The TCP Outcast Problem: Exposing Unfairness in Data Center Networks

Submitted in requirement for the course

ADVANCED COMPUTER NETWORKS (CSN-503)

of Bachelor of Technology in Computer Science and Engineering

by

Abhishek Sajwan (15114002)

Ankita Saxena(15114011)

Nitish Bansal (15114048)

Sahil Grover (15114061)

Submitted to

Dr. P. Sateesh Kumar

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE

ROORKEE- 247667 (INDIA)

1. Introduction

We explore the problem of TCP outcast. Outcast presents itself under two specific conditions. The first is the use of tail-drop queuing in your routers. The second is an asymmetry in the number of flows arriving at different ports of a router with the same destination. This causes port-blackout. Port-blackout occurs when a series of back-to-back packets coming into a port are dropped. So when two different ports send to one port, one of the senders gets unlucky and has its packets dropped due to port-blackout.

The asymmetry exacerbates this problem since the port with few flows could lose a flow's congestion window tail. This leads to more catastrophic effects than seen with the other port, which may drop the same amount of packets but interspersed over a large number of flows. This is less consequential because from any one flow's view only few packets are dropped leading TCP to react in a less severe fashion.

The conditions for outcast are actually fairly easy to meet and are met by many data centers. The use of commodity off-the-shelf (COTS) routers means the tail-drop requirement is often satisfied as most COTS routers resort to this for their queuing control.

The multi-rooted tree topology of data centers contributes to the asymmetry requirement. If we consider a common topology, the fat-tree topology, we see exactly this condition arising due to the fact that senders are likely to originate in different parts of the network and are thus likely to arrive at different input ports to the switch right before the receiver. Figure 1 shows that the number of nodes 2n hops away increases exponentially, and

therefore the number of nodes in disparate locations is high. Figure one demonstrates this happening in a fat-tree topology.

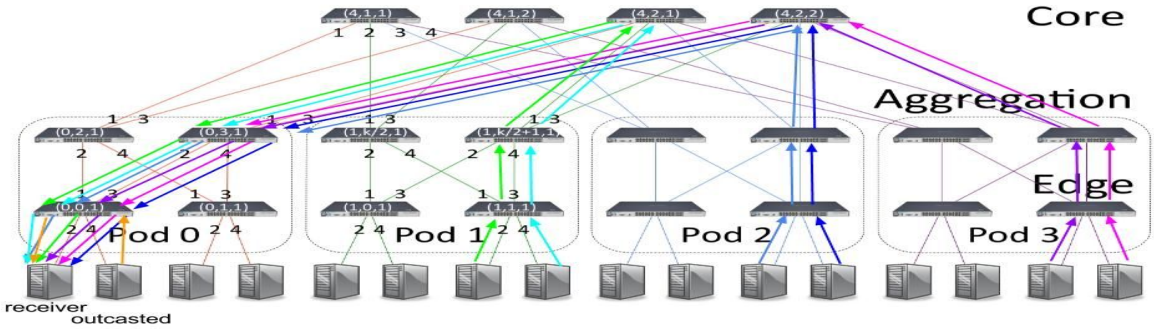


Figure 1. *Example of outcast asymmetry in a k=4 Fat Tree topology*

In this work we recreate the experiments of the paper showing outcast occurring. We will do this by setting up a port-asymmetric arrival pattern to tail-drop switches, where (n-1) of n flows will arrive at one port and 1 flow will arrive at another. The work will show the adverse effect of the asymmetry on the flow arriving at a port by itself. We will do this first on a more contrived topology to demonstrate the effect and then consider a realistic fat-tree topology. To finish we consider the effect of non-static routing on Outcast by using hashed routing to route the flows in fat-tree. The primary goal will be to replicate Figure 2 from the paper.

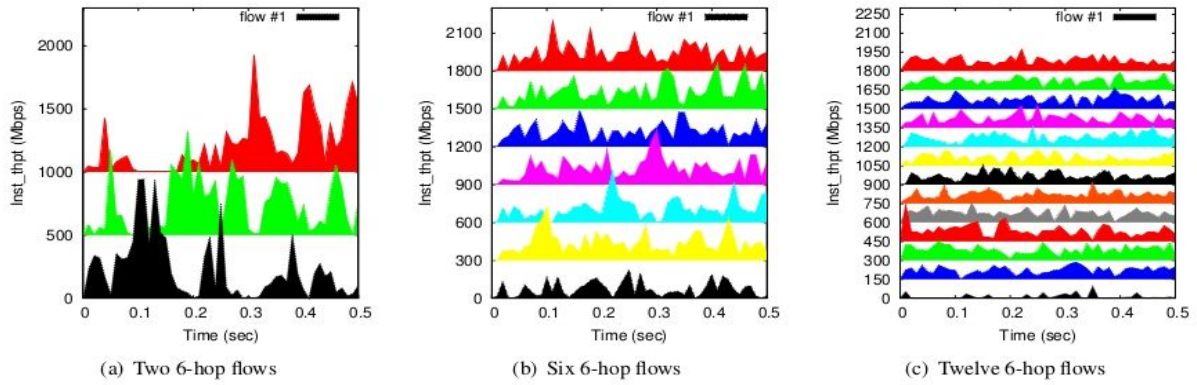


Figure 2. *The results showing Outcast from the original paper*

2. Motivation

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.

This paper presents a surprising observation called 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 in the paper revealed the underlying phenomenon of port blackout where each input port occasionally 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.

In the paper, a set of solutions such as RED, SFQ, TCP pacing, and a new solution called Equal-length routing are evaluated that can mitigate the Outcast problem. This set of solutions make the paper very relevant by providing various ways to deal with the TCP Outcast problem.

3. 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. The TCP Incast problem was first exposed in [8], later explored in [11, 3, 12]. Here the authors discover the adverse impact of barrier-synchronized workloads in storage network on TCP performance. [11] proposes several solutions to mitigate this problem in the form of fine grained kernel timers and randomized timeouts, etc. In [1], 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 [7, 5]. 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 [9], 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. The second relevant body of work advocates network isolation and provides each tenant with a fixed share

of network resources [6, 10, 2]. For example, SecondNet [6] uses rate controllers in hypervisors to ensure per-flow rate limits. Seawall [10] uses hypervisors to share the network resources according to some pre-allocated weight to each customer. Finally, Oktopus [2] 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. [4], 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, they do not mention the exact Outcast problem we observe in this paper. RTT bias has also been documented in [4] where TCP throughput is inversely proportional to the RTT. TCP variants such as TCP Libra 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.

4. 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 the figure below. While the fat-tree topology is used here, the problem is just not occurs in fat-tree but all the other topologies as well. Also, in a fat-tree all links are of the same capacity (assume 1Gbps in this case). Now suppose there are 15 TCP flow's fi (i = 1...15) from sender Si to Dest. All these flows need not start simultaneously, but we mainly consider the portion of time when all the 15 flows are active.

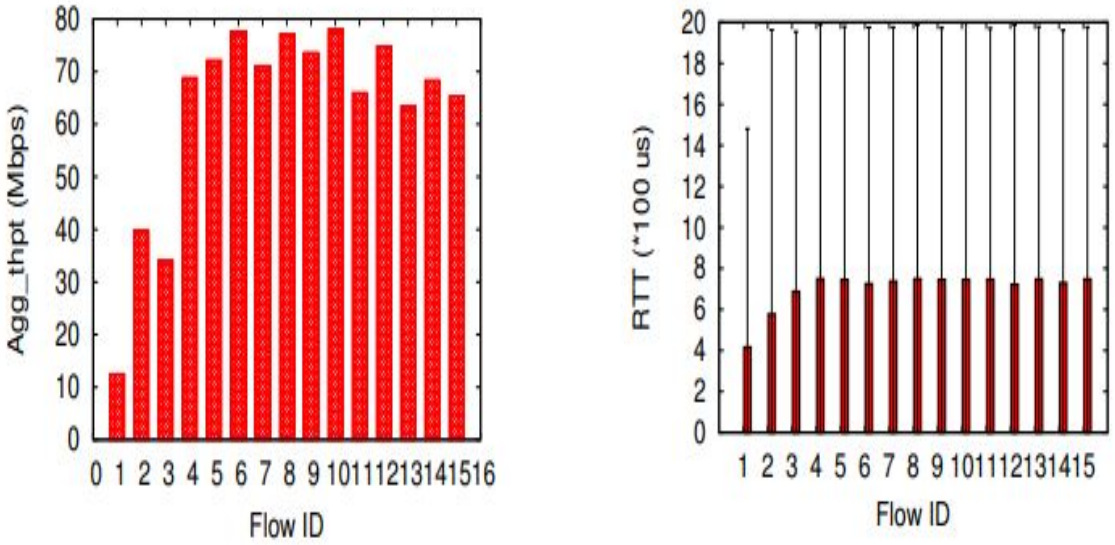


Figure 3. *The TCP Outcast Problem.*

The above figure shows that the per-flow aggregate throughput obtained by these 15 flows exhibit a form of somewhat surprising unfairness: Flow f1 which has the shortest RTT achieves significantly lesser aggregate throughput than any of f4 - f15 — almost 7-8× lower. Flows f2 and f3 also achieve lesser throughput (about 2×) than f4 - f15. This was observed as a

general trend under many different traffic patterns, and at different locations of the bottleneck link, although the degree of unfairness varies across scenarios.

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 f1, and to some extent f2 and f3, are outcasted by the swarm of other flows f4 – f15. Although f1 – f3 differ in their distances (in terms of number of hops between the sender and the receiver) compared to flows f4 – f15, this does not alone explain the Outcast problem. If any, since f1 has a short distance of only 2 links, its RTT should be much smaller than f4 – f15 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, throughput is inversely proportional to the RTT, which suggests f1 should obtain higher throughput than any of f4 – f15. However, the Outcast problem exhibits exactly the opposite behavior.

The reason for this counter-intuitive result is twofold: First, taildrop queuing leads to an occasional “port blackout” where a series of back-to-back incoming packets to one port are dropped. 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.

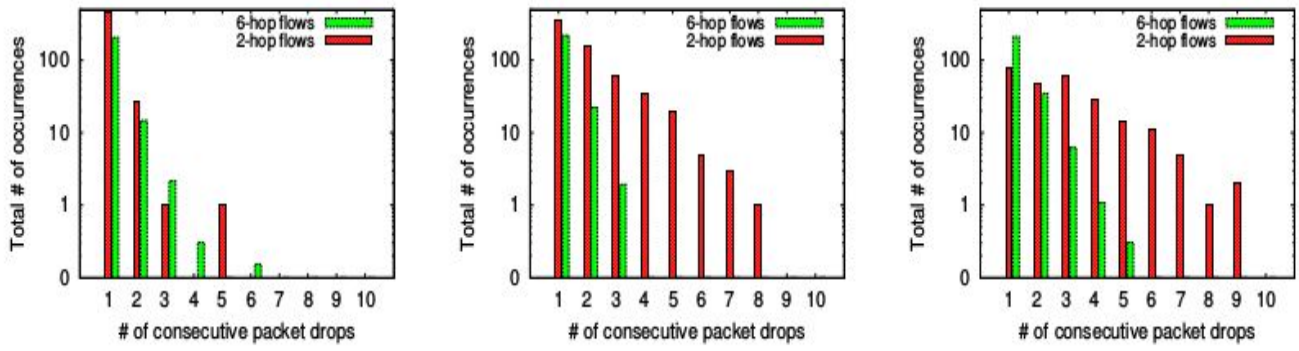
To summarize, the two conditions for the Outcast problem to occur are as follows:

- The network consist of COTS switches that use the taildrop queue management discipline.
- 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.

5. Solution

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

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.



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

Figure 4. *Distribution of consecutive packet drops under RED, SFQ, and TCP pacing solutions*

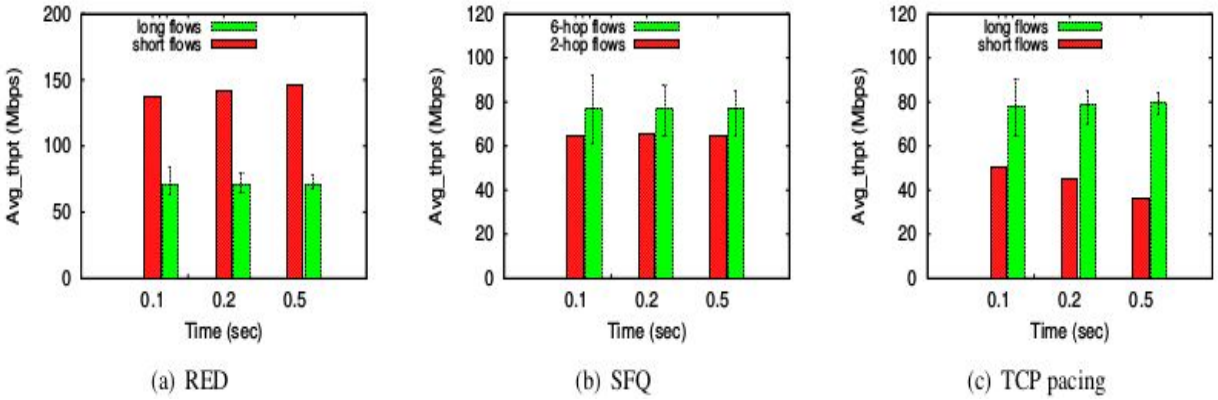


Figure 5. *Average throughput under RED, SFQ, and TCP pacing solutions*

5.1 RED

RED 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 6, with 12 6-hop flows and 1 2-hop flow destined to a given receiver.

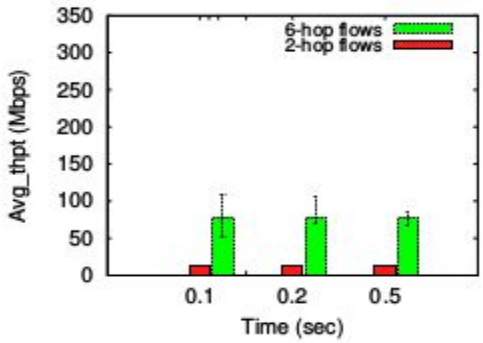


Figure 6. *Twelve 6-hop flows*

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 4(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 5(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 ~ $1/(RTT * \sqrt{droprate})$). 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), 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 5(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 4(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. 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 setup and repeated the same experiment as before. Figure 5(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 6) to 2×. However, the Outcast problem remains. This is also seen in Figure 4(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

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 the location of the servers cannot be changed, 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. In a fat-tree 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, 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. 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.

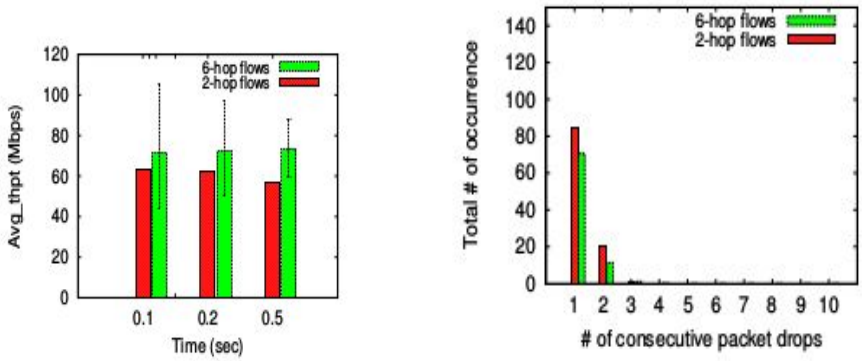


Figure 7. *Effectiveness of equal-length routing*

6. Experiment

The experimental results from the paper use a data center testbed configured in a k=4 fat-tree topology, with 16 hosts at the leaf and 20 servers acting as 4-port gigabit switches with 16KB tail-drop queued packet buffers per port. To reduce the effects of TCP Incast, minRTO on the hosts is configured to 2ms. All other TCP parameters are left at their defaults.

The authors use a fat-tree topology to illustrate how the conditions for TCP Outcast could occur in practice from a many-to-one traffic pattern, yet the results show that the only requirements are switches that use a taildrop queuing discipline and an imbalance in the number of flows on each incoming port. As such we started with a contrived minimal topology consisting of only two switches (one acting as an aggregator for the larger group of flows), then later emulated a k=4 fat-tree topology as used in the paper. In both cases, we mimic the environment in the original paper using the following configuration:

hosts: minRTO is set to 2ms using the "ip route replace …" tool. For comparison against results presented in the paper, all traffic is TCP BIC flows generated by iperf.

links: The "port blackout" effect illustrated above is sensitive to the queuing discipline of the switch, and the burst behavior of the links. If the burst-rate of ports A and B differ, they will not synchronize and prevent us from observing the effect. Mininet emulates links using linux "tc qdisc" to set the queuing discipline and rate-limiting. To emulate the taildrop queuing necessary for TCP Outcast, we experiment with both "token bucket filter" (TBF) [13] and "hierarchical token bucket" (HTB) queuing disciplines, the latter

being Mininet's default. Our experiments use a 100 Mbps link speed, down from 1 Gbps in the paper which we were not able to reliably emulate on Mininet. Although we experimented with both TBF and HTB, we present results with TBF for the simple topology and with HTB for fat tree.

routing: The paper uses shortest distance routing with fat tree and uses the last bits of the destination host address to select the higher level switches to break ties between multiple shortest routes. We emulate routing in mininet on fat tree using the same algorithm.

To measure per-flow throughput we collect unsampled TCP dumps at the bottleneck switch interfaces. As in the paper, we join the TCP dumps from different ports to reconstruct the switch queue and investigate the "port blackout" phenomenon.

We use Mininet to recreate the topology in figure 8. We use a sender per flow, as outcast is dependent on flow count versus host count this works well with the requirements. We make (n-1) senders send to an aggregate switch which outputs to one port of the switch before the receiver. We use another port for a single host sending a single flow. This creates an asymmetry as now there are (n-1) flows going into port-0 of the receiver switch and 1 flow going into port-1. We plot the throughput of each sender.

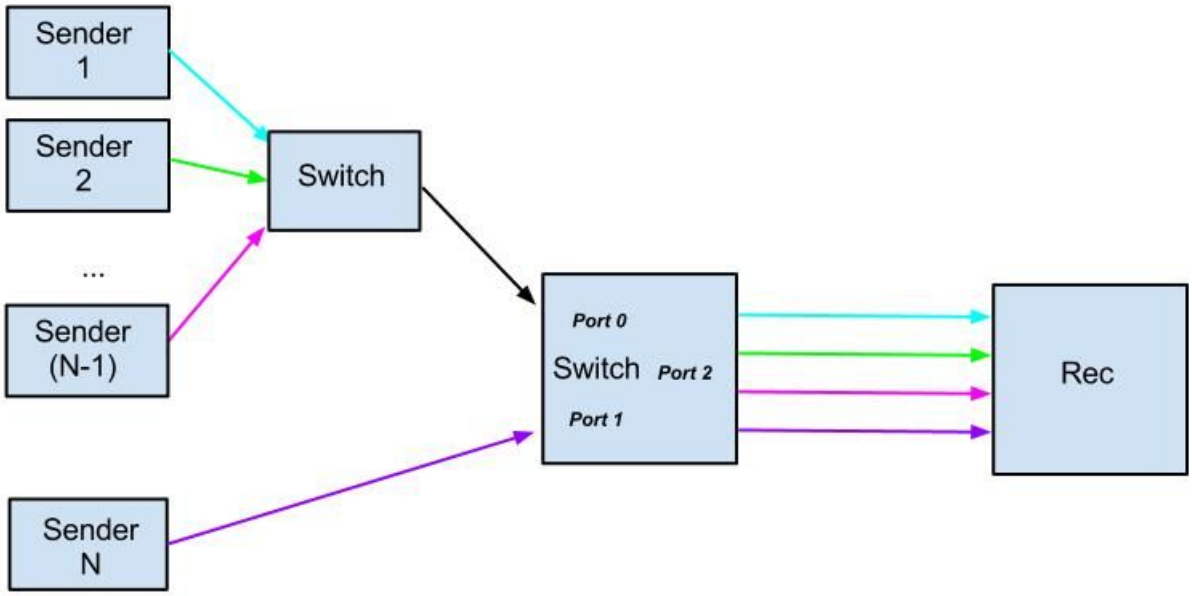


Figure 8. *Topology built in MiniNet to simulate Outcast*

We also use Mininet to build a k=4 fat-tree topology. We set up one host to be the receiver and 15 other hosts as senders. One of the sender hosts shares a switch in the same pod as the receiver and so will have its own port on the switch. The other senders will be coming in through the remaining two ports on the switch connecting to the receiver.

7. Results

7.1 Observing Outcast

We let the senders run for 60 seconds plotting the throughput of each sender to the aggregate or the receiver switch. In our data, S1 was the receiver switch and S2 was the aggregate switch. We ran 2,6 and 12 flows in competition with the one flow sending to S1.

Our results closely mimic those of the paper. We see that as the number of competing flows increase, the throughput of the single-flow drops dramatically, while the flows coming in within the larger set share bandwidth fairly equally.

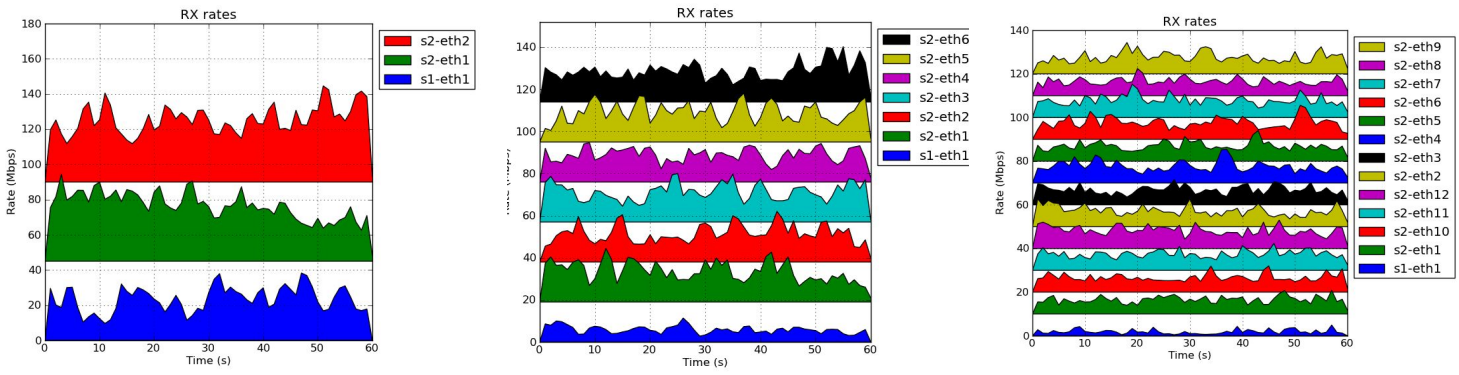


Figure 9. *Outcast with topology of Figure 1 with 2,6 and 12 competing flows*

7.2 Effects of Topology and Routing

We next consider the effect of topology on outcast. To do this we form a k=4 Fat Tree topology, having 15 of the 16 hosts send to one other host. We show only a subset of the hosts which do not share the same pod switch as the

receiver because the graph gets harder to read with the full 15 and their behavior is the same as the selected ones.

We first use spanning tree routing which ensures that there is one path leading to the pod switch before the receiving host. This should recreate the earlier conditions where 14 of the 15 sending hosts (those not connecting to the same pod switch as the receiver) will all send through one port while one of the sending hosts will be sending through another, competing for the one output link. We get the results in Figure 10(a). These results show an extreme occurrence of Outcast where the throughput of the isolated sender does not even register. This makes sense as now there are even more competing hosts than in the previous example where the throughput was quite low already.

We also tested the effects of routing, considering routing based on hashes. We wanted to see if the distribution across the incoming ports would lead to a lessening of the effect. Our results can be seen in Figure 10(b). We actually observe no improvement to the throughput of the outcasted node as a result of this. One possible reasoning is that the distribution is just not enough given that there are only two ports to spread the load over. Even with perfect uniform hashing this leads to about seven hosts on each port competing with the one. We do, however, see other flows getting starved in the hashing case. One way to explain this is that now instead of having all the flows except one go on one port you have all the flows except, say, 2 go on one port and then you are creating an asymmetry between the 13 flow bulk and the two flows that are now by themselves. This could result with unfortunate hashing, but this is just speculation and further study would be necessary. Finally, the drop in throughput around the thirty second mark could be due to RipL's response to timed-out paths, something that is currently untested and possibly leads to lost flows. The spikes at the end are harder to explain, but could be explained

due to special behavior arising out of tear-down procedures, though this is clearly simply speculation.

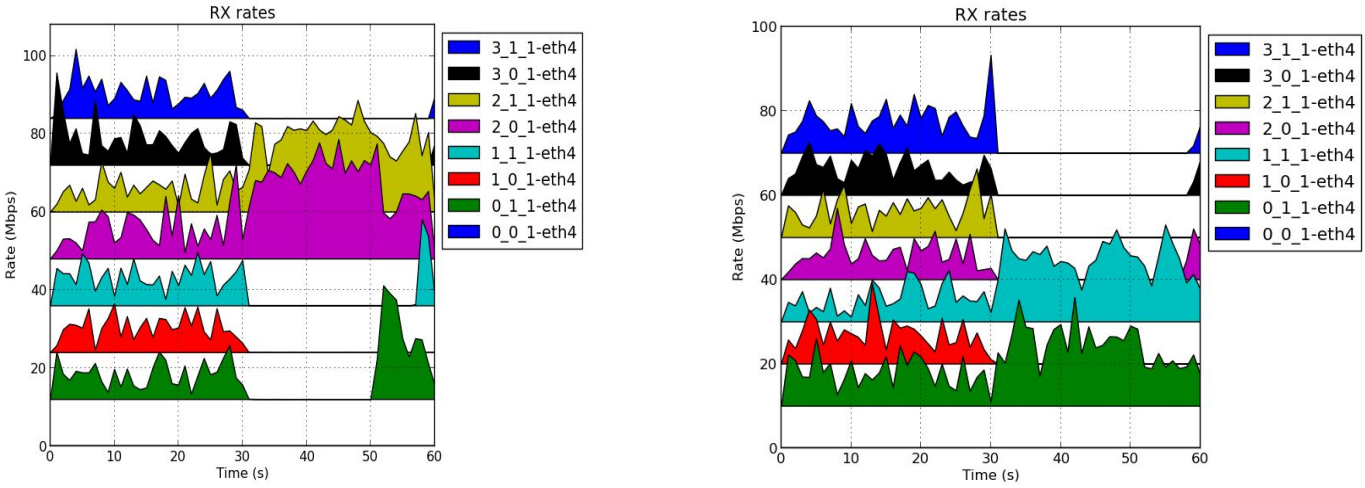


Figure 10. (a) *Outcast in K=4 Fat-tree topology using spanning tree routing*
(b) *Outcast in K=4 Fat-tree topology with hashed routing*

8. Drawbacks/Challenges

1. For the final project, a new instance type was created which used DCTCP and allowed us to leverage RipL for the Fat Tree topology. However, this version of Mininet changed the drop discipline for the router, which is one of the key components of the outcast problem. As a result, we were unable to reproduce our linear topology results using this instance.

2. The most challenging aspect was making the assignment work with the Fat Tree topology, which required us to introduce RipL and POX, additional libraries that we had yet to work with. Ultimately it felt like the configuration was very fragile and opaque, requiring a lot of conceptual overhead to integrate. An example of this is the fact that RipL's behavior after path timeout is untested, something that was not known and could have contributed to result inconsistency.

3. Finally, we saw minor performance issues on smaller instances. When we first started on a t1.micro instance, the tasks often failed to finish successfully, but switching to a c1.medium instance type solved this issue for us.

9. Improvements

Network isolation : The second relevant body of work advocates network isolation and provides each tenant with a fixed share of network resources. The focus in this paper, however, was on the flow-level fairness as opposed to
tenant-level isolation considered in these solutions.

Achieving Fairness: TCPRand, a transport layer solution to TCP outcast has been proposed by Lee et al[14] in 2017. The main idea of TCPRand is the randomization of TCP payload size, which breaks synchronized packet arrivals between flows from different input ports. Based on the current congestion window size and the CUBICs congestion window growth function, TCPRand adaptively determines the proper level of randomness. With extensive ns-3 simulations and experiments, the paper shows that TCPRand guarantees the superior enhancement of TCP fairness by reducing the TCP timeout period noticeably even in an environment where serious TCP outcast happens. TCPRand also minimizes the total goodput loss since its adaptive mechanism avoids unnecessary payload size randomization. Compared with DCTCP, TCPRand performs fairly well and only requires modification at the transport layer of the sender which makes its deployment relatively easier.

10. References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] Reference on TBF queueing discipline: http://www.opalsoft.net/qos/DS-24.htm

[14] TCPRand: Randomizing TCP payload size for TCP fairness in data center networks https://ieeexplore.ieee.org/document/7218550