Multi-Channel Scatter (MCS): Traffic Balancing Based on Edge-switching in Datacenter Networks

Zhaogeng Li, Jun Bi, Yangyang Wang
Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
Department of Computer Science and Technology, Tsinghua University, Beijing, China
Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing, China
li-zg07@mails.tsinghua.edu.cn, junbi@tsinghua.edu.cn, wangyy@cernet.edu.cn

***Abstract*—There are many traffic balancing solutions for datacenter networks. All of them require network fabric or/and end user modifications. In this paper, we propose Multi-Channel Scatter (MCS), a new traffic balancing solution in datacenter networks. MCS works in the edge-switching layer (e.g. virtual switches in hypervisors) between the network fabric and end users. It can be deployed by the datacenter operators at a relatively low cost. MCS scatters packets in one TCP flow to several different forwarding paths (channels) to balance traffic. It can filter duplicated ACKs triggered by packet out-of-order, and uses congestion detection with ECN to update the weight of different paths and avoid packet loss. MCS processes packets in a switching-based mechanism and introduces acceptable overhead. Our simulation results demonstrate that its performance is much better than ECMP and close to MPTCP.**

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

I. Introduction

In modern datacenter networks, there are many equivalent paths between any server pair, which provide sufficient server-to-server bandwidth and quite a good failure tolerance. ECMP, which uses the hash value of TCP/UDP 5-tuple for next hop selection, is often used for distributing traffic among equivalent paths [1], [2], [3], [4]. However, the effectiveness of ECMP is limited because the path selection is totally random.

Recently, many advanced traffic balancing solutions have been proposed. Some of these solutions propose that the datacenter should modify the network by introducing centralized scheduling or using specialized switches to balance the traffic [5], [6], [7], [8]. Some other solutions propose that end users should employ a new transport protocol to balance the traffic in order to get better performance [4], [9]. There are also some solutions which introduce spraying-based balancing mechanisms [10], [11], [12], [13]. However, none of these solutions are widely used.

Problems of the above solutions are obvious. Centralized scheduling has a scalability problem. Using specialized switches introduces more cost and complexity. New transport protocol and reordering required by spraying-based mechanisms are only possible when the operating systems of end users can be controlled by the datacenter operator. Therefore, we need a new traffic balancing solution other than the above types.

ISBN 978-3-901882-94-4

In this paper, we propose a traffic balancing solution based on edge-switching, named Multi-Channel Scatter (MCS). MCS requires no modification to the network (as long as ECMP is used) or the operating systems of end users. MCS works in a shim layer between the network fabric and end users, which is often virtual switches in hypervisors. MCS scatters packets in one TCP flow to several different forwarding paths (channels) by changing the 5-tuple. MCS leverages an elaborate switching procedure to eliminate the impact of packet out-of-order, uneven traffic distribution, and failures. In order to improve the performance, MCS does not buffer any packet or use any fine-grained timer. According to our preliminary implementation based on OVS, the overhead of MCS is acceptable.

We have evaluated MCS with two types of simulations. First, we performed simulations of a scenario with a simple topology as a benchmark. The results show that 8-channel MCS has a much better performance than ECMP and is approximate to MPTCP, even in the case of asymmetric topology. It is also demonstrated that MCS can guarantee fairness among scattered flows, and the impact of failures is restricted. Second, we performed a simulation of a scenario with a relatively large-scale topology (8-ary fat tree [14]). The results also show that MCS can achieve a good performance very close to MPTCP. In a word, MCS is effective.

The main contribution of this paper is to propose a traffic balancing solution based on edge-switching without modifications to the network. Since MPTCP is approximate to the optimal traffic balancing, we do not argue that MCS is better than MPTCP. In fact, the aim of MCS is to get a performance close to MPTCP without assistance of end users. MCS and MPTCP have different application scenarios. MPTCP is used by end users, while MCS is deployed by the datacenter operator. MPTCP may not be used by end users because of its inherent drawbacks. However, MCS can still work in such a case.

The rest of this paper is organized as follows: In Section II, we introduce the related works. Section III describes MCS from a high-level aspect. We explain the design details of MCS in Section IV. Section V makes some discussions on the channel number, fairness, GRO friendliness, adaption to some other techniques and overhead. In Section VI, we show the evaluation of MCS. We conclude this paper in Section

VII.

II. Related Works

There are a lot of literatures focusing on traffic balancing in datacenter networks. Hedera [5] and MicroTE [6] require a centralized scheduler and openflow switches to reschedule flows. Conga [7] and HULA [8] require specialized switches with functionalities of flowlet splitting and utilization probing. The above two types of solutions have inherent drawbacks: The former one cannot balance highly dynamic traffic; The latter one cannot handle dense flow arrival very well. Besides, network modifications in these solutions are costly and have not been proved viable in a production environment.

There are also some proposals that only change end hosts with new TCP protocols. MPTCP [4] is a solution which uses several subflows to transmit data. It has some disadvantages, including high computation overhead and incompatibility to middleboxes [15]. FlowBender [9] uses ECN feedback (ECE flag in TCP header) to detect congestion and change forwarding path by using different TTL value (switches must support TTL-ECMP [16]). These solutions require datacenter users changing the transport protocols by themselves, which is not always possible in a cloud scenario.

Some other literatures propose per-packet spraying to balance traffic, like DRB [11] and DRILL [12]. DeTail [10] is also a per-packet spraying solution on a lossless ethernet (with PFC pause). Similarly, Presto [13] splits one flow to segments of 64KB (flowcell) and uses different paths to forward them. Since these spraying-based solutions introduce pervasive packet out-of-order, they require an extra reordering mechanism at the end hosts. Besides the overhead of reordering, these solutions have another drawback that they cannot fit asymmetric topology very well [17]. Therefore, spraying-based solutions are not good choices in a production environment.

To reduce the amount of packet losses, end users in datacenters often use DCTCP [18] to reduce the congestion window. For the cases when the users do not enable these TCP modifications, a virtualized congestion control with similar behavior can be deployed in the virtual switches [19], [20]. The virtualized congestion control limits the sending rate by reducing the receive window size, because the actual sending rate is determined by the minimum of the receive window size and the congestion window size.

III. MCS Overview

MCS works in a shim layer between the network fabric and end users, like [19] and [20]. This shim layer can be the virtual switch when VMs are provisioned, or the edge switch when physical hosts are provisioned. In this paper, we only consider MCS in a virtual switch. MCS aims to scatter the packets in one TCP flow into several different forwarding paths. Neither the network fabric nor the end user is aware of the scatter: The network fabric uses ECMP, and end users use original TCP. Note that MCS will not change UDP or other non-TCP packet processing. Fig. 1 depicts the deployment of MCS.

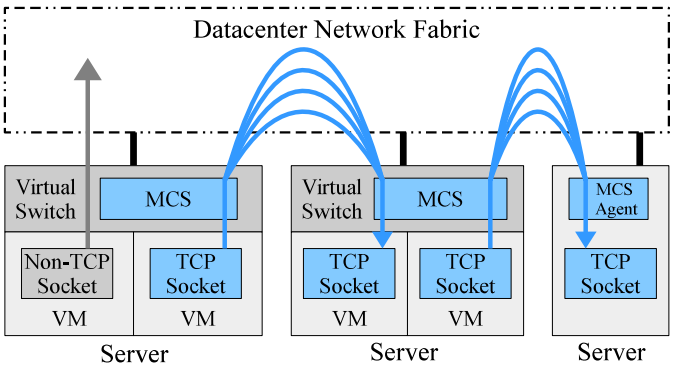


Fig. 1. MCS works in a shim layer between the network fabric and end users (e.g. virtual switches in hypervisor). MCS scatters the packets in one TCP flow into several different forwarding paths (channels) when the both sides support MCS and reside in different hosts.

For simplicity, we use the term *u-flow* to refer to the TCP flow from the view of end users, while using the term *n-flow* to refer to the TCP flow from the view of network fabric. In the case of ECMP, a *u-flow* is identical to the corresponding *n-flow*. ECMP cannot result in perfect traffic balancing because a *u-flow* can only get through one path (determined by *n-flow*) during its lifetime. However, the 5-tuple of *u-flow* and *n-flow* have different uses for the network fabric (as a forwarding identifier) and end users (endpoint identifier). Therefore, the identifiers of *u-flow* and *n-flow* should be decoupled.

MCS breaks this coupling by changing (destination/source) port number in TCP headers. MCS at the transmit side will set different port numbers for different data packets, and put the information for recovering the original port number in an extra TCP option named MCO (Multi-Channel Option, shown in Fig. 2). Different port numbers lead to different forwarding paths when ECMP is used. Inspired by wireless transmission, we use the term *channel* to refer to one port number and the corresponding forwarding path. MCS at the receive side will remove the MCO and recover it to the original packet. In this way, packets in different channels will traverse different paths, which is transparent to both the network fabric and end users.

MCS introduces pervasive packet out-of-order, which will harm TCP performance significantly if it is not well handled [21]. MCS does not employ reordering buffer like [11] and [12] does. The reason is that we want to avoid large buffers and fine-grained timers to reduce overhead and make hardware offloading simpler. Instead, MCS uses a switching-based processing[1] for each inbound and outbound packet. MCS can differentiate packet out-of-order from packet loss, and filter duplicated ACKs to prevent unexpected retransmission. In addition to out-of-order handling, MCS can detect congestion and failure to change traffic distribution across different channels. MCS also extends the window control mechanism introduced by [19] and [20] to avoid packet loss.

MCS maintains a capability table in it, which keeps the

[1]There are only three types of action for each TCP packet: change the packet header, forward it and drop it.

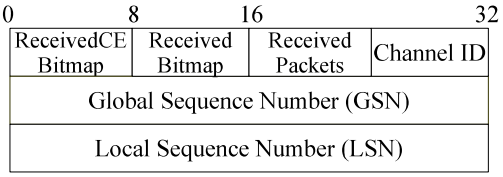


Fig. 2. Composition of MCS Option (MCO).

information whether a server supports MCS or not[2]. Only when the peer side is not in the same host and supports MCS, MCS processing will be applied. Across-internet TCP flows can also use MCS, as long as the gateway supports MCS processing. For a packet belonging to a TCP flow which is scattered by MCS, a flag in the reserved bits of the TCP header should be set to inform the peer side.

MCS and MPTCP have some similarities. However, they are totally different solutions. The main difference is that MPTCP is a TCP implementation which can be used by end users, while MCS is a switching functionality at the network edge which should be deployed by the datacenter operator. MCS is not an MPTCP proxy [22] either. An MPTCP proxy should establish connections by itself and maintain its own buffers, which is not a good choice because of high overhead. Therefore, how to design MCS is an entirely new problem.

IV. MCS Details

A. MCO and Channels

In MCS, packets should carry some extra information, including local sequence numbers and feedbacks. This information is exchanged as a TCP option (MCO), which is inserted by MCS on the transmit side and removed by MCS on the receive side. As shown in Fig. 2, it is composed of six fields (12 bytes in total): First three fields are feedbacks (see in Section IV-D, IV-E and IV-F). The fourth field is the channel ID. The fifth field is the global sequence number (*GSN*). The sixth field is the local sequence number (*LSN*). Note that MCO is invisible to end users, so that it does not need *Kind* and *Length* fields like other TCP options. Because the segmentation offloading (e.g. TSO) happens in physical NICs (not in the virtual NICs of VMs), MCS processes TCP data packets before segmentation. That means inserting MCO will not introduce a noteworthy MTU problem.

For one TCP flow, each channel has its own local sequence space. The sequence number field in the TCP header should be changed to the local sequence number in the current channel. Although the local sequence number is already in the required field of a TCP header (denoted by *QSN*), *LSN* in MCO is still indispensable. The reason is the existence of segmentation offloading. The offloading segments a large TCP data packet to small ones which comply with MTU, while it just copies all TCP options to each segment packet. Putting *LSN* in MCO makes it simpler to compute the actual sequence number of a packet (denoted by *ASN*): $ASN = GSN + (QSN - LSN)$.

[2]Since MCS works only when the both sides support it at the same time, it can also be implemented as an agent in the protocol stack if the server is not virtualized (see Fig. 1).

```
Flow States (68 Bytes):
    32-bit    NextLocSeqNo[8]      (NLSN)
    32-bit    NextSeqNo            (NSN)
    32-bit    LastACKNo            (LAN)
    32-bit    RetrSeqNo            (RSN)
    32-bit    MilestoneSeqNo       (MSN)
    16-bit    WindowSize           (WS)
    16-bit    FreshnessVector      (FV)
    16-bit    SkipTokenVector      (SKV)
    8-bit     WeightVector         (WV)
    8-bit     CongestedBitmap      (CB)
    8-bit     ActiveBitmap         (AB)
    8-bit     InitializedBitmap    (IB)
    8-bit     ReceivedBitmap       (RB)
    8-bit     ReceivedCEBitmap     (RCB)
    4-bit     WindowScale          (WSC)
    4-bit     CurrentChannel       (CC)
    8-bit     Flags                (FL)
    8-bit     Alpha                (AL)
    8-bit     ReceivedPackets      (RP)
    32-bit    Timestamp            (TS)
```

Fig. 3. Flow states in 8-channel MCS. The notations in brackets are abbreviations of the fields. Many fields are bitmaps or vectors, meaning that each of them contains the corresponding value for all channels. The *Flags* field includes *SlowStart*, *Close*, *Loss*, and *Initialized*. The usage of most fields is explained in Section IV.

MCS on the transmit side selects the channel for each TCP data packet in a weighted-round-robin mechanism (more details are in Section IV-D). In the rest of this section, we assume that 8 channels are used (from channel 0 to channel 7). Let *NPN* denote the (source/destination) port number used for the *n-flow* and *UPN* denote the (source/destination) port number used for the *u-flow*. If MCS on the transmit side decides to use channel i to send a data packet, it should change the port number in the TCP header to *NPN* = *UPN*$-i$. MCS on the receive side should change the port number in the reverse way: *UPN* = *NPN*$+i$.

B. Flow States

MCS maintains a set of states for each flow as shown in Fig. 3 (8 channels). The usage of most fields will be explained in the following. For one flow, the states require 68 bytes for 8-channel MCS. All the flow states are organized in a hash table. Note that 2 flows are associated with one TCP connection (paired in the table). According to [23], one server has 100s to 1000s concurrent connections. Therefore, MCS needs O(100KB) to store all the flow states, which is not a problem for current CPU cache.

C. Out-of-order Handling

The flow states stored in MCS include the next local sequence number (*NLSN*) for each channel. If a received packet carries an expected local sequence number in the corresponding channel, it will not be viewed as a packet loss even if the global sequence number is not in sequence. Note

that the local sequence number will never be acknowledged. If a packet with a specific local sequence number is dropped, the retransmission of this packet will carry a new local sequence number, and may use another channel.

Algorithm 1 shows the simplified procedure of out-of-order handling. Since Last ACK Number (*LAN*) is also kept in the flow states, MCS can identify duplicated ACKs. If there is no packet loss, all duplicated ACKs will be dropped directly by MCS (line 13). If the ACK is piggybacked by a data packet, MCS will unset the ACK flag instead of dropping the packet. If there is a packet loss (when a local sequence gap is found in any of the channels), MCS on the receive side will set the *Loss* flag in the flow states (line 6). The duplicated ACKs will pass MCS when the *Loss* flag of the reverse flow is set. In order to unset the *Loss* flag after the retransmisson is completed, the flow states also include a retransmission sequence number (*RSN*). When a packet loss is found, *RSN* is set to the global sequence number of the current packet (line 7). After the reverse ACK number is larger than *RSN*, the *Loss* state will be unset (line 17).

Algorithm 1 Out-of-order Handling

```
1:  procedure OnReceiveData(Packet p, Channel c)
2:      qsn ← Sequence Number in p
3:      lsn ← Local Sequence Number in MCO
4:      gsn ← Global Sequence Number in MCO
5:      if NLSN_c < sn then
6:          Flags.Loss ← true
7:          RSN ← gsn + (qsn − lsn)
8:      NLSN_c ← qsn + p.size

9:  procedure OnSendACK(Packet p, Channel c)
10:     an ← Acknowledgement Number in p
11:     if LAN = an then                          ▷ Duplicated ACK
12:         if Flags.Loss = false then
13:             unset ACK flag in p or drop p
14:     else if LAN > an then
15:         LAN ← an
16:     if LAN > RSN then
17:         Flags.Loss ← false
```

The above mechanism of duplicated ACK filtering may incur a severe problem when a non-duplicated ACK packet is dropped in the fabric. As this ACK never reaches the transmit side, there might be a retransmission. However, since MCS on the receive side has already changed *LAN* kept in the flow states, it will always regard the ACK to the retransmission as a duplicated one. This may result in endless retransmission. To cope with this problem, MCS will change *LAN* to the global sequence number of the data packet, when *LAN* is larger than that (not shown in Algorithm 1).

There is a special case of packet out-of-order. A packet with FIN flag may arrive at the receive side before the ACK in 3-way handshake (mainly for small flows). As described in RFC 793, the TCP receive side will return an RST to close the connection if it receives a FIN when it is in the state of SYN-RECV. In order to avoid this problem, after MCS on the receive side receives a FIN packet, it will generate a TCP packet without any flag (TCP null packet) first and send it to the receiver to change the TCP state from SYN-RECV to EST before the arrival of FIN.

D. Channel Selection

Scattering packets to different channels in round-robin mechanism does not result in perfect traffic balancing, because the forwarding path of one channel is random. To achieve better traffic balancing, MCS detects congestion of different channels and tries to transmit less data on congested channels. MCS on the receive side uses CE tag of ECN in IP header to detect congestion. If a channel is congested, MCS on the receive side will keep a record in the flow states (*RCB*) and give a feedback to the transmit side along with the next packet (as *ReceivedCE Bitmap* in MCO) going back.

When MCS receives a congestion feedback, it will keep a record in the flow states (*CB*). MCS will update the selection weight of different channels (*WV*) according to *CB* every RTT. MCS uses the method shown in Algorithm 2 to realize RTT estimation (widely used in TCP implementations). If a channel is congested in the recent RTT, the selection weight of this channel is increased by 1 unless it equals to 3 (line 6-7). In order to maintain the stability, at most one channel is allowed changing its weight (line 8). If the weight of every active channel is larger than 0, we decrease each of them by 1 as a normalization (line 10).

Algorithm 2 Weight Update

```
1:  procedure OnReceiveACK(Packet p)
2:      an ← Acknowledgement Number in p
3:      if MSN ≤ an then
4:          for each channel c do  ▷ Update selection weight
5:              if c is congested in the recent RTT then
6:                  if WV_c < 3 then
7:                      WV_c ← WV_c + 1
8:                      break
9:          if WV_i > 0 for any i ∈ [0, 7] then
10:             WV ← WV−0x55
11:         MSN ← NSN + 1
```

There are four possible values for selection weight: 0,1,2,3. The weight value w means that for each channel, the probability to select this channel is 2^{-w}. It equals to weighted-round-robin which uses 2^{-w} as the weight. MCS uses Skip Token (*SKV*) to realize the channel selection as shown in Algorithm 3. Skip token means how many times this channel should be skipped before it is selected. If one channel is skipped, MCS will check the next channel in the round-robin sequence.

E. Window Control

Weight adjustment can help MCS to move traffic from congested channels to uncongested channels. However, if all the channels are congested, traffic shifting does not work. In

Algorithm 3 Channel Selection

```
1: procedure ONSENDDATA(Packet p)
2:     for i in [1,9] do
3:         c ← (c + i) & 7        ▷ in the round-robin sequence
4:         if AB_c = true then    ▷ the channel must be active
5:             if SKV_c = 0 then
6:                 SKV_c ← (1 ≪ WV_c) − 1
7:                 break
8:             else
9:                 SKV_c ← SKV_c − 1
```

order to avoid packet loss, MCS introduces the window control mechanism in VCC [19] and AC/DC TCP [20]. MCS keeps a window size (*WS*) in the flow states and changes it in the way of DCTCP. Therefore, MCS can change the window size field in the TCP header of ACK packets (as the receive window) to regulate the flow rate, because the flow rate is determined by the minimum of the congestion window and the receive window.

For each packet received by MCS, *WS* kept in MCS can be increased. Algorithm 4 shows this procedure. Similar to ordinary TCP congestion control, the increase of *WS* has two types: slow-start increase (at the beginning) and congestion-avoid increase (after a congestion). Here MSS is the maximum segment size. *Received Packets* in MCO is the number of packets (without CE tag) acknowledged by this packet[3]. MCS on the receive side increases the value of *RP* (in flow states) when a packet without CE tag is received, and will put it in *Received Packets* in MCO when a packet is returned to the send side (set *RP* to 0 at the same time). The reason why MCS needs *Received Packets* is that the ACK sequence may be disrupted in MCS (mainly caused by disorders).

Algorithm 4 Window Update

```
1: procedure ONRECEIVE(Packet p)
2:     rp ← Received Packets in MCO
3:     if Flags.SlowStart then
4:         adder ← MSS×rp/2
5:     else
6:         adder ← MSS×MSS×rp/WS
7:     WS ← WS+adder

8: procedure ONNEWRTT              ▷ see in Algorithm 2
9:     if CB ≠ 0 then
10:        Alpha ← (1 − g)×Alpha+g
11:        WS ← WS×(1−Alpha/2)
12:    else
13:        Alpha ← (1 − g)×Alpha
```

Similar to DCTCP, *WS* must be reduced every RTT when there is congestion (line 11). Here g is preconfigured according to [18]. Note that MCS does not use the accurate ratio of ECE like DCTCP because the ratio is meaningless since more than one paths are used simultaneously. Instead, MCS assumes the ratio is always 1 when there is a congestion (line 10).

[3]Note that GSO will copy TCP options to each of the segments, which will make the sum of *Received Packets* in MCO much larger than the actual value. Therefore, MCS only considers the MCO with *LSN* equaling to *QSN* when it updates *WS*.

F. Failure Detection

Failure happens frequently in datacenter networks. Assuming a link or a switch fails, for original TCP, all connections using this failed link or failed switch cannot continue before routing update completes. MCS is a double-edged sword for failures. On one hand, for each ongoing flow, scattering traffic guarantees that the flow can continue with many timeouts before routing update completes, unless all channels get through the failure. On the other hand, the number of flows affected by one failure is amplified. Note that the impact of failures discussed here only exists before routing update completes. After routing update completes, no traffic will get through the failure.

MCS can detect failed channels and circumvent them during the remaining of its lifetime. Failure detection is different from congestion detection because no packet is able to traverse a failed channel to arrive at the receive side. Therefore, MCS uses an indirect feedback mechanism. Similar to congestion detection, MCS on the receive side will give a feedback of channel availability. The *Received Bitmap* in MCO is used for the feedback. MCS on the receive side will set the corresponding bit in *RB* (in flow states) to 1 when it receives a packet from one channel. When a packet is going back to the transmit side, *RB* should be put as *Received Bitmap* in MCO, and be unset afterwards.

MCS on the transmit side uses freshness (*FV*) to find failed channel. Algorithm 5 shows the relevant procedures. When MCS receives a MCO, *FV* should be updated according to *Received Bitmap* (line 2-6). When MCS transmits a data packet in one channel, the freshness of this channel can be changed from N to $N-1$ (line 7-9). On every RTT, the freshness of each channel will be decreased by 1 (line 13). If the freshness of one channel is 0, it means that there is no packet arriving at the receive side from this channel in the recent $N-1$ RTTs. Therefore, MCS regards this channel as an inactive one (line 15). MCS will not use a channel which is inactive.

In MCS, we set $N = 3$ (2 bits are required for the freshness of one channel). That means if a data packet has not been received in 2-3 RTT/RTO, it can be viewed as a failure. Assuming RTT is O(100μs) and RTO is O(10ms), MCS can detect failure in not longer than 100ms in practical, which can be faster than routing updates. This failure detection has tiny false positive when the packets sent in one channel are all dropped before they reach the receive side. Fortunately, false positive has no significant impact because there are still other available channels.

G. Flow States Removal

MCS should remove the flow states once after a connection is disconnected. A connection is disconnected after packets with FIN flag are received by both sides, and there is a LAST

Algorithm 5 Failure Detection

```
1: procedure ONRECEIVE(Packet p)
2:     rb ← Receive Bitmap in MCO
3:     for each channel c do
4:         if rb & (1 ≪ c) ≠ 0 then
5:             FV_c ← N
6:             AB_c ← true

7: procedure ONSENDDATA(Packet p, Channel c)
8:     if FV_c = N then
9:         FV_c ← N − 1

10: procedure ONNEWRTT                ▷ see in Algorithm 2
11:     for each channel c do
12:         if 0 < FV_c < N then
13:             FV_c ← FV_c − 1
14:         else if FV_c = 0 then
15:             AB_c ← false
```

ACK packet to acknowledge the second FIN packet as a final step. MCS keeps a *Close* flag in the flow states. If a packet with FIN flag is received, the *Close* flag is set to 1. When an ACK packet is received, if the *Close* flags in the corresponding flow and the reverse flow are both 1, the flow states will be removed.

Sometimes, the connection does not disconnect normally. An entry of flow states should be removed if it has not been accessed for some time (e.g. 1 second), with the help of timestamp (*TS*) in the flow states. This mechanism may make states of some flows be removed unexpectedly. Therefore, we allow the states of a flow to be initialized by any non-FIN packet (not only SYN packet). A flow whose states are removed by timeout can regenerate the state information in this way.

V. DISCUSSIONS

A. Channel Number

In Section IV, we assume MCS uses 8 channels. MCS can use more channels or fewer channels. Generally, more channels result in better traffic balancing. However, if MCS uses more than 8 channels (e.g. 16 channels), MCO and flow states must be larger than those described in Section IV, and the processing overhead is also slightly increased. According to our evaluation in Section VI, since the performance of 8-channel MCS is already approximate to that of MPTCP, we suggest to use 8-channel MCS as a default configuration.

B. Fairness

Different TCP flows scattered by MCS can fairly share the bandwidth because the window size update (in Section IV-D) complies with AIMD. However, the fairness between flows scattered by MCS and flows not scattered by MCS is not guaranteed. Since MCS imposes ECN on the scattered flows, it will starve the flows which are not scattered nor enable ECN because of the reason revealed in [19], [20]. Therefore, we had better deploy MCS in as many servers as possible to maintain the fairness.

C. GRO Friendliness

GRO, which can assemble small TCP segments into a large packet in NICs, is widely used to reduce the CPU overhead on the receive side. Different from [13], MCS does not need reordering in GRO. This is because there is no out-of-order from the aspect of *n-flows*. Hence the segments which are segmented from one data packet by GSO can be assembled by GRO easily. Note that the segments which are segmented from different data packets will not be assembled because the MCOs in these segments are different.

D. Adaption to Other Techniques

1) Adaption to VxLAN: VxLAN has some impacts on MCS. Since the source port number in the outer UDP header is often a hash of headers of the inner frame, the multi-channel scatter can still work. However, MCS relies on ECN. That means the network must support ECN in the context of encapsulation. Otherwise, the channel selection (Section IV-D) will degrade to round-robin and window control (Section IV-E) will not take effect.

2) Adaption to SR-IOV: In order to improve the performance of virtualization, many datacenters employ hardware acceleration techniques like SR-IOV, which allow VMs to get/put data from/to physical NICs directly. In this case, to realize multi-channel scatter, MCS must be implemented in the physical NICs. We do not further explore this area in this paper, but we believe it is not hard to develop NIC-based MCS because MCS is a switching-based solution without any buffer or any fine-grained timer.

E. Overhead

One main concern of MCS is its overhead. The communication overhead is obvious: 12-byte MCO in each packet. According to the traces uncovered by Facebook [23], the average packet size in the network fabric is about 400 bytes for web servers, and about 1400 bytes for Hadoop servers. Therefore, the average communication overhead of MCS is about 3% for web servers and 0.9% for Hadoop servers. This overhead is acceptable because traffic balancing is improved (both average and maximal flow completion time can be reduced).

The computation overhead of MCS deserves more attention. We only consider the extra overhead beyond ordinary virtual switches like OVS. Besides switching-based operations like search in the flow table and update of packet headers, MCS introduces flow states manipulations and MCO manipulations. Fortunately, all the data used by MCS can be stored in the CPU cache. To evaluate the computation overhead, we implemented MCS in the datapath of OVS as a prototype. We let two servers (1Gbps NICs) connect directly with a link. We use *iperf* to generate a TCP flow with a throughput larger than 900Mbps. When original OVS is used in both servers, the *sys* CPU utilization is about 0.53% (Intel i7-4790, 3.60GHz).

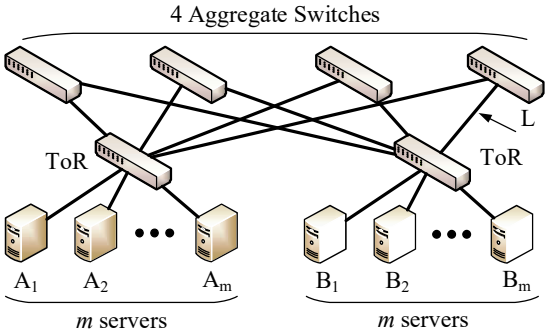


Fig. 4. Simple two-tier topology used in the benchmark simulation.

When MCS-enabled OVS is used, the *sys* CPU utilization is increased to 0.49% (smaller than 1.1×). It demonstrates the computation overhead of MCS is acceptable.

VI. Evaluation

A. Methodology

We evaluated MCS with simulations in NS3. To show the improvement, we used ECMP with VCC [19] as the baseline solution. As comparisons, we also implemented other two traffic balancing solutions. First is MPTCP [24] with 8 subflows (with DCTCP congestion control). A data packet is always assigned to the subflow with the largest available window. Second is round-robin packet spraying with a reorder buffer and DCTCP congestion control at end hosts, denoted by PerPacket.

We performed simulations of two different scenarios. In the first scenario, we simulated a simple topology as shown in Fig. 4. There are two edge switches (ToR), both connect to 4 aggregate switches and m hosts. All links in this topology have a capacity of 10Gbps. Host A_i sends 50MB data to host B_i. All the flows start at the same time. We regard this scenario as a benchmark. The main metric is flow completion time, especially average flow completion time (AFCT) and maximal flow completion time (MFCT). To eliminate the impact of randomness, we repeated the benchmark simulation for tens of times to get the mean value of AFCT and MFCT.

In the second scenario, we simulated an 8-ary fat tree topology [14] (with 128 servers), and all links have a capacity of 10Gbps. We generated all-to-all TCP traffic randomly. We used two types of TCP flows in this simulation. 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 pareto distribution. And the interval time of two consequent flows is subject to exponential distribution. The overall average link utilization is 50%. The simulation results in this scenario can reveal the effect of MCS in practical.

B. Comparing Different Channel Numbers

To evaluate the impact of the channel number, we ran the benchmark simulation with different channel numbers. Fig. 5 shows the results (AFCT and MFCT). We find that MCS is better than Baseline even when there are only 2 channels.

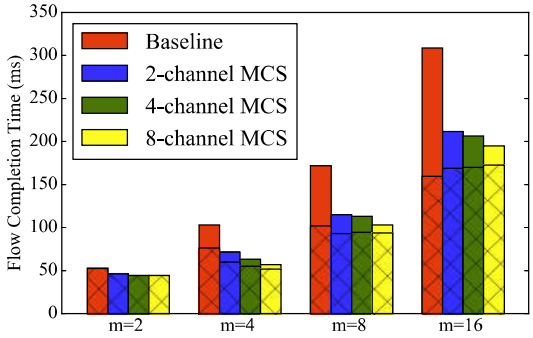
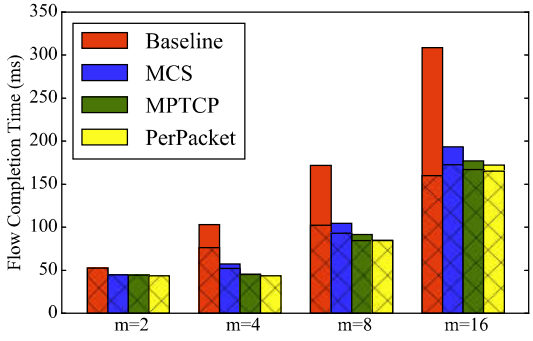
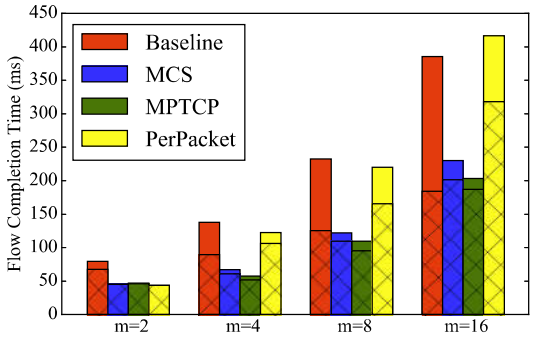


Fig. 5. Results of the benchmark simulation when different channel numbers are used in MCS. The height of each bar represents MFCT, while the height of each hatched area represents AFCT.



(a) Symmetric Topologoy



(b) Asymmetric Topologoy

Fig. 6. Results of the benchmark simulation which compares ECMP, MCS, MPTCP and PerPacket. The height of each bar represents MFCT, while the height of each hatched area represents AFCT.

More channels can introduce better performance. However, the benefit of increasing channels is limited. That is why we do not consider 16-channel MCS (needs larger MCO and flow states). In the following, we take 8-channel MCS as the default configuration.

C. Comparing Different Traffic Balancing Solutions

We compared MCS with Baseline, MPTCP and PerPacket. Fig. 6(a) shows the simulation result. PerPacket is the best

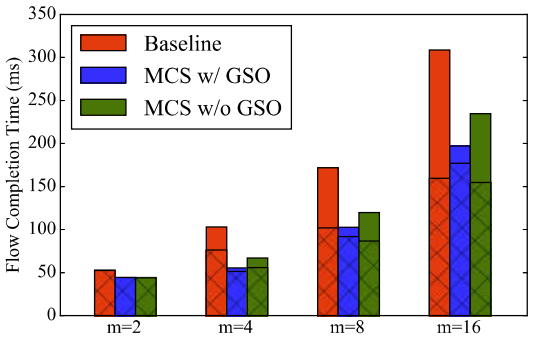


Fig. 7. Results of the benchmark simulation in the GSO case. The height of each bar represents MFCT, while the height of each hatched area represents AFCT.

solution because it can realize completely even traffic distribution. The performance of MPTCP is very close to PerPacket. Baseline has the worst performance. MCS is much better than Baseline and close to MPTCP. Since MCS is a switching-based solution, the performance gap between MCS and MPTCP is acceptable. Note that a solution with a higher MFCT may have lower AFCT because when all the links are fully utilized, unfair bandwidth sharing will lead to lower AFCT.

Asymmetric topology is a common situation in real datacenter networks, which is often caused by failures. We also compared the benchmark results when the topology is asymmetric: we changed the capacity of link L in Fig. 4 to 5Gbps. We did not consider weighted ECMP [25] or Weighted PerPacket. Fig. 6(b) shows the result. The performance of Baseline is still not good. However, PerPacket is almost as poor as Baseline when m is large. This consequence is expectable [17], because of fixed weight and the incorrectness of congestion control. In fact, the reorder buffer in PerPacket consumes more than 1MB because of packet out-of-order in this case. MPTCP still has a very good performance, and MCS is close to it.

D. GSO Impact

GSO is very useful to reduce the computation overhead of data transmission. In this simulation, we evaluate the impact of GSO. Fig. 7 shows the result in this case. It is demonstrated that MCS with GSO is still much better than Baseline. However, it is a little worse than MCS without GSO. This is because with GSO, the granularity of traffic scheduling (among different channels) may be coarser. The degradation extent depends on the data size in one packet before segmentation.

E. Fairness

We evaluated the fairness among different flows scattered by MCS. Fig. 8 shows the throughputs of the 4 flows in the benchmark simulation when $m = 4$. The throughput is calculated at the receive side every 1ms. Each line in this figure represents one flow. It demonstrates although the throughputs change with time, they fluctuate around approximate values (about 8Gbps) after the slow start stage.

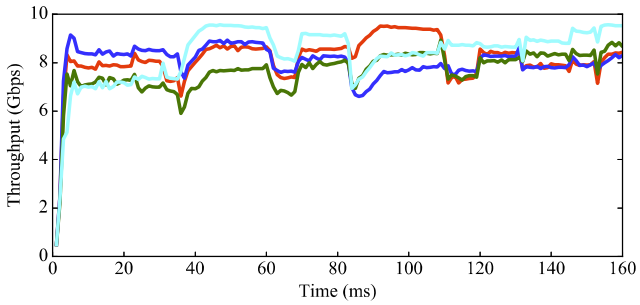


Fig. 8. Results of the benchmark simulation when $m = 4$. The throughput is calculated at the receive side every 1ms.

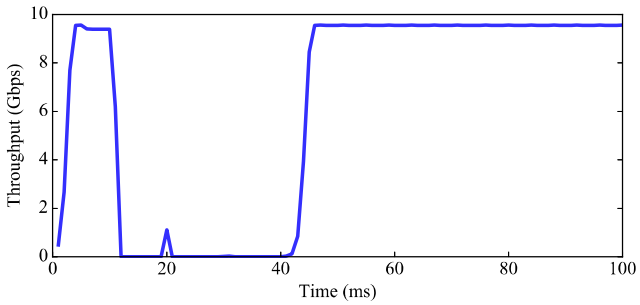


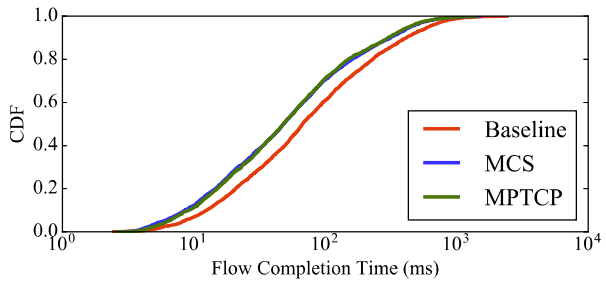
Fig. 9. Results of the benchmark simulation in the failure case. The throughput is calculated at the receive side every 1ms.

F. Failure Detection

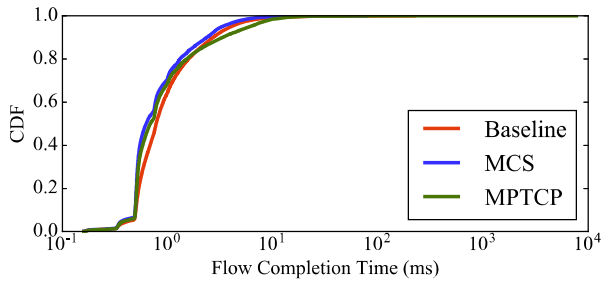
We also evaluated the failure detection of MCS. In this simulation, we considered only one flow from host A_1 to host B_1. We let link L fail at the time 10ms after the flow starts. The TCP timeout is set to 10ms. Fig. 9 shows the throughput of this flow when some channels go through link L. At 10ms after the flow starts, the throughput decreases dramatically because there are packet losses. Since all the duplicated ACKs are filtered in this situation, fast retransmission cannot be triggered. After a timeout (at 20ms), the send side can retransmit data. However, it encounters packet losses again because the failed channels are still used. At about 42ms, MCS detects the failed channels. Then it will skip the failed channels forever, and the throughput increases to the normal value. It takes MCS about 32ms to detect failed channels in this example, which complies with the analysis in Section IV-F.

G. Fat-Tree Simulation

In this simulation, we simulated the second scenario described in Section VI-A. Fig. 10(a) and Fig. 10(b) show the CDF of flow completion time of large flows and small flows separately. For large flows, the performances of MCS and MPTCP are close to each other and both better than Baseline (reduce the flow completion time by about 30% at 50th percentile). For small flows, MCS still outperforms Baseline and is approximate to MPTCP.



(a) Large Flows



(b) Small Flows

Fig. 10. CDF of flow completion time in the fat-tree simulation.

VII. Conclusion

In this paper, we propose a traffic balancing solution based on edge-switching, named MCS. MCS can scatter packets in one TCP flow to several different forwarding paths (channels) to make traffic distribute more evenly. It can filter duplicated ACKs triggered by packet out-of-order to avoid unnecessary retransmission and window suppression. MCS uses congestion detection with ECN to update the weight of different paths and avoid packet loss. Failed channels can be detected and circumvented in MCS. MCS introduces acceptable communication and computation overhead.

We evaluated MCS with simulations. The benchmark simulation results show that 8-channel MCS has a performance close to MPTCP, even in the case of asymmetric topology. Besides, MCS has good robustness when there is a failure. In a large-scale datacenter, MCS can reduce flow completion time of both large flows and small flows effectively. In a word, MCS is a good choice for the datacenter operator to achieve traffic balancing when they cannot control the operating systems of end users.

Acknowledgment

This research is supported by the National Science Foundation of China (No.61472213) and sponsored by Huawei. Jun Bi is the corresponding author.

References

[1] 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.

[2] 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.

[3] "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.

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

[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] "HULA: Scalable Load Balancing using Programmable Data-Planes," https://ons2016.sched.org/event/6IRj/sosr-hula-scalable-load-balancing-using-programmable-data-planes, 2016.

[9] 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.

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] "Avoiding Network Polarization and Increasing Visibility in Cloud Networks Using Broadcom Smart Hash Technology," http://tec.icbuy.com/uploads/2013/1/5/Smart-Table_Technology_%E2%80%94_Enabling_Very_Large_Server,_Storage_Nodes,_and_Virtual_Machines_to_Scale_Using_Flexible_Network_Infrastructure_Topologies.pdf, 2013.

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

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

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

[20] 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.

[21] K. Leung, V. Li, and D. Yang, "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges," *IEEE Transactions on Parallel and Distributed Systems*, April 2007.

[22] X. Wei, C. Xiong, and E. Lopez, "MPTCP Proxy Mechanisms," https://tools.ietf.org/html/draft-wei-mptcp-proxy-mechanism-02, 2015.

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

[24] C. Paasch, R. Khalili, and O. Bonaventure, "On the Benefits of Applying Experimental Design to Improve Multipath TCP," in *CoNEXT*, 2013.

[25] 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.