On the Impact of Packet Spraying in Data Center Networks

Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella
Purdue University

Abstract—**Modern data center networks are commonly organized in multi-rooted tree topologies. They typically rely on equal-cost multipath to split flows across multiple paths, which can lead to significant load imbalance. Splitting individual flows can provide better load balance, but is not preferred because of potential packet reordering that conventional wisdom suggests may negatively interact with TCP congestion control. In this paper, we revisit this "myth" in the context of data center networks which have regular topologies such as multi-rooted trees. We argue that due to symmetry, the multiple equal-cost paths between two hosts are composed of links that exhibit similar queuing properties. As a result, TCP is able to tolerate the induced packet reordering and maintain a single estimate of RTT. We validate the efficacy of random packet spraying (RPS) using a data center testbed comprising real hardware switches. We also reveal the adverse impact on the performance of RPS when the symmetry is disturbed (*e.g.,* during link failures) and suggest solutions to mitigate this effect.**

I. Introduction

In recent years, data centers have become the cornerstones of modern computing infrastructure. Many distributed processing applications (*e.g.,* search, social collaboration, high-performance computing) are routinely run in large-scale data centers that contain upwards of 100,000 servers. Because of the inherently distributed nature of computation, the network fabric that connects these different servers becomes critical in determining the performance of these applications, which translates to user satisfaction and ultimately money for the data center operator.

Meanwhile, recent papers characterizing data center traffic have found significant spatial and temporal variation in traffic volumes [1], [2], [3], which means that the data center network design cannot pre-assume a given traffic matrix and optimize the routing and forwarding for it. Recent trends therefore favor network fabric designs based on multi-rooted tree topologies with full bi-section bandwidth (or with low oversubscription ratios such as 4:1) such as the fat-tree architecture [4]. While in theory these topologies provide (almost) full bi-section bandwidth, achieving this is hard in practice because it is dependent on the underlying routing scheme. Traditional single-path routing is inadequate since the full bi-section bandwidth guarantee assumes that all paths that exist between a pair of servers can be fully utilized. Thus, equal-cost multipath (ECMP) has been used as the *de facto* routing algorithm in these data centers.

In ECMP, flows (as identified by the TCP 5-tuple) between a given pair of servers are routed through one of the paths using hashing; therefore, two flows between the same hosts may take different paths, and ECMP does not affect TCP congestion control. However, because not all flows are identical in their size (or their duration), this simple scheme is not sufficient to prevent the occurrence of hot-spots in the network. In a recent study [2], the authors find that 90% of the traffic volume is actually contained in 10% of flows (heavy-hitters); if two heavy-hitter flows are hashed to the same path, they can experience significant performance dip. Several solutions (*e.g.,*, Hedera [5], Mahout [6]) focus on addressing this hot-spot problem by tracking and separating long-lived (elephant) flows among link-disjoint paths. However, it is fundamentally not always feasible to pack flows of different size/duration across a fixed number of paths in a perfectly balanced manner.

A recently proposed solution called MP-TCP [7] departs from the basic assumption that a flow needs to be sent along one path, by splitting each flow into multiple sub-flows and leveraging ECMP to send them along multiple paths. Since MP-TCP requires significant end-host protocol stack changes, it is not always feasible in all environments, especially in public cloud platforms where individual tenants control the OS and the network stack. Further, it has high signaling and connection establishment complexity for short flows, which typically dominate the data center environment [2], [3].

In this paper, we study the feasibility of an intuitive and simple multipathing scheme called *random packet spraying* (RPS), in which packets of every flow are randomly assigned to one of the available shortest paths to the destination. RPS requires no changes to end hosts, and is practical to implement in modern switches. In fact, many commodity switches today (*e.g.,* Cisco [8]) already implement a more sophisticated per-destination round-robin packet spraying technique.

RPS approach, however, can potentially result in reordering of packets that belong to a flow—a problem that is known to negatively interact with TCP congestion control[1], at least in the wide-area networks [9]. Specifically, packets in a given flow that traverse multiple paths with potentially different latencies may arrive at the receiver out of order, *i.e.,* later-sent packets may be received ahead of earlier-sent ones. Since TCP can not distinguish reordered packets from lost packets, it will trigger congestion avoidance by cutting down its congestion window leading to suboptimal performance. Because of the potential packet reordering and its implication on TCP, networking researchers as well as practitioners have cautiously kept packet

[1] This is the reason that the feature though supported in commodity switches is not turned on by default

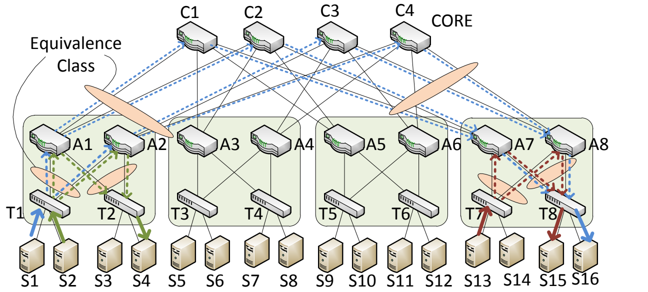


Fig. 1. Fat-tree topology with equivalence classes and imbalance with ECMP.

spraying out of consideration for data center networks.

In this paper, we make two *key observations* that together suggest RPS is unlikely to be affected by packet reordering and hence a promising multipathing scheme for data center networks. First, we observe that modern data center networks based on multirooted tree topologies tend to be *symmetrical*, which essentially causes links along multiple paths between a source-destination to be grouped into *equivalence classes*. As a result, paths between a source-destination pair are likely to exhibit similar queue build-up, keeping latencies roughly equal. In addition, data center networks are often engineered to provide low latencies to service latency sensitive traffic anyway. Solutions such as DCTCP [10] and HULL [11] provide even lower latencies at the (slight) expense of throughput. Low end-to-end latencies help RPS since the worst case latency differential between two paths is also going to be small.

Second, standard TCP originally designed for the wide area Internet already has a built-in mechanism to tolerate mild packet reordering. In particular, TCP does not perform fast retransmit unless 3 duplicate ACKs (DUPACKs) arrive for the same packet. Newer implementations of TCP in the Linux kernel are even more robust to packet reordering. They use timestamps and DSACK options to detect spurious fast retransmissions. If a spurious fast retransmission is detected, TCP reverts the reduction in congestion window size. Also, the TCP duplicate ACK threshold is dynamically adjusted. Hence, even if some occasional reordering happens in the data center network under RPS, the reordering may only mildly affect TCP performance.

In this paper, we conduct an empirical study to validate these observations and study the overall performance under RPS multipath routing using a *real testbed* comprising of hardware NetFPGA-based switches organized in 4-ary fat-tree topology. Our experiments indicate that our observations typically hold true in practice and as a result, RPS achieves much better network-wide TCP throughput than ECMP.

While our experiments above show that RPS works well in symmetric topologies, production data centers are prone to link failures which may disturb the overall symmetry of the network. Such asymmetry in the topology can potentially lead to unequal load on links leading to sub-optimal throughput of RPS. However, no prior studies have quantified the impact of failures on the performance of RPS in data center networks. Thus, in second part of this paper, we conduct detailed empirical analysis of RPS under failure conditions. We observe that if RPS alone is used, it can lead to significantly lower throughput in failure scenarios. We observe however that if the queue lengths are kept sufficiently small using simple active queue management scheme such as Random Early Discard (RED), the performance of RPS can be much better, almost comparable to complex solutions such as MP-TCP.

Contributions. In summary, the main contributions of the paper include the following. (1) We conduct a first of its kind empirical study to debunk the myth that random packet spraying is inherently harmful to TCP, in the context of designing an effective multipathing scheme for data center networks. (2) Using a data center testbed with real RPS implementation over NetFPGA switches, we conduct detailed study on the reasons why RPS performs better than existing schemes such as ECMP and similar to MP-TCP (with long-lived flows). (3) We also study the adverse effect of link failures on the performance of RPS. Exploiting the key insight that smaller queues result in better performance even under failures, we propose an approach based on RED to mitigate these adverse effects.

II. Random Packet Spraying (RPS)

In this section, we start with an overview of RPS followed by theoretical analysis on why we expect RPS to perform well in data center networks.

A. RPS Overview

The basic idea of RPS is simple: Like ECMP, RPS uses all the equal-cost shortest paths between every source and destination pair. However, instead of hashing the flow key of a packet to determine the next hop for forwarding as in ECMP, RPS randomly spreads all packets that belong to each flow equally along different shortest paths. For example, in Figure 1, we show a flow from $S1 - S16$ that traverses the paths $S1 \rightarrow T1 \rightarrow \{A1, A2\} \rightarrow \{C1, C2, C3, C4\} \rightarrow \{A7, A8\} \rightarrow T8 \rightarrow S16$ to reach the destination. Thus, if the flow consists of 100 packets, roughly 25 packets will be routed through each of the four paths via core routers $C1 - C4$.

As shown before in literature [9], packet spraying can lead to severe packet reordering in the wide-area—the packets of a flow which take different paths may have orders of magnitude differences in latencies since there is no guarantees that the paths will be of equal lengths or have similar congestion. Even in data center environments, where latencies are low and uniform, RPS will potentially introduce packet reordering. TCP performs poorly in the presence of packet reordering. When the TCP sender receives three duplicate acknowledgments (DupACK), it assumes that a segment has been lost and reduces its congestion window size, which results in a drop in throughput. TCP maintains an estimate of round-trip (RTT) times. If paths have hugely varying latencies, TCP's RTT estimate will also be meaningless, which can lead to spurious retransmissions and timeouts. In fact, this concern of *potential* packet reordering is why none of the existing data centers use or existing proposals advocate the use of simple packet spraying schemes.

We make three key observations that indicate that packet spraying techniques like RPS are unlikely to result in significant packet reordering, and consequently should not affect TCP's performance in data center networks that employ multi-rooted tree topologies such as the fat-tree. Specifically:

Observation 1. In a multirooted tree topology like a fat-tree shown in Figure 1, links can be grouped together into *equivalence classes*. All links within each equivalence class have equal amount of load if all flows in the networks use RPS. Thus, even though each flow is routed along several paths, each of these paths is similarly loaded. So, the latency differential between these paths is expected to be quite small, and the amount of induced reordering due to packet spraying is likely to be small. (We analyze this in more detail next.)

Observation 2. TCP congestion control is robust to small amount of packet reordering in the network anyway. Given that TCP was designed for the wide area network, where some amount of reordering can happen due to failures and other events. The sender typically waits for 3 duplicate ACKs to infer that a loss event has occurred after which it performs fast retransmit and cuts its window in half. Besides this, the TCP implementation in newer Linux kernels detects spurious fast retransmission using the DSACK and timestamp options of TCP to rollback any erroneous reductions in the congestion window [12]. TCP also proactively avoids spurious fast retransmissions in the future by increasing the DupACK threshold [13].

Observation 3. Even if packet spraying using RPS induces slightly more fast retransmits compared to say a flow based technique like ECMP, the extra loss in throughput, due to the sender reducing its congestion window by half every time a fast retransmit event occurs, can be a small penalty compared to the better usage of the total aggregate available bandwidth across all paths. Thus, RPS' overall performance will be likely better than that of ECMP.

Further, data center operators are increasingly more concerned about end-to-end latencies. Thus, future data center designs are likely to ensure low and uniform latencies, using mechanisms such as HULL [11], DCTCP [10], DeTail [14]. If latencies across all paths are low and uniform, TCP end-host can maintain a single estimate of RTT for all paths.

In spite of low latencies in data center networks and improvements to TCP, researchers have focused mainly on load balancing schemes which avoid packet reordering. No measurement studies have been conducted to study the impact of these improvements on packet spraying in data center networks. Our analysis in this paper shows that TCP is able to perform well with packet spraying in a data center environment, as long as packets are sprayed over equal length paths and queue lengths are kept almost equal along all paths. We hope that this result will encourage more research in simple packet spraying techniques for data centers.

B. Analysis

We formalize the concept of equivalence classes stated in Observation 1 above, which gives a key reason why different from in the Internet, significant packet reordering is unlikely to happen when RPS is running in data center networks which typically employ multi-rooted tree typologies such as fat trees.

When RPS is used to route packets between a source and a destination via all equal-cost paths, an *equivalence class* comprises all outgoing links from the switches at the same hop along all the equal-cost paths. For simplicity, we exclude links to/from end hosts (leaves in the tree) in the discussion. In a depth-h K-ary fat tree (each switch has K ports), each flow goes through $2h$-hop equal-cost paths and passes through $2h$ equivalance classes of links. Note different source-destination pairs can share some equivalance classes. Together, there are $2h$ *types* of equivalance classes in a depth-h fat tree. In particular, there are 4 equivalance classes in the depth-2 fat tree in Figure 1:

Type 1: A Type 1 class consists of the links from a ToR switch, ToR_i, to the $\frac{K}{2}$ aggregate switches Agg_j within the same pod.[2]

Type 2: A Type 2 class is the mirror image of a Type 1 class, and consists of the links from the $\frac{K}{2}$ aggregate switches Agg_j within a pod, to a ToR switch, ToR_i.

Under RPS, for an X-packet flow, the expected number of packets that will be routed through each of the $\frac{K}{2}$ links in a Type 1 or Type 2 equivalence class is $\frac{2X}{K}$.

Type 3: A Type 3 class consists of the links from all the aggregate switches Agg_i within a pod, to all $\frac{K^2}{4}$ core switches, C_n.

Type 4: A Type 4 class is the mirror imagine of a Type 3 class, and consists of the links from all the core switches C_n, to all aggregate switches, Agg_j, within a pod.

Under RPS, for an X-packet flow, the expected number of packets that will be routed through each of the $\frac{K^2}{4}$ links in a Type 3 or Type 4 equivalence class is $\frac{4X}{K^2}$.

Example. Consider the two paths between S1 and S5 in Figure 1. There are four equal-cost paths between them. The first hops of all paths form to the Type 1 equivalence class ($T1 \rightarrow A1$, $T1 \rightarrow A2$), the second hops of all paths belong to the Type 3 equivalence class ($A1 \rightarrow C1$, $A1 \rightarrow C2$, $A2 \rightarrow C3$, $A2 \rightarrow C4$), and so on. This hop-by-hop equivalence holds for paths between all hosts in the fat tree even if they are in different pods, in the same pod, or under the same ToR switch.

The equal spread of packets of each flow among the links in its hop-by-hop equivalence ensures that, given any set of flows, load and hence the queue lengths (measured in number of packets) among the links in each eqivalent class stays the same. This in turn implies that for a given flow, its packets traversing different paths will encounter the same queuing delay, and hence the same end-to-end delay. Thus, the receiver will observe only a few reordered packets due to small differences in queue lengths introduced by (1) difference in packet sizes; (2) flow sizes are not always in multiples of the number of paths; and (3) timing issues. However, these issues

[2] The set of switches {T1,T2,A1,A2}, {T3,T4,A3,A4}, etc. in Figure 1, are referred to as pods in the fat-tree.

are expected to cause only a small queue length differential which results in a small amount of reordering within the network. We experimentally confirm this in Section III.

III. Evaluating RPS

In this section, we evaluate RPS using a real hardware testbed. We first discuss the testbed configuration and our implementation of RPS and ECMP. We then provide comparisons of RPS with and ECMP and MP-TCP. Finally, we empirically confirm the three observations made in the previous section that explain the good performance of RPS in our testbed.

A. Testbed Configuration

Our testbed has 36 servers connected in a 4-ary ($k = 4$) fat-tree [4] topology (as shown in Figure 1). All the servers are equipped with 4GB RAM, Intel Xeon 2.40GHz quad-core processors running Centos 5.5 and two 1Gbps Ethernet ports. We have 20 NetFPGA boards, each deployed on a server, and interconnected in a fat-tree topology via 1 Gbps Ethernet links. Rest of the 16 servers form the endhosts connected to this network. A fat tree has an oversubscription ratio of 1:1. Removing two of the four core switches would have resulted in an oversubscription ratio of 2:1 but it would have reduced path diversity; there would be just two paths between hosts in different paths, which can bias our results significantly. Other oversubscription ratios (4:1, 8:1) would not be possible even. To overcome this, we emulate oversubscription of approximately 4:1 (and 8:1) by rate-limiting the core links to 230Mbps (and 115Mbps). The seemingly arbitrary choice of 230Mbps (instead of 250Mbps) stems from the limitations of the NetFPGA rate limiter, which allows only a few discrete values to choose from.

1) Implementation of RPS and ECMP: We implemented RPS and ECMP on NetFPGA switches by modifying the code base already provided by NetFPGA. For a packet arriving at the switch, we generate a random number (using the library provided by NetFPGA) to determine the output port (among all eligible output ports) to which the packet is forwarded. Implementing this is quite simple; we needed only about 100 lines of verilog code to implement this technique. RPS is a purely switch-based solution and does not require any help or modification at the end hosts.

2) Implementation of MP-TCP: To enable MP-TCP, we deployed the publicly released Linux kernel for MP-TCP [15] at the end hosts. This kernel still has a few performance and stability problems. For instance, we observed kernel panics sometimes when MP-TCP was handling many short-sized flows simultaneously. This prevented us from running experiments involving many short flows with the MP-TCP kernel. For long flows, we observed more stable results for MP-TCP. MP-TCP has also been noted to have a sub-standard performance with short flows because the control overhead of setting up and terminating many subflows becomes significant. For the above reasons, we present MP-TCP results for long flows in this paper. Since ECMP performs well with short and long flows, we compare RPS with ECMP in experiments involving both short and long flows.

B. TCP Throughput under Packet Spraying

We first measure the throughput obtained by long lived TCP flows in a random permutation matrix (similar to [16]). In such a setup, each host in the topology is either a sender or a receiver of exactly one TCP flow. All senders are randomly paired with receivers. A netperf client running at the sender sends a TCP flow to its receiver for the duration of the experiment. We measure the average throughput as a percentage of the ideal throughput and also compare performance of TCP flows under different schemes.

Figure 2 clearly depicts the gain in throughput experienced by TCP flows under a packet spraying technique (RPS). Even under different degrees of oversubscription, the throughput obtained under RPS is higher than those measured under MP-TCP or ECMP-like techniques. The low average throughput in case of ECMP-based forwarding can be attributed to the fact that two or more TCP flows may be forwarded over the same core link which becomes a bottleneck. For the entire flow duration of the flow, that link remains the hot spot in the network while leaving other links underutilized. Due to static allocation of paths in ECMP, if some of the flows are unlucky and are routed through a congested link, then they suffer permanently for the entire duration resulting in poor throughput.

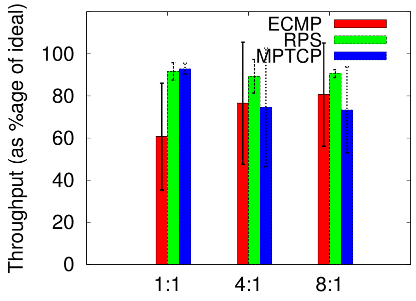


Fig. 2. Throughput for Permutation Matrix

Under RPS, average throughput achieved is about 90% of the ideal bandwidth in all 3 cases with different oversubscription ratios. Figure 2 also demonstrates that the variance in throughput obtained by different TCP flows is small. MP-TCP also achieves about 90% in case of a non-oversubscribed topology (subscription factor 1:1). This is consistent with results reported in [16] for a similar experimental setup. In case of oversubscribed topology though, the average throughput achieved by MP-TCP flows seems to suffer and it decreases from 90% to about 75%. This poor performance may be an artifact of MP-TCP itself or the released implementation of MP-TCP; unfortunately, there is no easy way for us to know precisely at the moment.

To study the effect of path diversity on RPS, we repeated the above experiment in simulation using fat trees with ($k =$) 6 and 8 pods. The number of paths between end hosts is 9 and 16 respectively ($k^2/4$). Intuitively, when the number of paths increases, the probability of packet reordering in packet spraying increases. However, we observed that the drop is not substantial showing that our analysis in Section II-B still largely holds.

C. Data Transfer Time

We repeat the experiment performed in [16] (but with mixed short and long flows) to study how much time TCP takes to transfer the same amount of data under different schemes. This experiment shows the ability of the underlying mechanism to consume bandwidth more efficiently to transfer the same amount of data. In this experimental setup, each end host executes two clients which have to transfer 2GB of total data, which is divided into many flows with flow sizes drawn from the real data-center flow size distribution reported in [1]. A client sends these flows in sequence to randomly chosen destinations. The client forks a new netperf client for each flow. All clients begin simultaneously. We plot the median, and first/third quartiles of the completion time of all clients in Figure 3(a).

We observe that TCP flows are able to complete faster under RPS as compared to ECMP. (We cannot do this experiment with MP-TCP as it is unstable when there are large number of concurrent connections.) With 1:1 oversubscription, we observe that ECMP and RPS perform equally well. This is because in such a topology and flows being setup between random pairs of hosts, the edge links are more likely to be the bottleneck than the core of the network. So, TCP does not benefit from a better traffic splitting technique. In case of 4:1 or 8:1 oversubscribed networks, the packet spraying technique helps TCP flows to utilize the available capacity in a much more efficient manner in spite of the reordering. Hence, the time to transfer the same amount of total data is 25% smaller in case of RPS than ECMP.

D. Packet Latencies

Packet latency is another important metric for flows in data center networks. Recent works like [11], [10] have focused on reducing packet latencies in the network so that applications can satisfy SLAs (service level agreements). To study the effect of packet spraying on packet latencies we ran background traffic between 14 (out of 16) end hosts in our testbed. The flow sizes for background traffic were drawn from the distribution in [1]. The flow arrival rate followed an exponential distribution and variable mean. We sent 200 back-to-back ping packets between the two hosts that did not carry background traffic. The two hosts which do not carry background traffic exchange ping packets. For ECMP, MP-TCP and RPS, a ping packet randomly takes one of the 4 paths between the end hosts. Buffers at the two end hosts are always empty because they do not transmit or receive any of the background traffic. So, ping packets experience similar latencies at end hosts. Since the ping packets are sent back-to-back, we can assume that packets taking the same path also observe very similar latencies. So, the variation in latencies between packets is almost entirely due to variation in latencies between different paths in the network. Figure 3(b) shows the mean RTT for the 200 back-to-back ping packets and the errorbars show the mean deviation reported by ping. We observed that the latency varied widely with ECMP indicating that different paths between the two hosts different loads. Packets experience similar mean latencies with RPS and MP-TCP, but experience higher variance with MP-TCP.

E. Effect on MapReduce

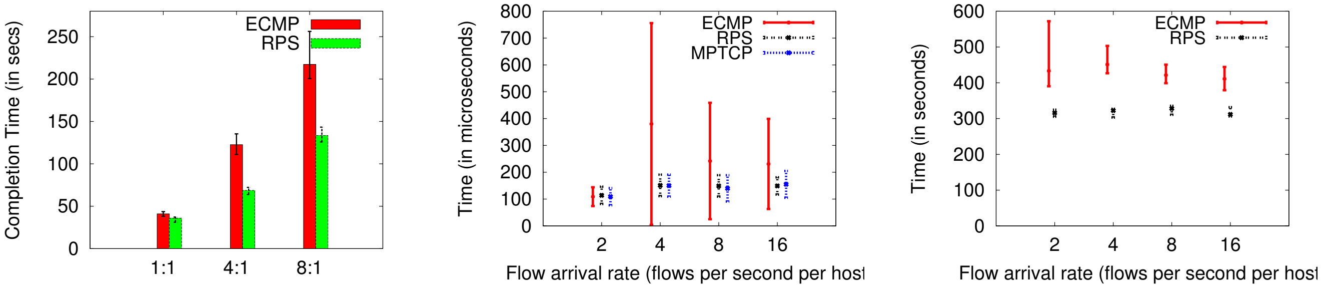
In order to quantify the impact of packet spraying on applications, we run Hadoop Sort application on 4 of the 16 end hosts in our testbed (other 12 hosts have background traffic between them as before). To emulate a network constrained cloud application, we reduce the bandwidth of each link to 115Mbps but kept the oversubscription ratio at 1:1.

Figure 3(c) shows the time taken for the shuffle phase of Hadoop sorting 4GB of data averaged over 3 runs. On the x-axis, we vary the intensity of background traffic (that is, flow arrival rate of background traffic). We observe a 20% to 30% reduction in shuffle time with RPS. Also, the variance in completion time is much smaller with RPS than ECMP. Since a fat-tree is provides full bisection bandwidth, end hosts running Hadoop can communicate with each other at full line rate even when background traffic intensity increases. So, increasing background traffic intensity does not affect shuffle phase completion time. RPS completes the sort phase quicker than ECMP because it is able to utilize the available bisection bandwidth more efficiently than RPS. We were not able to perform this experiment with MP-TCP due to the stability issues with MP-TCP implementation.

F. Analysis of Packet Spraying

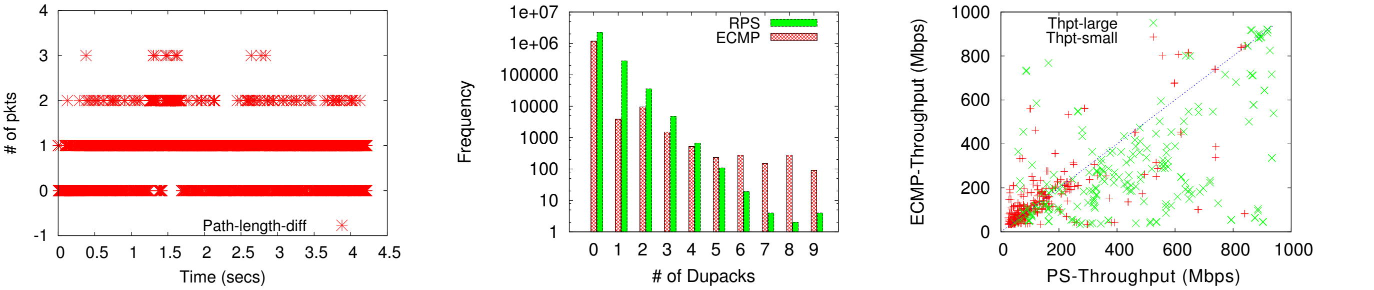
Now, we conduct experiments to validate our analysis and our understanding of why TCP performs well under packet spraying in data center networks. Specifically, we empirically validate our key observations made in Section II using experiments conducted on our testbed. In these experiments, each end host starts new flows with start times based on a Poisson distribution with 2 arrivals per second. As before, flow sizes are drawn from the distribution reported in [1]. The traffic matrix was executed two times, once each for ECMP and RPS. Both times, the random number seeds were initialized to the same value to ensure that flows of the same size were started at the same time in both runs, allowing us to make a flow-by-flow comparison between RPS and ECMP.

1) Queue length differential: In Section II-B, we argued that packet reordering will be limited because all paths between a pair of hosts have similar latencies. Latencies are largely determined by queue lengths that a packet encounters at every hop. We polled the queue lengths at every hop along a path between a pair of hosts in our testbed. We were able to poll about 1000 times per second; this is the maximum rate allowed by our NetFPGA platform. We used NTP to synchronize timestamps at all switches. By summing the queue lengths of all hops, we determined the path queue-length, that is, the total queue-length that a packet would encounter if it were forwarded along that path. We did this for all paths



(a) Completion Time (b) Ping RTT (c) Hadoop Shuffle Time

Fig. 3. Performance of RPS with different traffic patterns.



(a) Q-length diff. across a src-dest pair (b) # of consecutive DupAcks (c) Throughput comparison with ECMP

Fig. 4. Microscopic analysis to validate our understanding about RPS performance.

between a source destination pair and plotted the instantaneous difference between the highest and the lowest path queue-lengths in Figure 4(a). In a perfectly balanced network, this path queue-length differential will always be zero. However, in Figure 4(a), the queue-length differential is less than or equal to one 93% of the time. Flows between the pair of end hosts under observation may experience some reordering when the path queue-length differential increases to two or three, but that is relatively infrequent (less than 7%).

2) DupACKs: We now measure the number of dupACKs that an end host will receive. We log the number of dupACKs received at the sender and plot Figure 4(b). The x-axis shows the number of dupACKs that the sender received for a particular TCP segment. For both ECMP and RPS, the sender received no dupACKs for almost one million TCP segments. RPS received exactly one dupACK almost 200,000 TCP segments, and exactly two dupACKs for 30,000 and so on. We see that the frequency of k dupACKs reduces exponentially with increasing k.

TCP does not enter fast-retransmission until it sees greater number of dupACKs than the dupAckThreshold (default is 3). So, the first three bars in the figure will not lead to a drop in throughput. Since the number of dupACKs reduce exponentially, we observe that fewer that 55,000 (about 2%) of the transmitted TCP sequence numbers cross the three dupACK threshold. Surprisingly, we find a similar order of magnitude dupACKs in ECMP. However, dupACKs in ECMP are entirely due to packet losses and therefore cause a drop in throughput. But, dupACKs in RPS are due to a combination of reordered and lost packets. While it is difficult to ascertain the exact number of reordered and lost packets, note that dupACKs due to reordered packets are handled well as stock Linux TCP implementation has adaptive dupACKThreshold to reduce spurious reductions in TCP congestion window due to reordered packets. In any case, RPS should perform, if not better, no worse than ECMP because of dupACKs. But, flows have higher available bandwidth in the case of RPS than ECMP due to the availability of combined bandwidth across all sub-paths, which increases the performance of RPS compared with ECMP.

3) Effect on throughput of individual flows: We compare the performance of large and small flows in RPS and ECMP. It is generally expected that small flows should obtain good throughput under ECMP; it is the large flows that usually suffer. In Figure 4(c), we plot the throughput observed by the same flows under ECMP and RPS. The x-coordinate of a point corresponds to a flow's throughput under RPS while the y-coordinate is its throughput with ECMP. Points below the diagonal line indicate higher throughput with RPS. The green crosses represent large flows (greater than 1MB) while the red pluses are for short flows. From the graph, we can clearly see that large flows benefit most with RPS, while small flows perform equally well with both ECMP and RPS. Although only 10% of the flows in the distribution are large, the difference in throughput is significant enough to affect applications. Also, the number of bytes belonging to large flows is a larger fraction of the overall network utilization than the number of large flows. We also conducted a similar experiment as in Section III-C, and observed a significant reduction in data transfer time. This experiment has an important implication:

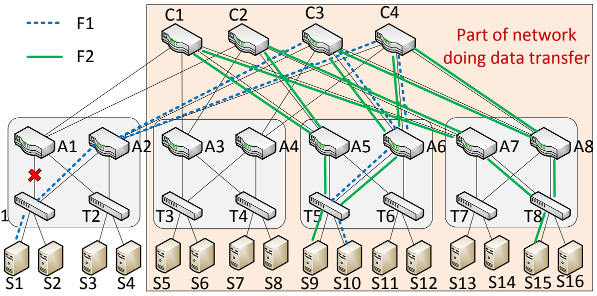


Fig. 5. Experimental setup in case of failure

RPS cannot benefit much by treating large and small flows differently. Some earlier proposals [5] work only on large flows and let ECMP handle smaller flows. Since RPS handles small flows just as well as ECMP, we can apply RPS to all flows in the network. This avoids the additional complexity of trying to identify large flows in the network.

IV. Handling Asymmetry

So far, we have investigated the behavior of packet spraying in symmetric multirooted tree topologies. But in the real world, a data center network may not be symmetric at all times. The data center may have an asymmetric topology to start with. Even in networks with symmetric topologies, asymmetries may arise due to various reasons. For instance, a failure condition (link/switch failure or link degradation) can result in an asymmetric topology. Under the above scenarios, different paths between a pair of end hosts in the network may see different levels of congestion.

In Section III-F1, we showed that the queue length differential is low in a symmetric network topology. However, the queue length differential can be significant in an assymmetric network topology due to the absence of equivalence classes. Below, we first show how flows suffer from this queue length differential. We then show how we can force queue lengths to be almost equal using existing techniques that prevent queues from growing large.

A. Problem Illustration

In this section, we use a very simple setup to demonstrate how asymmetries can impact RPS. We consider two scenarios (failures and mixture of routing strategies) which we believe are common in data centers.

Two Flow Experiment: We describe our experimental results obtained using the testbed with an oversubscription ratio of 4:1. Results for the 1:1 and 8:1 are similar and are hence skipped for brevity. To illustrate the problem, we consider only 2 flows in the network: flow F1 from S1 to S10 and and flow F2 from S15 to S9 as shown in Figure 5. For this experiment, ignore the shaded box indicating data transfer. Both flows use RPS and last for the entire duration of the experiment. As expected, F1 and F2 observe a throughput of 407Mbps. Now, we fail the link between T1 and A1 which lies along the path to F1's destination. So, while RPS is able to evenly spray F2's packets over all four paths to S9, F1's packets are sprayed over the remaining two paths to S10. This ensures F2's packets will see higher queue lengths along paths shared with F1 and lower queue lengths along other paths. As a result, F2's throughput drops from 407Mbps to 155Mbps.

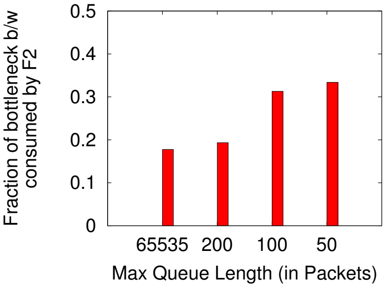


Fig. 6. RPS performance with a link failure.

B. Key Observation

We first demonstrate that the drop in throughput is indeed due to difference in queue lengths and reducing it alleviates the problem. To do so, in the two-flow experiment setup described above, we statically limit the buffer size at the output buffer of all ports in the network. When the output buffer is unrestricted, the queue length differential reaches more that 300 packets. But, limiting the output buffer limits the queue length differential too. However, it also causes a drop in throughput due to lower link utilization. To find out throughput loss due to queue length differential, we observe the fraction of the bandwidth that F2 receives at the bottleneck link. A bottleneck link is always one of the links shared by both flows. So, F2 should always get 50% of the bottleneck bandwidth. However, as Figure 6 shows, when the queue length is unrestricted (corresponding to the bar at 65535), F2 consumes just 17% of the bandwidth. Reducing buffer sizes reduces the queue length difference between the paths that carry F2's packets. As a result, it is able to sustain a higher throughput, reaching 35% when buffer sizes are restricted to 50 packets. This demonstrates that the higher the queue length differential, the lower the throughput.

C. A Practical Solution: Keeping Queue Lengths Equal

The queue length differential due to assymetries is the main factor that impacts RPS performance, but we observe that the extent of this impact is very much dependent on the maximum size a queue is allowed to grow. Modern data center operators try to keep queue lengths to the minimum to keep end-to-end latencies low and predictable, which helps RPS. Standard active queue management techniques like RED and numerous newer solutions [17], [10], [11] can be used to achieve this. Later in this section, we show how RED improves the performance of RPS in the presence of link failures. We were not able to test out RPS with newer solutions since they are still emerging.

RED probabilistically drops packets as soon as the queue length crosses some threshold. Setting the threshold too low results in being aggressive in dropping packets, and hence, queue length is kept relatively low. However, the total throughput and utilization is reduced as well. We also introduce a new variant of RED that limits the ill-effects of RED while still reducing queue length differentials. We call this variant Selective-RED (SRED). SRED *selectively* enables RED only for flows that induce a queue length differential. These are flows which do not use all the multiple paths (like F1 in the two flow experiment) because of link failures or otherwise. Intuitively, restricting the queue length share of these flows should reduce the queue length differential. Packets of flows using all the paths (like F2) continue to use droptail since these packets do not contribute to any queue length differentials.

We envision implementing SRED using packet marking and a topology aware centralized fault manager. When a link fails, the centralized fault manager configures end hosts or ToR routers to mark all packets of flows affected by that failure. Marking can be done using TOS bits in the IP header. Downstream routers and employ RED only on marked packets, thus emulating SRED. Other packets are queued and dropped using droptail policy; this limits the ill effects of RED to only those queues which induce queue imbalances in the network. Note that, while centralized controllers like Hedera [5] need to respond to new flow arrivals, the fault manager responds only to topology changes. Hence, it can scale well to larger networks.

SRED requires changes to switches however. Logically, each output port will need to maintain 2 queues, one using droptail for unmarked packets, while the other using RED for marked packets. We can of course implement SRED using a single physical queue itself; unmarked packets are inserted if space is available while marked packets are queued probabilistically using RED. Only the number of marked packets in the queue are used to calculate the average queue length used by the RED algorithm. We now evaluate this idea but a detailed analysis of SRED is left for future work.

1) Two Flow Experiment: We repeat the above experiment with RED (threshold$_{max}$ = 20, threshold$_{min}$ = 10, p$_{max}$ = 0.1) enabled at all switches. We observe the throughput of flow F2 under two scenarios: RED is applied to both flows and SRED (RED applied only to F1's packets). We want to show that the reduced throughput experienced by F2 is entirely due to the unequal queue lengths induced by F1's packets. Hence, limiting F1 in the routers' queues should be sufficient to restore F2's throughput. As seen in Figure 7(a), F2's throughput falls from 417Mbps in the ideal case to less than 200Mbps when packets from flow F1 are spread over 2 (out of 4) paths. This is expected as F1 creates high queue length differential for packets of flow F2 which are spread over all the 4 paths.

However, when we restrict the queue length of the switches using RED, the throughput of F2 increases (to 360 Mbps) resulting from lesser queue length differential. It is still low (compared to ideal) as we are limiting the link utilization by limiting the total queue length. Under SRED, we can clearly observe that F2 gets close to ideal throughput. When using 2 paths under SRED, flow F1 gets a throughput (not shown in figure) close to 195 Mbps, almost half of the ideal. This is by design since we believe it is acceptable for flows that are directly impacted by the failure to suffer throughput loss, but we want to ensure that other flows not directly impacted by the failure continue unaffected.

2) Permutation Matrix under Failure: We repeat the experiment with a permutation matrix on a 4:1 oversubscribed topology. As before, we fail the link between A1 and T1. Figure 7(b) shows the average throughput of all the flows which are not affected by the failure and spray their packets on all four paths. We also compare their average throughput with that in a topology without failures. With RPS over droptail, the mean throughput of these flows almost halves as compared to that without failure. RPS in presence of RED reduces the average throughput of these flows even in the absence of failures (due to lower link utilization). But, when a failure happens, the mean throughput is not affected by a lot (changes by less than 10%) due to limits on queue length differential.

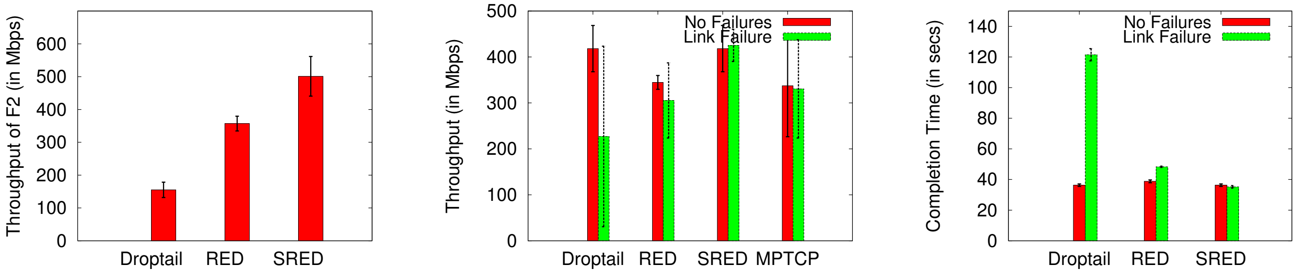
SRED is exactly like droptail when there are no failures in the network (since no flow is subjected to RED). In case of failures, flows affected by failure (using only 2 out of 4 paths in our case) are handled using RED. These flows get an average throughput of 156Mbps (not shown in graph). Other flows are not affected by the failure at all and they continue to achieve a high share of throughput. The figure clearly shows that the failure has a negligible impact of the average throughput of these flows. As expected, MPTCP remains almost unaffected by the link failure.

3) Data Transfer Time under Failure: We repeat the experiment in Section III-C with unequal traffic splitting. We use the same oversubscribed fat-tree (4:1) as discussed above. However, we transfer 1GB of data between hosts in 3 pods as shown in Figure 5. We also inject flow F1 from the 4^{th} pod to a randomly selected host in one of the 3 pods. For this experiment, ignore flow F2 shown in the figure. We plot the average time taken by the last host to complete the transfer in Figure 7(c) and compare the with and without failure cases. We average this over 10 runs and the errorbars represent the standard deviation.

A link failure in the 4^{th} pod should not affect the 1GB data transfer because its traffic does not traverse this pod. However, the flow which is injected from the 4^{th} pod creates the queue length differential as it is sprayed over two instead of four core switches (the other two core switches are inaccessible to this flow due to the failure). This imbalance greatly increases the data transfer completion time in RPS with droptail from 36 seconds to 120 seconds. In case of RED, the data transfer completion time increases marginally from 36 to 48 seconds for the same failure scenario. With SRED, a failure in the 4^{th} pod has virtually no effect on the traffic of the 3 pods. The completion time stays the same for both with and without failure scenarios.

V. Related Work

The most related to our work are those mechanisms that rely on flow-level traffic splitting such as ECMP and Hed-



(a) Throughput in 2 flow experiment

(b) Throughput with permutation matrix

(c) Completion time for data transfer

Fig. 7. RPS performance with a link failure.

era [5]. Mahout [6] is a recent scheme that uses end-host mechanisms to identify elephants, and uses flow scheduling schemes similar to Hedera. BCube [18] proposes a server-centric network architecture and source routing mechanism for selecting paths for flows. When a host needs to route a new flow, it probes multiple paths to the destination and selects the one with the highest available bandwidth. Techniques like Hedera, Mahout and BCube which select a path for a flow based on current network conditions suffer from a common problem: When network conditions change over time, the selected path may no longer be the optimal one. To overcome this problem, they periodically re-execute their path selection algorithm. VL2[1] and Monsoon[19] propose using Valiant Load Balancing (VLB) at a per-flow granularity, but they too do not split an individual flow across multiple paths.

Two research efforts propose traffic splitting at a sub-flow granularity. The first effort, MPTCP[7], splits a TCP flow into multiple sub-flows at the end hosts, which are routed over different paths in the network using ECMP. The receiving end host aggregates the TCP sub-flows and resequences packets. As we mentioned before, MP-TCP requires end-host changes which may not be feasible in all environments. It also suffers from high overhead for short flows that dominate data centers. The second effort, although in the context of the Internet, is FLARE [20]. FLARE exploits the inherent burstiness of TCP flows to break up a flow into bursts called flowlets, and route each flowlet along a different path to the destination. However, FLARE requires each router to maintain some per-flow state and estimate the latency to the destination. We did experiment with some simple variants of FLARE, such as keeping a small number of packets of a flow go through the same path. But we observed that any simple variant of FLARE does not achieve as good a throughput as our RPS, since bursts of packets may actually lead to higher queue length differentials.

VI. Conclusion

Multi-rooted tree topologies (*e.g.,* fat-trees) have emerged as the architecture of choice for many data center environments. Unfortunately, default multipath routing protocols such as ECMP can lead to significant load imbalance, resulting underutilization of network bandwidth. We show how a simple packet-level traffic splitting scheme called RPS not only leads to significantly better load balance and network utilization, but also, somewhat surprisingly, incurs little packet reordering since it exploits the symmetry in these networks. Furthermore, such schemes are of lower complexity (than MP-TCP) and readily implementable, making them an appealing alternative for data center networks. Real data centers also need to deal with failures which may disturb the symmetry, impacting the performance of RPS. We observed that by keeping queue lengths small, this impact can be minimized. We exploited this observation by proposing a simple queue management scheme called S-RED that can cope well with failures.

References

[1] A. Greenberg, J. R. Hamilton *et al.*, “VL2: A scalable and flexible data center network,” in *SIGCOMM*, 2009.

[2] S. Kandula, S. Sengupta *et al.*, “The nature of data center traffic: measurements & analysis,” in *IMC*, 2009.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” in *WREN*, 2009.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *ACM SIGCOMM*, 2008.

[5] M. Al-fares *et al.*, “Hedera: Dynamic flow scheduling for data center networks,” in *NSDI*, 2010.

[6] A. Curtis *et al.*, “Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection,” in *Infocom*, 2011.

[7] A. Ford, C. Raiciu, and M. Handley, “TCP Extensions for Multipath Operation with Multiple Addresses,” IETF, Internet-Draft, Oct. 2009.

[8] “Per packet load balancing,” http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/pplb.html.

[9] M. Laor and L. Gendel, “The effect of packet reordering in a backbone link on application throughput,” *Network, IEEE*, vol. 16, no. 5, sep 2002.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz *et al.*, “Data Center TCP (DCTCP),” in *SIGCOMM*, 2010.

[11] M. Alizadeh *et al.*, “Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center,” in *NSDI*, 2012.

[12] E. Blanton and M. Allman, “Using TCP DSACKs and SCTP Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions,” IETF, RFC 3708, Feb. 2004.

[13] “Improving the robustness of TCP to non-congestion events,” IETF RFC 4653 (Experimental), 2006.

[14] D. Zats *et al.*, “DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks,” in *ACM SIGCOMM*, Mar 2012.

[15] Sebastien Barr, “MultiPath TCP in the Linux Kernel,” https://scm.info.ucl.ac.be/trac/mptcp/wiki/install.

[16] C. Raiciu, C. Pluntke *et al.*, “Data Centre Networking with Multipath TCP,” in *ACM HotNets*, 2010.

[17] K. K. Ramakrishnan, S. Floyd, and D. Black, “The Addition of Explicit Congestion Notification (ECN) to IP,” IETF, RFC 3168, Sep. 2001.

[18] C. Guo *et al.*, “BCube: A High Performance, server-centric network architecture for modular data centers,” in *ACM SIGCOMM*, 2009.

[19] A. Greenberg *et al.*, “Towards a next generation data center architecture: scalability and commoditization,” in *ACM PRESTO*, 2008.

[20] S. Sinha, S. Kandula, and D. Katabi, “Harnessing TCPs Burstiness using Flowlet Switching,” in *ACM HotNets*, 2004.