff*). Due to the reason of space, we do not elaborate the algorithms of inserting into and removing from the *Re-sequence Buffer*. Note that we do not show the timeout detection daemon in Fig. 4, which is an independent thread.

V. Evaluation

A. Comparing Different Channel Numbers

We simulate a simple topology shown in Fig. 5. There are two edge switches (ToR), both connect to 4 aggregate switches and m servers. All links in this topology have a capacity of 10Gbps. Server A_i sends data to server B_i. All the flows start at the same time. There are four available paths for each flow. We regard this scenario as a benchmark. The metric used here is the distribution of (converged) flow throughput (we repeat the simulations for hundreds of times.).

We evaluate the impact of different channel numbers. Fig. 6 shows the simulation results in boxplots when $m = 2, 4, 8, 16$.

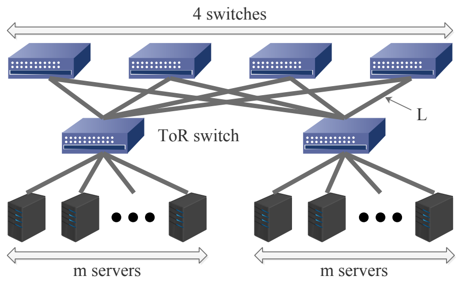
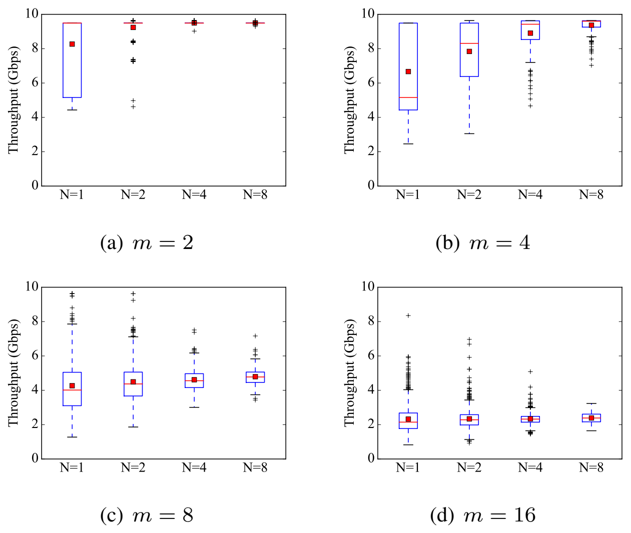


Fig. 5. Simple two-tier topology used in the benchmark simulation.



(a) $m = 2$ (b) $m = 4$ (c) $m = 8$ (d) $m = 16$

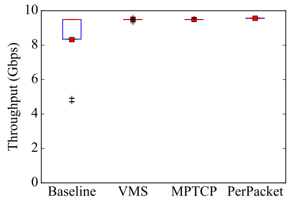
Fig. 6. Boxplots of the benchmark simulation results when $N = 1$ (no scatter), $N = 2$, $N = 4$ and $N = 8$.

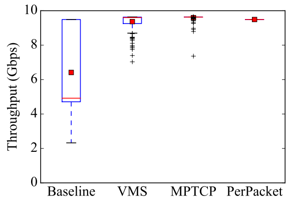
Note that $N = 1$ refers to single-channel transmission without scattering, which is equivalent to AC/DC TCP [23]. It is obvious that VMS with more channels results in higher and stabler throughput. When $m = 4$, 8-channel VMS has an average throughput about 9.5Gbps, while 2-channel VMS has only about 8.5Gbps. All the following simulations use $N = 8$.

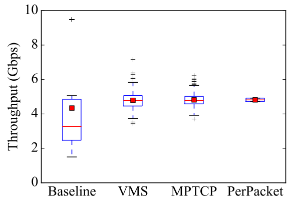
B. Comparing Different Traffic Balancing Solutions

We compare VMS with three other solutions. The first is Baseline, which means the servers use DCTCP and the network uses ECMP. The second is MPTCP, which means the servers use MPTCP (8 subflows with DCTCP congestion control) and the network uses ECMP. The third is PerPacket, which means the servers use DCTCP and have sufficient reorder buffers, and the network uses round-robin packet spraying. For DCTCP and VMS, $g = 1/16$ [24].

We also use the benchmark scenario like Section V-A in this simulation. Fig. 7 shows the simulation results in boxplots when $m = 2, 4, 8, 16$. The Baseline solution has unstable performance, which completely depends on the random hashing. And the mean value of throughput is the lowest. PerPacket is the best solution in this simulation because it realizes ideally even traffic distribution. The average throughput of PerPacket is the highest and the variation range is the smallest.


(a) $m = 2$


(b) $m = 4$


(c) $m = 8$

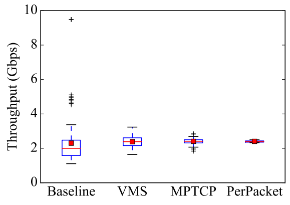
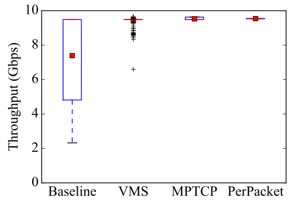
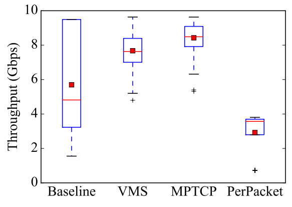
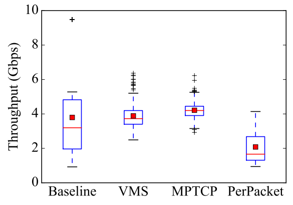

(d) $m = 16$

Fig. 7. Boxplots of the benchmark simulation results (symmetric topology) when different traffic balancing solutions are used.


(a) $m = 2$


(b) $m = 4$


(c) $m = 8$

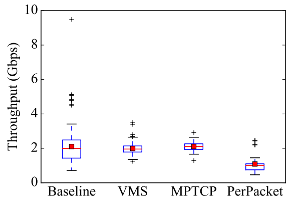

(d) $m = 16$

Fig. 8. Boxplots of the modified benchmark simulation results (asymmetric topology) when different traffic balancing solutions are used.

The performance of VMS and MPTCP is a little worse than PerPacket, but still very close.

Note that we also tried to compare VMS with Conga. However, we found it is very hard to configure the time gap of flowlets. In our simulation, 500μs (as [7] recommended) introduces very poor performance because no flowlet is split, while 50μs introduces good performance. However, it is not fair to say Conga is good or Conga is not good. Since the time gap is predefined in the switches, the actual results will be inconsistent for different traffic patterns at runtime. On the contrary, the above four solutions (Baseline, MPTCP, VMS and PerPacket) are all consistent for almost all traffic patterns. Therefore, we do not compare VMS with Conga.

Topology asymmetry is a common situation in real datacenter networks. We also simulate the scenario when the topology is asymmetric. We reduce the capacity of link L in Fig. 5 to 5Gbps. Due to the reason explained in Section II, we do not consider weight update for Baseline and PerPacket. Fig. 8 shows the simulation results. The performance of Baseline is still very poor. MPTCP keeps good performance. VMS is a little worse than MPTCP, but still much better than Baseline. The results also suggest that the performance of PerPacket is even worse than Baseline when the overall utilization is high. The reason is that the traffic distribution is even while the capacity of different paths is different so that the flow rate has to be limited according to the most congested path. What is worse, the congestion control of DCTCP cannot work correctly because the proportion of ECEs is inaccurate since they come from different paths.

C. The Impact of Packet Drop

We evaluate the impact of packet drop (VMS Reset). We use the benchmark scenario shown in Fig. 9 ($m = 4$). Here we let link L drop one packet at the time of 10ms. Fig. 9 shows the throughputs (calculated every 1ms on the receive side) over time of the 4 flows (the throughput of flows may exceed 10Gbps because of the existence of re-sequence buffers). The blue dashed line represents the flow whose packet is dropped.

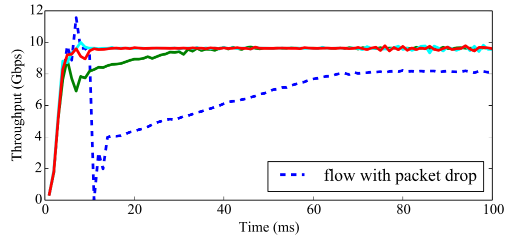
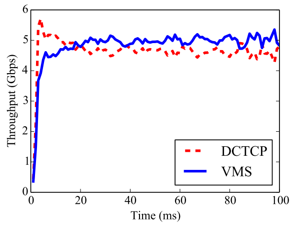


Fig. 9. Throughput of flows in the benchmark simulation when there is packet drop (4 flows).

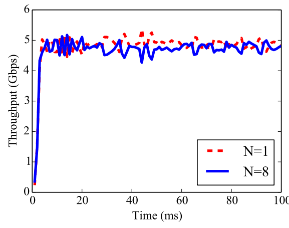
At first (0-3ms), all the channels of the flows are in the slow start stage. After that, the channels enter the congestion avoidance stage one by one and flows achieve stable throughput. At 11ms, the throughput of the blue-line flow decreases to 0 because of the packet drop. After VMS Reset caused by the drop, only one channel of it will be used. The throughput of this flow increase gradually (12-70ms) because of the window control described in Section III-F. Then the throughput is stable at about 8Gbps (71-100ms), smaller than other flows. The reason is that other flows share the bandwidth with this flow, however, they can use other paths in the meanwhile. The other three flows have almost the same throughput with each other, which proves the fairness of VMS.

D. Fairness

Fairness is an important feature of TCP-related techniques. There are four types of fairness concerned with VMS. The first is the fairness between non-VMS flows without ECN and VMS flows. The second is the fairness among VMS flows. The third is the fairness between non-VMS flows with ECN

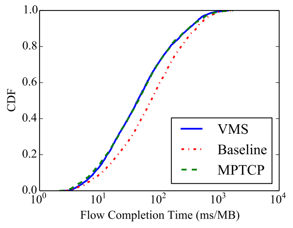


(a) DCTCP flow and VMS flow

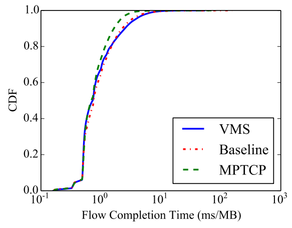


(b) 1-channel and 8-channel

Fig. 10. Throughput of flows over time when different types of flows share the same path at the same time.



(a) Large Flows



(b) Small Flows

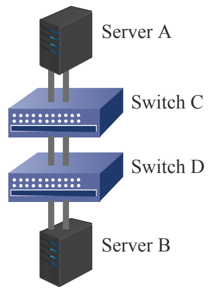
Fig. 11. CDF of flow completion time in the fat-tree simulation.

and VMS flows. The fourth is fairness between multi-channel-VMS flows and single-channel VMS flows (e.g. after VMS Reset). VMS cannot guarantee the first type of fairness because of RED queuing management [22], [23]. VMS can guarantee the second type of fairness easily due to the essence of AIMD (shown in Section V-C). We mainly focus on the third and the fourth type of fairness here. Note that we only focus on the fairness when they share the same path. If one flow can only use one path while the other can use multiple paths, we do not guarantee the strict fairness.

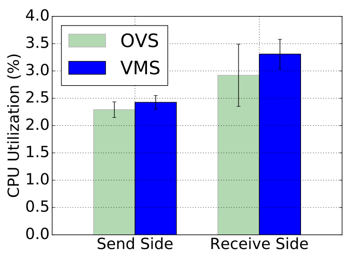
In the topology of Fig. 5, we let server A_1 and server A_2 transmit data to server A_3 at the same time. In the first simulation, we let A_1 use DCTCP and A_2 use VMS. In the second simulation, we let A_1 use 1-channel VMS and A_2 use 8-channel VMS. Fig. 10 shows the flow throughputs (calculated every 1ms) over time of the two simulations. In Fig. 10(a), the result demonstrates the fairness between the DCTCP flow and the VMS flow is good. The slight throughput gap comes from the difference between feedbacks in VMS and acknowledgements in DCTCP. In Fig. 10(b), the result demonstrates that 1-channel VMS and 8-channel VMS can achieve almost the same throughputs. In a word, VMS will not introduce a notable fairness problem.

E. Fat-Tree Topology Simulation

We also simulate an 8-ary fat tree topology [25] (with 128 servers) to evaluate the performance of VMS. In this scenario, all links have a capacity of 10Gbps. We generate all-to-all TCP traffic randomly. There are two types of TCP flows in this simulation (according to [26]). One is large flows with average size 50MB. The other is small flows with average size 20KB. The size of both types of flows is subject to a pareto distribution. And the interval time of two consequent flows is subject to an exponential distribution. The overall average link utilization is 50%.



(a) Testbed



(b) CPU Utilization

Fig. 12. (a) The testbed used in the overhead evaluation. (b) CPU utilization when standard OVS and VMS are used respectively.

Fig. 11(a) and Fig. 11(b) show the distribution of flow completion time of large flows and small flows separately. For large flows, the performances of VMS and MPTCP are very close to each other and both better than Baseline. For example, at 50th percentile, VMS can reduce the flow completion time from 73ms to 44ms. For small flows, VMS is close to Baseline and worse than MPTCP. The reason is that the benefit of better traffic balancing is counterbalanced by the feedback delay introduced by re-sequencing.

F. Overhead

To evaluate the overhead of VMS, we set up the testbed shown in Fig. 12(a). Switch C and Switch D (H3C S6300-F) connect to each other by two 10Gbps links, and both support ECMP and ECN. Either of the two servers connects to one of the switches through two 10Gbps NICs. Server A uses *iperf* to send data to Server B with two flows: f_1 and f_2. VMS on server A scatters the flows. Different channels in one flow will use different links between C and D. We take the CPU utilization in the two servers as the metrics. Each server has two Intel Xeon X5650 (2.67GHz, 6-core) CPUs.

Fig. 12(b) shows the CPU utilization (mean value and standard error) when standard OVS and VMS are used on the two servers respectively. Note that when standard OVS is used, the throughput of f_1 and f_2 may reach only 4.7Gbps because of *ECMP collision*. Therefore, we only consider the CPU utilization when there is no *ECMP collision*[3]. Since OVS datapath is multithreading, we consider overall CPU utilization. The result in Fig. 12(b) demonstrates that the overhead introduced by VMS is tolerable ($< 1.15\times$ OVS Overhead). We should mention that MPTCP (v0.91) introduces much higher CPU utilization. If we use MPTCP in this experiment, the flow throughput can only reach not more than 3Gbps because of the CPU bottleneck.

[3]Because the switch capacity of switches we used is only about 12Gbps, two flows without *ECMP collision* can only reach about 6Gbps.

VI. Related Works

There are a lot of literatures focusing on traffic balancing in datacenter networks. Hedera [5] and MicroTE [6] require a centralized controller and Openflow switches to reroute flows. Conga [7] and HULA [8] require specialized switches with functionalities of flowlet splitting and utilization detection. Some other literatures propose per-packet spraying to balance traffic, like DRB [10] and DRILL [13]. DeTail [11] is also a per-packet spraying on a lossless Ethernet (with PFC pause). Similarly, Presto [12] splits flows to segments of 64KB (flow-cell) and uses different paths to forward them. MPTCP [14] is a TCP modification which uses several subflows to send data in order to achieve better traffic balancing. FlowBender [15] is also a TCP modification used for the same purpose, which changes the forwarding path by changing TTL (or VLAN) when it detects congestion.

To reduce the amount of packet losses, servers in datacenters often use DCTCP [24] to reduce the congestion window. For the cases when the VMs do not enable these TCP modifications, a virtualized congestion control with similar behavior can be deployed in virtual switches like VCC [22] and AC/DC TCP [23]. These virtualized congestion control mechanisms can also guarantee fairness among different TCP implementations.

VII. Conclusion

In this paper, we propose a new traffic balancing solution, named Virtual Multi-channel Scatter (VMS). VMS runs in the virtual switches and requires no modification to the network fabric or VMs. It can (not must) scatter packets in one TCP flow to several different forwarding paths. It uses adaptive path selection based on the virtual window size of each channel to realize even traffic distribution.

We evaluate VMS with simulations. Both in the benchmark simulation and a relatively large-scale simulation, VMS can get very good performance, which is very close to MPTCP. In addition, through our prototype implementation based on OVS, we show the overhead of VMS is tolerable. Therefore, we believe that VMS is a good replacement of previous solutions. It can be deployed when the datacenter operators do not attempt to modify the network fabric and cannot control the transport protocol of virtual machines. We will try to offload VMS in future works.

Acknowledgment

This research is supported by National Key Research and Development Plan of China (2017YFB0801701) and the National Science Foundation of China (No.61472213), and sponsored by Huawei. We thank Keqiang He for sharing the code of AC/DC TCP. Jun Bi is the corresponding author.

References

[1] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. Snoeren, "Inside the Social Network's (Datacenter) Network," in *SIGCOMM*, 2015.

[2] "Introducing: Data Center Fabric, the Next-generation Facebook Data Center Network," https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[3] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, E. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hlzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *SIGCOMM*, 2015.

[4] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *SIGCOMM*, 2015.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *NSDI*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *CoNEXT*, 2011.

[7] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. Lam, F. Matus, R. Pang, N. Yadav, and G. Varghese, "CONGA: Distributed Congestion-aware Load Balancing for Datacenters," in *SIGCOMM*, 2014.

[8] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable Load Balancing Using Programmable Data Planes," in *SOSR*, 2016.

[9] A. Dixit, P. Prakash, and R. Kompella, "On the Efficacy of Fine-grained Traffic Splitting Protocolsin Data Center Networks," in *SIGCOMM*, 2011.

[10] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks," in *CoNEXT*, 2013.

[11] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *SIGCOMM*, 2012.

[12] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based Load Balancing for Fast Datacenter Networks," in *SIGCOMM*, 2015.

[13] S. Ghorbani, B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Micro Load Balancing in Data Centers with DRILL," in *HotNets*, 2015.

[14] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *SIGCOMM*, 2011.

[15] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks," in *CoNEXT*, 2014.

[16] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch." in *NSDI*, 2015, pp. 117–130.

[17] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching." in *NSDI*, 2017, pp. 407–420.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM CCR*, 2014.

[19] A. Dixit, P. Prakash, Y. Hu, and R. Kompella, "On the Impact of Packet Spraying in Data Center Networks," in *INFOCOM*, 2013.

[20] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers," in *EuroSys*, 2014.

[21] C. Raiciu, C. Paasch, S. Barreand, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *NSDI*, 2012.

[22] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized Congestion Control," in *SIGCOMM*, 2016.

[23] K. He, E. Rozner, K. Agarwal, Y. Gu, W. Felter, J. Carter, and A. Akella, "AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks," in *SIGCOMM*, 2016.

[24] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *SIGCOMM*, 2010.

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

[26] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *SIGCOMM*, 2009.