The TCP Outcast Problem: Exposing Unfairness in Data Center Networks

Pawan Prakash, Advait Dixit, Y. Charlie Hu, Ramana Kompella
Purdue University
{*pprakash, dixit0, ychu, rkompella*}*@purdue.edu*

Abstract

In this paper, we observe that bandwidth sharing via TCP in commodity data center networks organized in multi-rooted tree topologies can lead to severe unfairness, which we term as the *TCP Outcast problem*, under many common traffic patterns. When many flows and a few flows arrive at two ports of a switch destined to one common output port, the small set of flows lose out on their throughput share significantly (almost by an order of magnitude sometimes). The Outcast problem occurs mainly in taildrop queues that commodity switches use. Using careful analysis, we discover that taildrop queues exhibit a phenomenon known as *port blackout*, where a series of packets from one port are dropped. Port blackout affects the fewer flows more significantly, as they lose more consecutive packets leading to TCP timeouts. In this paper, we show the existence of this TCP Outcast problem using a data center network testbed using real hardware under different scenarios. We then evaluate different solutions such as RED, SFQ, TCP pacing, and a new solution called *equal-length routing* to mitigate the Outcast problem.

1 Introduction

In recent years, data centers have emerged as the cornerstones of modern communication and computing infrastructure. Large-scale online services are routinely hosted in several large corporate data centers comprising upwards of 100s of thousands of machines. Similarly, with cloud computing paradigm gaining more traction, many enterprises have begun moving some of their applications to the public cloud platforms (*e.g.*, Amazon EC2) hosted in large data centers. In order to take advantage of the economies of scale, these data centers typically host many different classes of applications that are independently owned and operated by completely different entities—either different customers or different divisions within the same organization.

While resources such as CPU and memory are strictly sliced across these different tenants, network resources are still largely shared in a *laissez-faire* manner, with TCP flows competing against each other for their fair share of the bandwidth. Ideally, TCP should achieve *true fairness* (also known as RTT fairness in the literature), where each flow obtains equal share of the bottleneck link bandwidth. However, given TCP was designed to achieve long-term throughput fairness in the Internet, today's data center networks inherit TCP's *RTT bias*, *i.e.*, when different flows with different RTTs share a given bottleneck link, TCP's throughput is inversely proportional to the RTT [20]. Hence, low-RTT flows will get a higher share of the bandwidth than high-RTT flows.

In this paper, we observe that in many common data center traffic scenarios, even the conservative notion of fairness with the RTT bias does not hold true. In particular, we make the surprising observation that in a multi-rooted tree topology, when a large number of flows and and a small set of flows arrive at different input ports of a switch and destined to a common output port, the small set of flows obtain significantly lower throughput than the large set. This observation, which we term as the *TCP Outcast problem*, is surprising since when it happens in data center networks, the small set of flows typically have lower RTTs than the large set of flows and hence, according to conventional wisdom, should achieve higher throughput. Instead, we observe an *inverse RTT bias*, where low-RTT flows receive much lower throughput and become 'outcast'ed from the high-RTT ones.

The TCP Outcast problem occurs when two conditions are met: First, the network comprises of commodity switches that employ the simple taildrop queuing discipline. This condition is easily met as today's data center networks typically use low- to mid-end commodity switches employing taildrop queues at lower levels of the network hierarchy. Second, a large set of flows and a small set of flows arriving at two different input ports compete for a bottleneck output port at a switch. Interestingly, this condition also happens often in data center networks due to the nature of multi-rooted tree topologies and many-to-one traffic patterns of popular data center applications such as MapReduce [8] and Partition/Aggregate [3]. In particular, from any receiver node's perspective, the number of sender nodes that are $2n$ routing hops (n to go up the tree and n to come down) away grows exponentially, (*e.g.*, in a simple binary tree, the number grows as $2^n - 2^{n-1}$). Thus, if we assume we can place map/reduce tasks at arbitrary nodes in the data center, it is likely to have disproportionate numbers of incoming flows to different input ports of switches near the receiver node. When the above two conditions are met, we observe that the flows in the smaller set end up receiving much lower per-flow throughput than the flows in the other set—almost an *order of magnitude* smaller in many cases. We observed this effect in both real testbeds comprising commodity hardware switches and in simulations.

The reason for the existence of the TCP Outcast problem can be attributed mainly to a phenomenon we call *port blackout* that occurs in taildrop queues. Typically, when a burst of packets arrive at two ports that are both draining to an output port, the taildrop queuing discipline leads to a short-term blackout where one of the ports loses a series of incoming packets compared to the other. This behavior can affect either of the ports; there is no significant bias against any of them. Now, if one of the ports consists of a few flows, while the other has many flows (see Section 2), the series of packet drops affect the set of few flows since each of them can potentially lose the tail of an entire congestion window resulting in a timeout. TCP timeouts are quite catastrophic since the TCP sender will start the window back from one and it takes a long time to grow the congestion window back.

It is hard not to observe parallels with the other well-documented problem with TCP in data center networks, known as the TCP Incast problem [23]. The Incast problem was observed first in the context of storage networks where a request to a disk block led to several storage servers connected to the same top-of-rack (ToR) switch to send a synchronized burst of packets, overflowing the limited buffer typically found in commodity data center switches causing packet loss and TCP timeouts. But the key problem there was under-utilization of the capacity since it took a long time for a given TCP connection to recover as the default retransmission timeout was rather large. In contrast, the TCP Outcast problem exposes unfairness and, unlike Incast, it requires neither competing flows to be synchronized, nor the bottleneck to be at the ToR switch. One major implication of the Outcast problem, similar to the Incast problem, is the need to design protocols that minimize the usage of switch packet buffers. Designing protocols that take advantage of additional information in the data center setting to reduce buffer consumption in the common case can result in a range of benefits, including reducing the impact of Outcast. DCTCP [3] is one such effort but there may be related efforts (*e.g.*, RCP [9]) that may also be beneficial.

One key question that remains is why one needs to worry about the unfairness across flows within the data center. There are several reasons for this: (1) In a multi-tenant cloud environment with no per-entity slicing of network resources, some customers may gain unfair advantage while other customers may get poor performance even though both pay the same price to access the network. (2) Even within a customer's slice, unfairness can affect the customer's applications significantly: In the reduce phase of map-reduce applications (*e.g.*, sort), a reduce node fetches data from many map tasks and combines the partial results (*e.g.*, using merge sort). If some connections are slower than the others, the progress of the reduce task is stalled, resulting in significantly increased memory requirement in addition to slowing down the overall application progress. (3) TCP is still the most basic light-weight solution that provides some form of fairness in a shared network fabric. If this solution itself is broken, almost all existing assumptions about any level of fairness in the network are in serious jeopardy.

Our *main contribution* in this paper is to show the existence of the TCP Outcast problem under many different traffic patterns, with different numbers of senders and bottleneck locations, different switch buffer sizes, and different TCP variants such as MP-TCP [21] and TCP Cubic. We carefully isolate the main reason for the existence of the TCP Outcast problem using simulations as well as with traces collected at an intermediate switch of a testbed. We further investigate a set of practical solutions that can deal with the Outcast problem. First, we evaluate two router-based approaches—stochastic fair queuing (SFQ) that solves this problem to a large extent, and RED, which still provides only conservative notion of TCP fairness, *i.e.*, with RTT bias. Second, we evaluate an end-host based approach, TCP pacing, and show that pacing can help reduce but does not eliminate the Outcast problem completely.

TCP's congestion control was originally designed for the "wild" Internet environment where flows exhibiting a diverse range of RTTs may compete at congested links. As such, the RTT bias in TCP is considered a reasonable compromise between the level of fairness and design complexity and stability. In contrast, data centers present a tightly maintained and easily regulated environment which makes it feasible to expect a stricter notion of fairness, *i.e.*, true fairness. First, many network topologies (*e.g.*, multi-rooted trees, VL2 [13]) exhibit certain symmetry, which limits the flows to a small number of possible distances. Second, newly proposed topologies such as fat-trees that achieve full bisection bandwidth make shortest-path routing a less stringent requirement.

Motivated by the above reasoning, we propose and evaluate a simple new routing technique called *equal-length routing* that essentially side-steps shortest-path routing and makes all paths equal length. This simple counter-intuitive approach promotes better mixing of traffic reducing the impact of port blackouts. The technique effectively achieves true fairness, *i.e.*, equal throughput for competing flows sharing a congested link anywhere in the network since the flow RTTs are also balanced. The obvious downside to this approach is that it results in wasting resources near the core of the network. For certain topologies such as the fat-tree that already provide full bisection bandwidth, this may be alright since capacity is anyway provisioned. For data center networks with over-subscription, this approach will not be suitable; it may be better to employ techniques such as SFQ queuing if true fairness is desired.

2 Unfairness in Data Center Networks

In this section, we first provide a brief overview of today's data center networks. We then discuss our main observation in today's commodity data center networks, namely the TCP Outcast problem, which relates to unfair sharing of available link capacity across different TCP flows.

2.1 Data Center Network Design

The key goal of any data center network is to provide rich connectivity between servers so that networked applications can run efficiently. For full flexibility, it is desirable to build a data center network that can achieve full bisection bandwidth, so that any server can talk to any server at the full line rate. A lot of recent research (*e.g.*, fat-tree [2], VL2 [13]) has focused on building such full bisection bandwidth data center networks out of commodity switches. Most practical data center topologies are largely in the form of multi-rooted multi-level trees, where servers form the leaves of the tree are connected through switches at various levels—top-of-rack (ToR) switches at level 1, aggregation switches at level 2 and finally, core switches at level 3. Such topologies provide the necessary rich connectivity by providing several paths with plenty of bandwidth between server pairs.

Table 1: List of some 48-port COTS switches

48-port Switches	Congestion Avoidance
HP/3Com E5500G	Taildrop
Juniper EX4200	Taildrop
Brocade FastIron GS series	Taildrop
Cisco Catalyst 3750-E	Weighted Taildrop
Cisco Nexus 5000	Taildrop

Data center networks today are largely built out of commodity off-the-shelf (COTS) switches, primarily to keep the costs low. While these switches offer full line-rate switching capabilities, several features found in high-end routers are often missing. In particular, they typically have shallow packet buffers and contain small forwarding tables among other such deficiencies. In addition, they also typically implement simple queueing disciplines such as taildrop. In Table 1, we can see that almost all the commodity switches that are produced by popular vendors employ the taildrop (or variants of taildrop) queue management policy.

The choice transport protocol in most data centers today is TCP, mainly because it is a three-decade old mature protocol that is generally well-understood by systems practitioners and developers. Two aspects of TCP are generally taken for granted: First, TCP utilizes the network as effectively as possible, and hence is *work-conserving*; if there is spare bandwidth in the network, TCP will try to utilize it. Second, TCP is a *fair* protocol; if there are multiple flows traversing a bottleneck link, they share the available bottleneck capacity in a fair manner. These two aspects of TCP typically hold true in both wide-area as well as local-area networks.

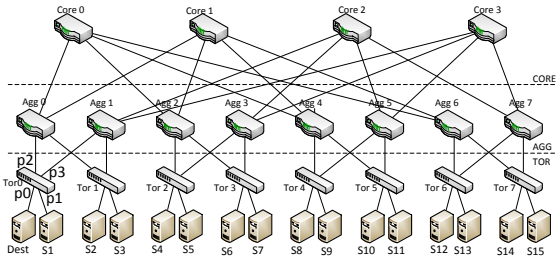


Figure 1: Data Center Topology

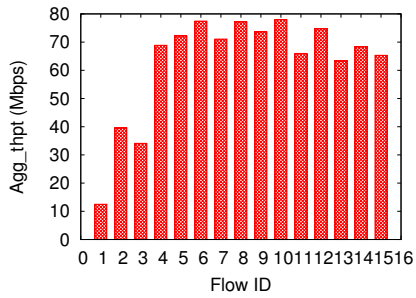
Unfortunately, certain data center applications create pathological conditions for TCP, causing these seemingly taken-for-granted aspects of TCP to fail. In cluster file systems [5, 25], for instance, clients send parallel reads to dozens of nodes, and all replies need to arrive before the client can proceed further—exhibiting a barrier-synchronized many-to-one communication pattern. When synchronized senders send data in parallel in a high-bandwidth low-latency network, the switch buffers can quickly overflow, leading to a series of packet drops which cause TCP senders to go into the timeout phase. Even when the capacity opens up, still some senders are stuck for a long time in the timeout phase, causing severe underutilization of the link capacity. This observation is famously termed as the *TCP Incast problem* first coined by Nagle *et al.* in [18]. The Incast problem has ever since generated a lot of interest from researchers—to study and understand the problem in greater depth [7, 27] as well as propose solutions to alleviate it [23, 26].

In this paper, we focus on a different problem that relates to the second aspect of TCP that is taken for granted, namely, TCP fairness.

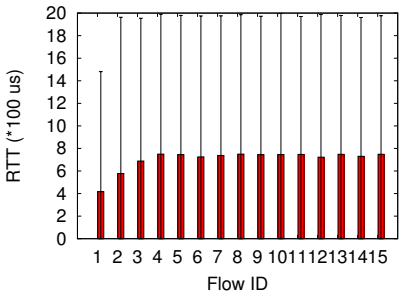
2.2 The TCP Outcast Problem

Consider a data center network that is organized in the form of an example (k=4) fat-tree topology as shown in Figure 1 proposed in literature [2]. (While we use the fat-tree topology here for illustration, this problem is not specific to fat-trees and is found in other topologies as well.) Recall that in a fat-tree topology, all links are of the same capacity (assume 1Gbps in this case). Now suppose there are 15 TCP flows, f_i ($i = 1...15$) from sender S_i to $Dest$. In this case, the bottleneck link is the last-hop link from $ToR0$ to $Dest$. All these flows need not start simultaneously, but we mainly consider the portion of time when all the 15 flows are active.

We built a prototype data center network using NetFPGA-based 4-port switches (discussed in more de-



(a) Aggregate throughput



(b) Average RTT

Figure 2: TCP Outcast problem

tail in Section 3.1) and experimented with the traffic pattern discussed above. Figure 2 shows that the per-flow aggregate throughput obtained by these 15 flows exhibit a form of somewhat surprising unfairness: Flow f_1 which has the shortest RTT achieves significantly lesser aggregate throughput than any of f_4-f_{15}—almost 7-8$\times$ lower. Flows f_2 and f_3 also achieve lesser throughput (about 2$\times$) than f_4-f_{15}. We observe this general trend under many different traffic patterns, and at different locations of the bottleneck link (discussed in detail in Section 3.2), although the degree of unfairness varies across scenarios. (Note that we also found evidence of unfairness by conducting a limited number of experiments with another institution's data center testbed consisting of commodity HP-ProCurve switches organized in a $k = 4$ fat-tree topology.)

We call this gross unfairness in the throughput achieved by different flows that share a given bottleneck link in data center networks employing commodity switches with the taildrop policy as the *TCP Outcast problem*. Here, flow f_1, and to some extent f_2 and f_3, are 'outcast'ed by the swarm of other flows $f_4 - f_{15}$. Although $f_1 - f_3$ differ in their distances (in terms of number of hops between the sender and the receiver) compared to flows $f_4 - f_{15}$, this does not alone explain the Outcast problem. If any, since f_1 has a short distance of only 2 links, its RTT should be much smaller than $f_4 - f_{15}$ which traverse 6 links. This is confirmed in Figure 2(b), which shows the average RTT over time along with the max/min values. According to TCP analysis [20], throughput is inversely proportional to the RTT, which suggests f_1 should obtain higher throughput than any of $f_4 - f15$. However, the Outcast problem exhibits exactly the opposite behavior.

The reason for this counter-intuitive result is two fold: First, taildrop queuing leads to an occasional "*port blackout*" where a series of back-to-back incoming packets to one port are dropped. Note that we deliberately use the term blackout to differentiate from a different phenomenon called 'lockout' that researchers have associated with taildrop queues in the past [10, 6]. The well-known lockout problem results from global synchronization of many TCP senders, where several senders transmit synchronized bursts, and flows that manage to transmit ever so slightly ahead of the rest manage to get their packets through but not the others, leading to unfairness. In contrast, the blackout problem we allude to in this paper occurs when two input ports drain into one output port, with both input ports containing a burst of back-to-back packets. In this case, one of the ports may get lucky while the other may incur a series of packet drops, leading to a temporary blackout for that port. Second, if one of the input ports contains fewer flows than the other, the temporary port blackout has a catastrophic impact on that flow, since an entire tail of the congestion window could be lost, leading to TCP timeouts. We conduct a detailed analysis of the queueing behavior to elaborate on these reasons in Section 4.

2.3 Conditions for TCP Outcast

To summarize, the two conditions for the Outcast problem to occur are as follows:

- (C1) The network consist of COTS switches that use the taildrop queue management discipline.
- (C2) A large set of flows and a small set of flows arriving at two different input ports compete for a bottleneck output port at a switch.

Unfortunately, today's data centers create a perfect storm for the Outcast problem to happen. First, as we mentioned before, for cost reasons, most data center networks use COTS switches (Table 1) which use the taildrop queue management discipline, which exhibits the port blackout behavior.

Second, it is not at all uncommon to have a large set of flows and a small set of flows arriving at different input ports of a switch and compete for a common output port, due to the nature of multi-rooted tree topologies commonly seen in data center networks and typical traffic patterns in popular data center applications such as MapReduce [8] and Partition/Aggregate [3]. For instance, in large MapReduce applications, the many map and reduce tasks are assigned to workers that span a large portion of the data center network. When a reduce

task initiates multiple TCP connections to different map tasks, the sources of the flows are likely to reside in different parts of the network. As a result, in any tree-like topology, it is very likely that these flow sources result in disproportionate numbers of flows arriving at different input ports of a switch near the receiver. This is because from any leaf node's perspective, the number of nodes that are $2n$ hops away grows exponentially. For example, in a simple binary tree, the number grows as $2^n - 2^{n-1}$.

3 Characterizing Unfairness

In this section, we present experimental results that demonstrate the throughput unfairness symptom of the TCP Outcast problem. We extensively vary all the relevant parameters (*e.g.*, traffic pattern, TCP parameters, buffer size, queueing model) to extensively characterize the TCP Outcast problem under different conditions.

3.1 Experimental Setup

Our data center testbed is configured in a $k = 4$ fat-tree as shown in Figure 1, with 16 servers at the leaf-level and 20 servers acting as the switches. The servers are running CentOS 5.5 with Linux kernel 2.6.18. Each switch server is equipped with NetFPGA boards acting as a 4-port Gigabit switch and running OpenFlow for controlling the routing. Each NetFPGA card has a packet buffer of 16KB per port, and all the queues in the NetFPGA switches use taildrop queuing.

Packets are routed in the fat-tree by statically configuring the shortest route to every destination. When there are multiple shortest paths to a destination, the static route at each switch is configured based on the trailing bits of the destination address. For example, at ToR switches, a packet destined to a server connected to a different ToR switch is forwarded to one of the aggregate switches as decided by the last bit of the destination address: the right aggregate switch if the bit is 0, and the left if the bit is 1. Similarly, at the aggregate switch, packets coming from ToR switches that are destined to a different pod are forwarded to one of the core switches. The aggregate switch selects the core switch based on the second last bit of the destination address. Our routing scheme is in principle similar to the Portland architecture [19].

To emulate condition C2 in Section 2.3, we used a simple many-to-one traffic pattern, mainly for convenience, for most of our experiments. We also used a more general pattern (Section 3.2.4) to show many-to-one pattern is not a necessity.

The many-to-one traffic pattern naturally leads to sources placed at different distances. For instance, consider the fat-tree topology in Figure 1 (same is true with other multi-rooted tree topologies such as VL2). From the perspective of any receiver, the senders belong to 3 classes—senders under the same ToR (2-hop), senders in the same pod (4-hop), and senders in different pods (6-hop). The senders belonging to a particular class are at the same distance from the receiver. We conducted experiments under different scenarios with 15 senders ($S1$-$S15$) as depicted in Figure 1, each of which initiates one or more TCP flows to a single receiver (labeled $Dest$), and measured the TCP throughput share achieved by different flows. Note that in all experiments, unless otherwise noted, we only consider the throughput share obtained by individual flows when *all* flows are active.

We used the default values for all TCP parameters except the minimum round-trip timeout (minRTO), which we set to 2 milliseconds to overcome the adverse effects of TCP Incast problem [23]. We disabled TCP segmentation offload since that would have further increased the burstiness of TCP traffic and probably increased unfairness. We experimented with different TCP variants (Reno, NewReno, BIC, CUBIC) and obtained similar results. Due to page limit, we present results under TCP BIC and CUBIC for experiments conducted on our testbed and under NewReno for ns-2 simulations.

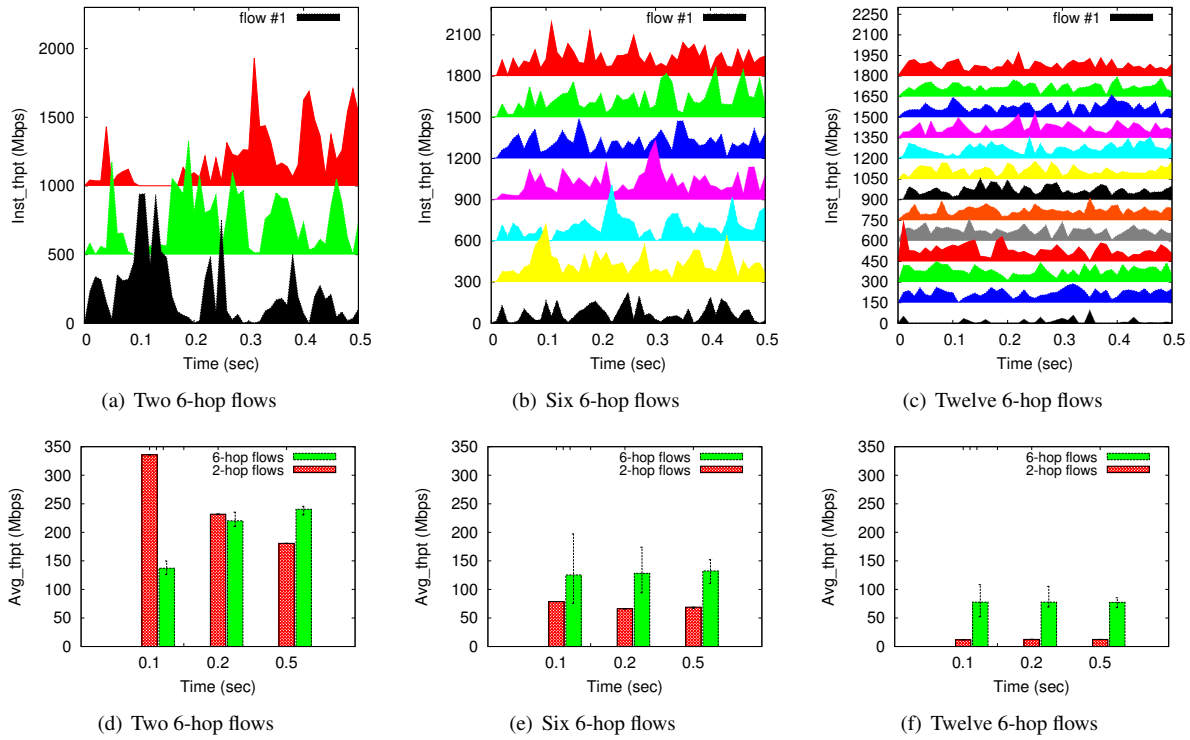
3.2 Throughput Unfairness Results

We start with the simplest base case where one flow from each sender is initiated to the destination, and show that it results in the TCP Outcast problem. We then show that the problem exists even if (1) there is disparity in the number of competing flows arriving at different input ports; (2) flows do not start simultaneously, unlike in the incast [26] problem; (3) the bottleneck link is in the core of the network; and (4) there is background traffic sharing the bottleneck link.

3.2.1 Case I – Different Flow Proportions

In this case, multiple long-distance flows, one from each sender node six hops away, arriving at port $p2$ of $ToR0$, and one flow from sender $S1$ (flow 1), arriving at port $p1$ of $ToR0$, compete for output port $p0$. Figures 3(a)-3(c) show the instantaneous throughput achieved by individual flows within the first 0.5 seconds when there are two, six, and twelve 6-hop flows, respectively. The y-axis for each flow is offset by 500, 300, and 150 Mbps respectively so that the instantaneous throughput per flow is clearly visible. Figures 3(d)-3(f) show the corresponding aggregate throughput of flow 1 and the average aggregated throughput of all the 6-hop flows within the first 0.1, 0.2, and 0.5 seconds.

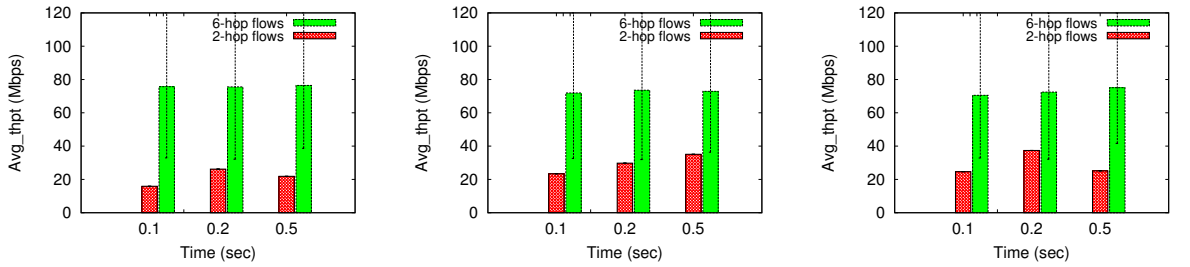
These figures show that with two 6-hop flows, flow 1 gets higher than average share in the beginning, but is occasionally starved by other flows after 0.1 seconds (due to reasons explained in Section 4). Figures 3(a)-3(b) show that as we increase the number of long-distance flows to six and twelve, flow 1's throughput becomes increasingly worse and practically starves compared to the



(a) Two 6-hop flows (b) Six 6-hop flows (c) Twelve 6-hop flows

(d) Two 6-hop flows (e) Six 6-hop flows (f) Twelve 6-hop flows

Figure 3: Instantaneous and average throughput in case of one 2-hop flow and multiple 6-hop flows



(a) 10 2-hop, 120 6-hop (b) 20 2-hop, 240 6-hop (c) 30 2-hop, 360 6-hop

Figure 4: Average throughput per host in case of fixed proportion (1:12) of 2-hop and 6-hop flows

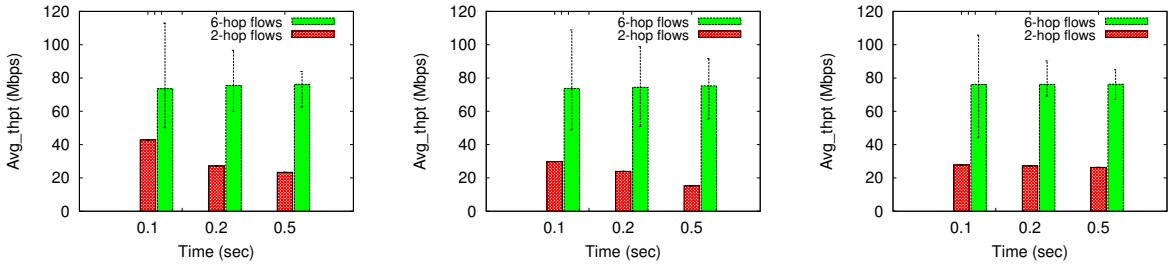
rest. Overall, the aggregate throughput of flow 1 is 2× and 7× worse than the average throughput of the six and twelve 6-hop flows, respectively.

3.2.2 Case II – Multiple Flows

Figure 3(f) shows the throughput unfairness when one 2-hop flow at one input port competes with twelve 6-hop flows at the other input port for access to the output port at switch $ToR0$. To explore whether the problem persists even with a larger numbers of flows competing, we vary the number of flows per host (10, 20 and 30) while keeping the ratio of flows arriving at the two input ports the same (1:12) as in Figure 3(f). Figure 4 shows when 10 flows arrive at port $p1$ and 120 flows arrive at port $p2$, the average throughput of the 2-hop flows is 3× worse than that of the 120 6-hop flows. Note that the y-axis in the figure is the average throughput on a per-host basis (*i.e.*, sum of all flows that originate at the host); since same number of flows start at all the nodes, the individual per-flow throughput is just scaled down by the appropriate number of flows per host. We see that the similar unfairness persists even in this scenario.

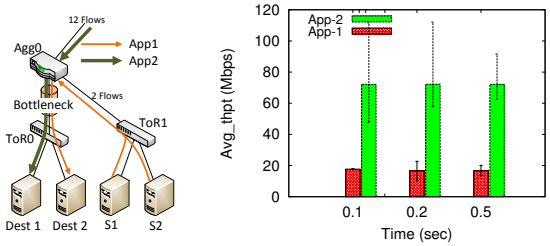
3.2.3 Case III – Unsynchronized Flows

One could perhaps conjecture that throughput unfairness observed so far may be because all flows are starting at the same time, similar to the TCP Incast problem. In order to verify whether this is a requirement, we experiment with staggered flow arrivals. We again consider the same thirteen flows (one 2-hop and twelve 6-hop flows) as in Figure 3(f), but instead of starting all flows at the same time, we stagger the start time of the 2-hop flow to be 100ms before, 100ms after, and 200ms after the 6-hop flows. In Figure 5, we can observe that the 2-



(a) Early start of 100ms for 2-hop flow (b) Delayed start of 100ms for 2-hop flow (c) Delayed start of 200ms for 2-hop flow

Figure 5: Average throughput in case of different start times for the 2-hop flow. Time on x-axis is relative to when all flows are active.



(a) Scenario (b) Throughput

Figure 6: Two applications sharing a network bottleneck

hop flow obtains significantly lower throughput even if it starts 100ms before the 6-hop flows, which allows it sufficient time to ramp up to a larger window. It appears that once the 6-hop flows begin, the 2-hop flows experience starvation as in Figure 3(f). In Figure 5(a), we can clearly observe the throughput disparity between the 2-hop flow and 6-hop flows increases with time as the impact of the initial gains due to initial higher window reduces over time.

3.2.4 Case IV – Distinct Flow End-points

So far, we have mainly experimented with the many-to-one traffic pattern. In order to show that this is not fundamental to TCP Outcast, we use a different traffic pattern where the 2-hop and 6-hop flows have different end-hosts, as show in 6(a). Here, the bottleneck link lies between the aggregate switch and the ToR switch, which is again different from TCP Incast. Figure 6(b) shows that unfairness persists, confirming that TCP Outcast can happen in aggregate switches and does not rely on many-to-one communication. Further, as shown in the previous section, these flows need not be synchronized. Together, these non-requirements significantly increase the likelihood of observing the TCP Outcast problem in production data center environments.

3.2.5 Case V – Background Traffic

Since many different applications may share the network fabric in data centers, in this experiment, we study if background traffic sharing the bottleneck switch can eliminate or at least mitigate the Outcast problem. We generated background traffic at each node similar to the experiments in [13], by injecting flows between random pairs of servers that follow a probability distribution of flow sizes (inferred from [13]). The network bandwidth consumed by the background traffic is controlled by the flow inter-arrival time. Specifically, if we want B background traffic, given a mean flow size F, the mean flow inter-arrival time is set as F/B, and we create an exponential distribution of flow inter-arrival time with the calculated mean. In this experiment, we also generated two 4-hop flows to confirm that there is nothing specific about 2-hop and 6-hop flows contending.

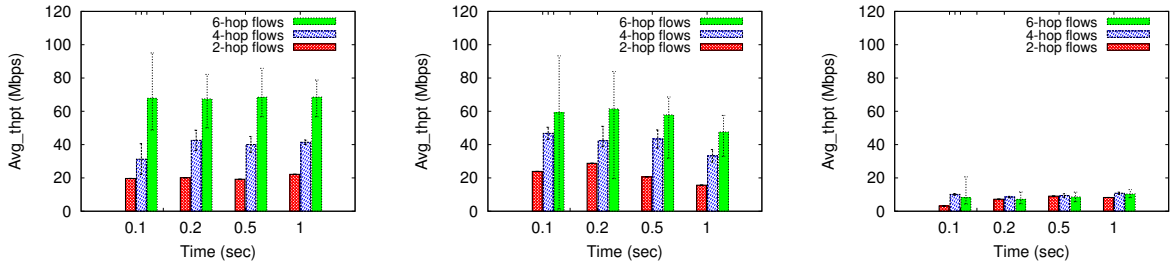
Figure 7 depicts the resulting average throughput for one 2-hop flow, two 4-hop flows, and twelve 6-hop flows under different amounts of background traffic. Clearly the presence of background traffic affects the average throughput of every flow. But the extent of unfairness is not mitigated by the background traffic completely. In particular, the gap between the throughput of 2-hop flows and 6-hop flows remain $4\times$ and $2.5\times$ under background traffic of 10%, 20% of the bottleneck link capacity (1 Gbps) respectively. Only when the background traffic reaches 50% of the bottleneck link capacity, the unfairness seems to taper off, that too after 0.2 seconds.

3.2.6 Case VI – Other Experiment Scenarios

We also vary other parameters such as buffer sizes and RTT on the TCP Outcast problem.

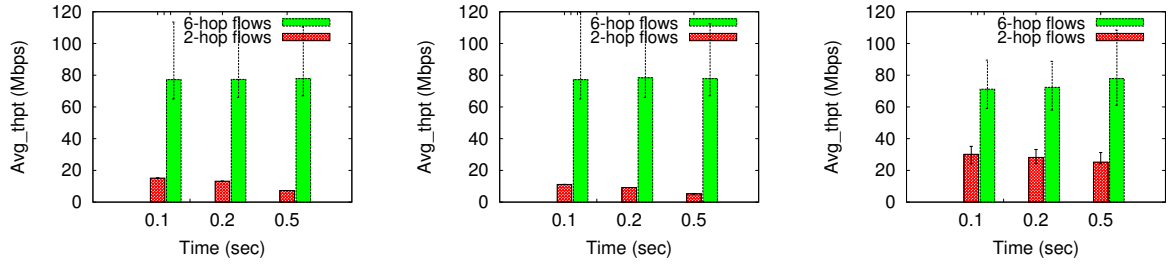
Buffer size. We found that increasing the buffer size does not have a significant effect on the TCP Outcast problem. A larger buffer size means that it would take longer for the queue to fill up and for port blackout to happen but it eventually happens. In our testbed, we have tried with buffer sizes of 16KB and 512KB and found that the unfairness still persists. Using ns-2 simulations, we simulated different buffer sizes of 32, 64, 128KB, and found similar results.

RTT. We simulate twelve flows from one 4-hop server and one flow from the other 4-hop sender (hence all flows have the same RTTs). We observed that the TCP Out-



(a) 100 Mbps background traffic (b) 200 Mbps background traffic (c) 500 Mbps background traffic

Figure 7: Average throughput of 2-hop, 4-hop, and 6-hop flows under background traffic



(a) TCP CUBIC without SACK (b) TCP CUBIC with SACK (c) MPTCP

Figure 8: Average throughput of 2-hop, and 6-hop flows under CUBIC, SACK, and MPTCP.

cast problem still exists and the single flow is starved for throughput. Thus, it appears the number of flows on the ports, as opposed to the RTT differential, impacts the unfairness.

Multiple ports contending. In the test bed, we modified routing so that we have flows coming on 3 input ports and going to one output port. Even in this case, the TCP Outcast problem is present. In ns-2, we have experimented with even more input ports (*e.g.*, 6-pod, 8-pod fat-tree, VL2 [13] topology) and found that the Outcast problem exists.

CUBIC, SACK, and MPTCP. We tested the existence of Outcast problem with TCP CUBIC with and without SACK, and with MPTCP [21]. Figure 8 depicts the occurence of Outcast in all these scenarios, although MPTCP seems to reduce the extent of unfairness. Since MPTCP opens many different sub-flows corresponding to each TCP flow, this scenario is roughly equivalent to the multiple flows experiment in Figure 4.

4 Explaining Unfairness

Routers with taildrop queues have been known to suffer from the *lockout* problem, in which a set of flows experience regular packet drops while other flows do not. Floyd *et al.* [10] have demonstrated that TCP phase effects can lead to these lockouts where packets arriving at a router after certain RTTs find the queue to be full and hence are dropped. TCP phase effects were studied in the context of the Internet and RTT was the primary factor in determining which flows will suffer from lockout.

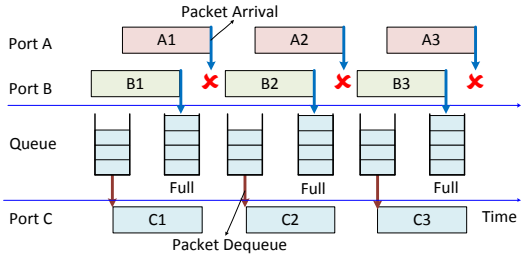
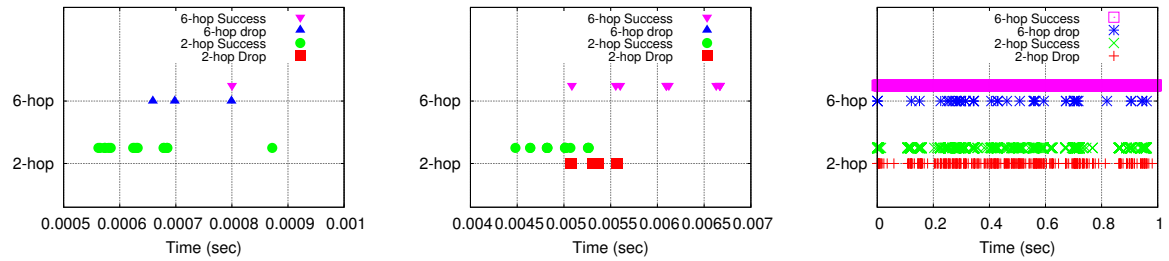


Figure 9: Timeline of port blackout

In this section, we demonstrate the existence of a different phenomenon called *port blackout* in the context of data center networks. Port blackout is defined as the phenomenon where a stream of back-to-back packets arriving on multiple input ports of a switch compete for the same output port, and packets arriving on one of the input ports are dropped while packets arriving on the other input ports are queued successfully in the output port queue. Port blackouts occurs when the switch uses taildrop queue management policy.

In the following, we explain how port blackouts can occur in data center networks. We also corroborate our observation with ns-2 simulations with configurations identical to our testbed. We then introduce a drop model using ns-2 simulation to demonstrate the effects of port blackout on TCP throughput. We end with insights into how port blackout can be prevented in data center networks.



(a) Blackout for 6-hop flows (b) Blackout for 2-hop flows (c) Packet loss and success event for 2-hop and 6-hop flows

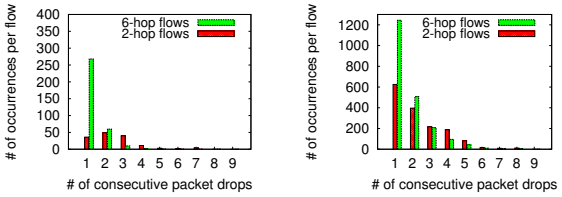
Figure 10: Blackout behavior observed in taildrop queues

4.1 Port Blackout in Data Center Testbed

Figure 9 schematically depicts the timeline of events occurring at a switch amidst a port blackout episode. A stream of packets $A1$, $A2$ and $A3$ arriving at port A and $B1$, $B2$ and $B3$ arriving at port B are competing for output port C which is full. Since most of the competing flows are long flows, their packets are of the same size, which means the time spent by each of the frames is the same on the wire. Now, since these packets arrive on two different ports, they are unlikely arriving at exactly the same time (the ports are clocked separately). However, the inter-frame spacing on the wire is the same for both ports, since there are back-to-back packets (assuming the senders are transmitting many packets) and no contention from any other source on the Ethernet cable (given switched Ethernet). Now, due to the asynchronous nature of these packet arrivals, one port may have packets slightly ahead of the others, *e.g.* in Figure 9, port B's packets arrive just slightly ahead of port A's packets.

After de-queuing a packet $C1$, the output port queue size drops to $Q-1$. Now, since packets from B arrive slightly ahead of A, packet $B1$ arrives at port B next (denoted by an arrow on the time line), finds queue size to be $Q-1$, and is successfully enqueued in the output queue making it full. Next, packet $A1$ arrives at port A, finds the queue to be full, and hence gets dropped. The above pattern of consecutive events then repeats, and $A2$ as well as $A3$ end up with the same fate as its predecessor $A1$. This synchronized chain of events among the three ports can persist for some time resulting in a sequence of packet losses from one input port, *i.e.*, that port suffers a blackout. Once the timing is distorted, either because there is a momentary gap in the sending pattern or due to some other randomness in timing, this blackout may stop. But, every so often, one of the ports may enter into this blackout phase, losing a bunch of consecutive packets. We note that either of the input ports can experience this blackout phenomenon; there is no intrinsic bias against any one port.

To investigate the existence of port blackout in our data center testbed, we collected the traffic traces close to output port $p0$ and input ports $p1$ and $p2$ of switch



(a) With 1 flow per host (b) With 20 flows per host

Figure 11: Distribution of consecutive packet drops

$ToR0$ during the experiment in Figure 3(f). The trace at port $p1$ consists of a stream of back-to-back packets from server $S1$ which is under the same $ToR0$. The trace at port $p2$ consists of packets from servers that are located outside $ToR0$. Both these streams of packets are meant to be forwarded toward output port $p0$ and hence, compete with each other. Correlating these two traces with the traffic trace at output port $p0$, we can infer the set of packets that were successfully forwarded and the set that were dropped.

Figure 10(c) shows the timeline of packets successfully sent and dropped at port $p2$ (for 6-hop flows) and port p1 (for 2-hop flows) of switch $ToR0$ during the experiment. When port blackouts happen, we can observe clusters of packet drops. To see the detailed timing of packet events during blackouts, we zoom into small time intervals. Figure 10(a) depicts a small time interval (about 500 microseconds) when port $p2$ carrying the flows from servers outside $ToR0$ experiences a port blackout, during which packets from port $p1$ are successfully sent while consecutive packets from port $p2$ are dropped. Figure 10(b) depicts a similar blackout event for port $p1$. While we highlight a single incident of port blackout here, Figure 11(a) shows the distribution of episodes with k consecutive packet losses. As we can see, the 2-hop flow experiences many more episodes of 3 and 4 consecutive packet drops than the 6-hop flows. This trend does not seem to change even with a larger number of flows per host as shown in Figure 11(b).

4.2 Port Blackout Demonstration in ns-2

While the traces above give us some insight that black-

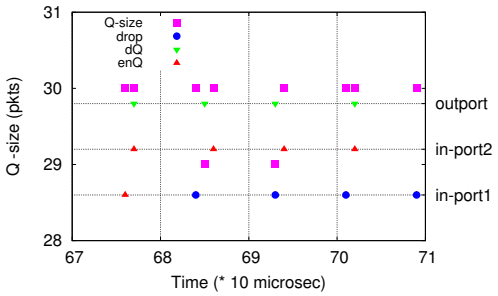


Figure 12: Queue dynamics at the ToR 0 switch

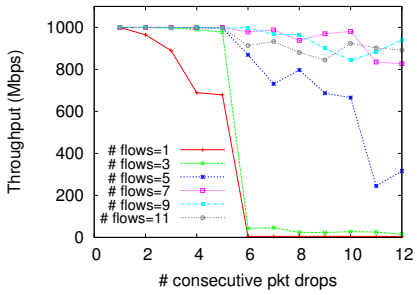


Figure 13: Effect of consecutive packet drops on TCP

outs may be happening, due to inaccuracies in timing we only get rough insights from the above trace. In order to understand this even more closely, we resort to simulations in ns-2. In simulations, we can easily observe the changing value of the dynamic output port queue size and the exact timing of how it correlates with packet enqueue and dequeue events. We simulate the same experiment, *i.e.*, fat-tree configuration and traffic pattern, as in Figure 3(c), in ns-2. Figure 12 depicts the exact timeline of different packet events, enqueue, dequeue, and drop, corresponding to the three ports.

For each port (right y-axis), packet enqueue, drop, and dequeue events are marked. The left y-axis shows the queue size at any given instant of time. The queue dynamics is shown for one of the intervals during which in-port1 is suffering from a port blackout episode. Consider the interval between 68 and 69 (*10 microsecond). First, a packet arrives at in-port1. But the queue was full (Q-size 30) at that instant as denoted by the square point. As a result this packet at in-port1 is dropped by the taildrop policy. Soon after that, a packet is dequeued from the output queue and now the queue size drops to 29. Next, a new packet arrives at in-port2, and is accepted in the queue making the queue full again. This pattern repeats and in-port1 suffers from consecutive packet drops, leading to an episode of port blackout.

4.3 Effect of Port Blackout on Throughput

We have explained the root cause for the port blackout phenomenon in previous sections using real traces collected from our data center testbed as well as using the ns-2 simulations. In this section, we present a simulation model to help us understand the impact of port blackout on the throughput of TCP flows. More specifically, we want to analyze the relationship between the number of flows on an input port (that experiences blackout) and the impact on their TCP throughput due to port blackout.

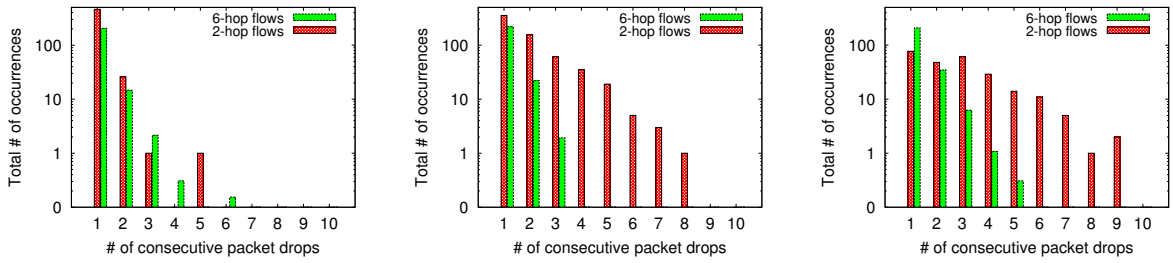
We simulate a simple topology in ns-2 consisting of a single sender node (node 1) and a single receiver node (node 2) connected via a switch. To simulate the port blackout behavior, we modified the taildrop queue at the switch to operate in two states. In the ON state, it drops all packets that it receives from node 1. In the OFF state, it does not drop any packet. The queue toggles from OFF to ON state after every k seconds, where k is chosen from an exponential distribution with a mean of 0.005 seconds, which is the approximate time period between two blackout periods we observed in our testbed. It remains in ON state for a fixed duration that corresponds to m consecutive packet drops. Note that an ON state does not necessarily correspond to m actual packet drops; it is the *time duration* in which the switch would have dropped m consecutive packets. In other words, we only drop consecutive packets if they appear back-to-back during the ON state.

Using this drop model, we study the impact of the length of ON duration on the throughput of f TCP flows from node 1. Figure 13 shows the aggregate TCP throughput (on y-axis) of f flows, as the number of consecutive packet drops m (on x-axis) varies. We observe that when there are 7 or more flows, port blackout, *i.e.* consecutive packets drops during the ON state, only affects the throughput of the flows slightly, even as m grows to 10. This is because packets dropped in the ON state are spread across the flows and each flow can recover quickly from few packet losses due to fast retransmission. However, when there are few flows, the consecutive packet drops have a catastrophic effect on their throughput because of timeouts that leads to reducing the congestion window significantly.

While it may appear from the above experiment that the Outcast problem may disappear if we have larger number of flows, Figure 4 clearly indicates that that is not true. The reason lies in the fact that if there are a larger number of flows, the duration of the blackout simply increases causing more consecutive packet drops, translating to a similar number of packet losses per flow as before. We find evidence of this effect in Figure 11(b) which shows the number of consecutive drops for 2-hop flows remains much higher than the 6-hop flows even with 20 flows per host.

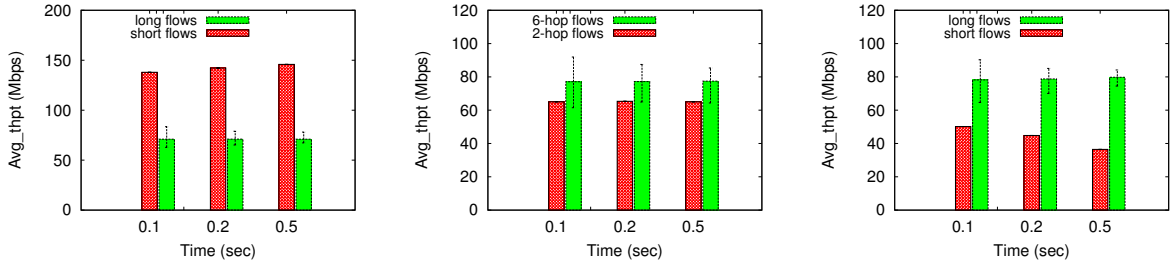
5 Mitigating Unfairness

The root cause of the TCP Outcast problem in data center networks is input port blackout at bottleneck switches



(a) RED (b) SFQ (c) TCP pacing

Figure 14: Distribution of consecutive packet drops under RED, SFQ, and TCP pacing solutions



(a) RED (b) SFQ (c) TCP pacing

Figure 15: Average throughput under RED, SFQ, and TCP pacing solutions

happening due to the taildrop policy of the output queue, which has a drastic effect on the throughput of the few flows that share the blackout input port. Hence, the key to solving the TCP Outcast problem is to distribute packet drops among all competing flows (for an output port) arriving at the switch to avoid blackout of any flow.

In this section, we study three approaches that all achieve this goal, but via rather different means. The first includes two solutions that directly get rid of the taildrop packet drop policy, by replacing it with RED or SFQ. The second, TCP pacing, tries to alleviate the burstiness of packets in each TCP flow (*i.e.*, window), and hence potentially reduces bursty packet loss for any particular flow. The third approach avoids port blackout by forcing flows with nearby senders to detour to take similar paths as flows with faraway senders so that their packets are well interleaved along the routing paths. We evaluate the effectiveness of each approach and further discuss their pros and cons in terms of implementation complexity and feasibility in data center networks.

5.1 RED

RED [11] is an active queue management policy which detects incipient congestion and randomly marks packets to avoid window synchronization. The random marking of packets essentially interleaves the packets from different input ports to be dropped and hence avoids blackout of any particular port. We simulate RED in ns-2 with the same configuration as Figure 3(f), with 12 6-hop flows and 1 2-hop flow destined to a given receiver. In our setup, we use the classical RED policy, with the minimum threshold set to 5 packets, the maximum threshold set to 15 packets, and the queue weight set to 0.002.

Figure 14(a) shows the distribution of different number of consecutive packet drops for 2-hop and 6-hop flows (since there are multiple 6-hop flows we take an average of all the twelve flows). We observe that the consecutive packet drop events are similar for 2-hop and 6-hop flows. More than 90% of packet drop events consist of a single consecutive packet loss, suggesting that blackouts are relatively uncommon, and all the flows should have achieved a fair share of TCP throughput. However, Figure 15(a) shows a difference in average throughput between 2-hop and 6-hop flows. This is explained by the well-known RTT bias that TCP exhibits; since the 2-hop flow has a lower RTT, it gets the a larger share of the throughput (TCP throughput $\sim \frac{1}{RTT \times \sqrt{droprate}}$). Thus, we can clearly see that RED queuing discipline achieves RTT bias but does not provide the true throughput fairness in data center networks.

5.2 Stochastic Fair Queuing

We next consider stochastic fair queuing (SFQ) [17], which was introduced to provide fair share of throughput to all the flows arriving at a switch irrespective of their RTTs. It divides an output buffer into buckets (the number of buckets is a tunable parameter) and the flows sharing a bucket get their share of throughput corresponding

to the bucket size. A flow can also opportunistically gain a larger share of the bandwidth if some other flow is not utilizing its allocated resources. We simulate the same experimental setup as before (twelve 6-hop and one 2-hop flow) in ns-2 with SFQ packet scheduling. We set the number of buckets to 4 to simulate the common case where there are fewer buckets than flows.

Figures 15(b) shows the average throughput observed by different flows. We see that SFQ achieves almost equal throughput (true fairness) between the 6-hop flows and the 2-hop flow. We can also observe in Figure 14(b) that the 2-hop flow experiences a higher percentage of consecutive packet drop events (20% of the time, it experiences 2 consecutive drops). Since the 2-hop flow has a lower RTT, it is more aggressive as compared to the 6-hop flows, leading to more dropped packets than those flows.

5.3 TCP Pacing

TCP pacing, also known as "packet spacing", is a technique that spreads the transmission of TCP segments across the entire duration of the estimated RTT instead of having a burst at the reception of acknowledgments from the TCP receiver (*e.g.*, [1]). Intuitively, TCP pacing promotes the interleaving of packets of the TCP flows that compete for the output port in the TCP Outcast problem and hence can potentially alleviate blackout on one input port. We used the TCP pacing in our ns-2 [24] setup and repeated the same experiment as before. Figure 15(c) shows that TCP pacing reduces throughput unfairness; the throughput gap between the 2-hop flow and 6-hop flows is reduced from 7× (Figure 3(f)) to 2×. However, the Outcast problem remains. This is also seen in Figure 14(c), where the 2-hop flow still experiences many consecutive packet drops. The reason is as follows. There is only a single (2-hop) flow arriving at one of the input ports of the bottleneck switch. Hence, there is a limit on how much TCP pacing can space out the packets for that flow, *i.e.* the RTT of that 2-hop flow divided by the congestion window.

5.4 Equal-Length Routing

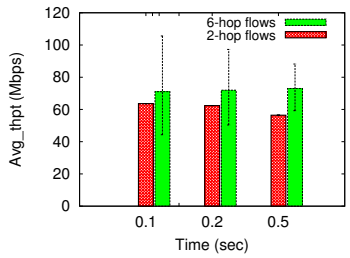
As discussed in Section 3, one of the conditions for TCP Outcast problem is the asymmetrical location of senders of different distances to the receiver, which results in disproportionate numbers of flows on different input ports of the bottleneck switch competing for the same output port. Given we can not change the location of the servers, one intuitive way to negate the above condition is to make flows from all senders travel similar paths and hence their packets are well mixed in the shared links and hence well balanced between different input ports. Before discussing how to achieve this, we briefly discuss a property of the fat-tree network topology that makes the proposed scheme practical.

In a fat-tree [2] topology, each switch has the same amount of fan-in and fan-out bandwidth capacity, and hence the network is fully provisioned to carry the traffic from the lowest level of servers to the topmost core switches and vice versa. Thus although the conventional shortest path routing may provide a shorter RTT for packets that do not need to reach the top-most core switches, the spare capacity in the core cannot be used by other flows anyways.

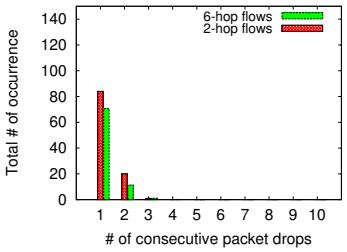
Based on the above observation, we propose *Equal-length routing* in a fat-tree data-center network topology, where data packets from every server are forwarded up to the core switch irrespective of whether the destination belongs in the same pod of the sender. Effectively, Equal-length routing prevents the precarious situations where a given flow alone suffers from consecutive packet losses (as discussed in Section 3). Since all the flows are routed to the core of the network, there is enough mixing of the traffic arriving at various ports of a core switch that the packet losses are uniformly shared by multiple flows. Equal-length routing ensures a fair share among multiple flows without conceding any loss to the total network capacity. It is simple to implement and requires no changes to the TCP stack.

Implementation. Equal-length routing can be implemented in a fat-tree by routing each packet to a core switch randomly or deterministically chosen. Under the random scheme, the core switch is randomly uniformly chosen [13]. Under the deterministic scheme, the core switch is determined based on the destination address as follows. On our testbed running OpenFlow for routing control, at the ToR switches, a packet coming from a server (down port) is forwarded to one of the aggregate switches (up ports) as decided by the destination address (*e.g.*, port selection is based on the last bit of the destination address). Similarly at the aggregate switches, packets coming from ToR (down) are forwarded to the core (up) switches and vice versa (*e.g.*, port selection based on the second last bit of the destination address). Consider the flow of packets from $S1$ to the $Dest$ in Figure 1. Without the Equal-length routing, the packets take the path $S1 \rightarrow Tor0 \rightarrow Dest$, but under Equal-length routing, the packets will go through $S1 \rightarrow Tor0 \rightarrow Agg0 \rightarrow Core0 \rightarrow Agg0 \rightarrow Tor0 \rightarrow Dest$.

Properties. Equal-length routing creates interesting changes in the dynamics of interactions among the TCP flows. Under the new routing scheme, all the flows are mixed at core switches (feasible in a network providing full-bisection bandwidth) which gives rise to two properties: (1) The deterministic scheme results in all flows in many-to-one communication sharing the same downward path, whereas the random scheme results in flows



(a) Average throughput



(b) Consecutive packet drops

Figure 16: Effectiveness of equal-length routing

going to the same destination being well balanced between different input ports at each switch in the downward paths. Both effects avoid the blackout of any particular flow; instead, all the competing flows suffer uniform packet losses. (2) All the flows have similar RTTs and similar congestion window increases. Together, they ensure that competing flows achieve similar true fair share of the bottleneck link bandwidth.

Evaluation. To analyze the proposed routing scheme, we implemented Equal-length routing in our data center testbed and conducted similar experiments as Figure 3(f). Other than the new routing scheme, all the setup was kept the same. We analyze the TCP throughput achieved by different flows as before. Note that even though every flow is now communicating via a core switch, we label them as 2-hop and 6-hop flows for consistency and ease of comparison with previous results.

Figure 16(a) depicts the TCP throughput share between different flows. We can observe that the flows get a fair throughput share which is comparable to what they achieved under the SFQ packet scheduling discipline. The fair share of throughput can be further explained from Figure 16(b), which shows that the flows experience similar packet drops; none of the flows has to suffer a large number of consecutive packet drops.

5.5 Summary

Table 2 summarizes the four potential solutions for the TCP Outcast problem we have evaluated. All solutions share the common theme of trying to break the synchronization of packet arrivals by better interleaving packets of the flows competing for the same output port and

Techniques	Fairness Property
RED	RTT bias
SFQ	RTT fairness
TCP Pacing	Inverse RTT bias
Equal-length routing	RTT fairness

Table 2: Fairness property of TCP Outcast solutions

hence evening out packet drops across them. We find that although all approaches alleviate the TCP outcast problem, RED still leads to RTT bias, and TCP pacing still leads to significant inverse RTT bias. SFQ and Equal-length routing provide RTT fairness but have their limitations too. SFQ is not commonly available in commodity switches due to its complexity and hence overhead in maintaining multiple buckets, and Equal-length routing is feasible only in network topologies without oversubscription. The final choice of solution will depend on the fairness requirement, traffic pattern, and topology of the data center networks.

6 Related Work

We divide related work into three main categories—TCP problems in data centers, new abstractions for network isolation/slicing, and TCP issues in the Internet context.

TCP issues in data centers. Much recent work has focused on exposing various problems associated with TCP in data centers (already discussed before in Section 2). The TCP Incast problem was first exposed in [18], later explored in [23, 7, 26]. Here the authors discover the adverse impact of barrier-synchronized workloads in storage network on TCP performance. [23] proposes several solutions to mitigate this problem in the form of fine-grained kernel timers and randomized timeouts, *etc*.

In [3], the authors observe that TCP does not perform well in mixed workloads that require low latency as well as sustained throughput. To address this problem, they propose a new transport protocol called DC-TCP that leverages the explicit congestion notification (ECN) feature in the switches to provide multi-bit feedback to end hosts. While we have not experimented with DC-TCP in this paper, the Outcast problem may potentially be mitigated since DC-TCP tries to ensure that the queues do not become full. We plan to investigate this as part of our future work.

In virtualized data centers, researchers have observed serious negative impact of virtual machine (VM) consolidation on TCP performance [15, 12]. They observe that VM consolidation can slow down the TCP connection progress due to the additional VM scheduling latencies. They propose hypervisor-based techniques to mitigate these negative effects.

In [21], the authors propose multipath TCP (MPTCP) to improve the network performance by taking advantage of multiple parallel paths between a given source and a destination routinely found in data center environments.

MPTCP does not eliminate the Outcast problem as we discussed in Section 3.2.6.

Network isolation. The second relevant body of work advocates network isolation and provides each tenant with a fixed share of network resources [14, 22, 4]. For example, SecondNet [14] uses rate controllers in hypervisors to ensure per-flow rate limits. Seawall [22] uses hypervisors to share the network resources according to some pre-allocated weight to each customer. Finally, Oktopus [4] provides a virtual cluster and a two-tier oversubscribed cluster abstraction, and also uses hypervisors to implement these guarantees. Our focus in this paper, however, is on the flow-level fairness as opposed to tenant-level isolation considered in these solutions.

Wide-area TCP issues. While this paper is mainly in the context of data centers, several TCP issues have been studied for almost three decades in the wide-area context. One of the most related work is that by Floyd *et al.* [10], where they study so-called phase effects on TCP performance. They discover that taildrop gateways with strongly periodic traffic can result in systematic discrimination and lockout behavior against some connections. While our port blackout phenomenon occurs because of systematic biases long mentioned in this classic work and others (*e.g.*, RFC 2309 [6]), they do not mention the exact Outcast problem we observe in this paper. RTT bias has also been documented in [10] where TCP throughput is inversely proportional to the RTT. TCP variants such as TCP Libra [16] have been proposed to overcome such biases, but are generally not popular in the wild due to their complexity. The typical symptom of the TCP Outcast problem in data centers is the exact opposite.

7 Conclusion

The quest for fairness in sharing network resources is an age-old one. While the fairness achieved by TCP is generally deemed acceptable in the wide-area Internet context, data centers present a new frontier where it may be important to reconsider TCP fairness issues. In this paper, we present a surprising observation we call the TCP Outcast problem, where if many and few flows arrive at two input ports going towards one output port, the fewer flows obtain much lower share of the bandwidth than the many flows. Careful investigation of the root cause revealed the underlying phenomenon of port blackout where each input port occassionally loses a sequence of packets. If these consecutive drops are distributed over a small number of flows, their throughput can reduce significantly because TCP may enter into the timeout phase. We evaluate a set of solutions such as RED, SFQ, TCP pacing, and a new solution called Equal-length routing that can mitigate the Outcast problem. In ongoing work, we are investigating the effect of the Outcast problem on real applications such as MapReduce.

8 Acknowledgements

The authors thank Praveen Yalagandula for help with experiments on HP's testbed. The authors also thank Jitu Padhye, Amin Vahdat (our shepherd) and anonymous reviewers for various comments that helped improve the paper. This work was supported in part by NSF Awards CNS-1054788 and CRI-0751153.

References

[1] A. Aggarwal, S. Savage, and T. E. Anderson. Understanding the Performance of TCP Pacing. In *IEEE INFOCOM*, 2000.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, 2008.

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

[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.

[5] P. J. Braam. File systems for clusters from a protocol perspective. `http://www.lustre.org`.

[6] B. Braden, D. Clark, J. Crowcroft, et al. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, 1998.

[7] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *USENIX WREN*, 2009.

[8] J. Dean, S. Ghemawat, and G. Inc. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[9] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor sharing flows in the internet. In *IEEE IWQoS*, June 2005.

[10] S. Floyd and V. Jacobson. On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 1992.

[11] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, August 1993.

[12] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds. In *2nd ACM Symposium on Cloud Computing (SOCC'11)*, 2011.

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

[14] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *ACM CoNEXT*, 2010.

[15] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In *ACM/IEEE SC*, 2010.

[16] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Y. Sanadidi, and M. Roccetti. TCP-Libra: Exploring RTT Fairness for TCP. Technical report, UCLA Computer Science Department, 2005.

[17] P. E. McKenney. Stochastic fairness queueing. In *IEEE INFOCOM*, pages 733–740, 1990.

[18] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *ACM/IEEE SC*, 2004.

[19] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM*, 2009.

[20] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM*, 1998.

[21] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM*, 2011.

[22] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *USENIX HotCloud*, 2010.

[23] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM*, 2009.

[24] D. X. Wei. *A TCP pacing implementation for NS2*, April 2006 (accessed October 03, 2011).

[25] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[26] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in data center networks. In *ACM CoNEXT*, 2010.

[27] J. Zhang, F. Ren, and C. Lin. Modeling and Understanding TCP Incast in Data Center Networks. In *IEEE INFOCOM*, 2011.