Turbo: Ultra Low Latency Datacenter Transport

Akshay Narayan Sarah Hung Kaifei Chen Gautam Kumar
Sylvia Ratnasamy Ion Stoica
University of California, Berkeley

Abstract

In this paper we motivate and present Turbo, a new datacenter transport protocol that is designed very low BDP environments in mind. At such low BDP, Turbo delivers perfect utilization while also minimizing average flow completion times for all flows, whether short or long. More importantly, Turbo delivers this minimized average flow completion time in a reliable and predictable fashion. Turbo is based on a simple idea - namely, that flows should simply send all the time while carrying a priority, and all congestion control should be decoupled from rate control and done entirely in the network using priority queues. We show through extensive experimental data that Turbo achieves its designed goals of predictability and good performance.

1 Introduction

Contemporary datacenter network workloads consist primarily of various requests and responses among both internal servers and to the end hosts. Applications common in a datacenter network tend to be heavy-tailed [1, 2]; that is, most of the flows are short flows, but most of the bytes sent across the network belong to the small number of long flows. This leads to scenarios in which FIFO queueing can lead to disastrous performance on metrics such as the mean slowdown, which measures the mean of how much greater a flow's completion time in the network compared to the "Oracle FCT" - that is, what it would have been had no other flows existed. The mean slowdown metric is especially affected in networks with FIFO queues because the multitude of packets from the long flows can cause prolonged queueing delay for the few packets from short flows. Since the short flows have a low oracle FCT, the queueing delay affects them especially. So, we conclude that FIFO queues are not appropriate for our purposes.

Meanwhile, link latency in a datacenter network can theoretically approach 1 microsecond [3], but in practice is usually up to three orders of magnitude slower. This high latency is caused by queueing delays experienced by each packet at times of heavy traffic and congestion. The negative effects are most evident when a packet is enqueued into a large buffer, where it must wait for a large and, more importantly, highly variable amount of time while until it is selected for dequeuing. This observation paved way for the conjecture that it is possible to achieve a very low latency datacenter network by reducing the internal switches' buffer size. If link latency is set to 1 or 2 microseconds and link bandwidth is a standard 10 Gbps, the bandwidth-delay product (BDP) of the link approaches one or two packets. Buffer sizes should then also be set to approximately one or two packets. Thus we reach our second conclusion in the motivation of Turbo; namely, achieving an ultra low latency network requires minimal switch buffering.

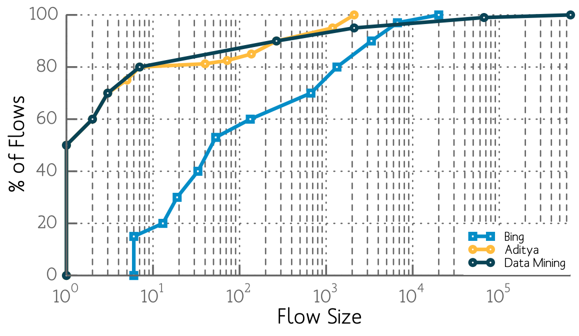
Attempting to use another tried and tested tool of networking, TCP congestion control, is similarly detrimental to flow completion time in the context of datacenter networks. While congestion control is necessary in order to keep the network functional, Most existing congestion control proposals fall under two categories. Proposals in the first category explicitly compute and assign appropriate rates to all flows based on some parameters and certain intrinsic informations of those flows, say size or deadline. In contrast, proposals from the second category implicitly allocate rate to the flows yet also alter them dynamically upon congestion signals or other event notifications. However, all existing proposals rely on some form of rate control as the basic mechanism. We find that rate based congestion control in datacenters leads to starvation scenarios in which the flows are unable to react quickly to current network conditions. The final motivation for Turbo, then, is a desire to move away from rate based congestion control.

The goal of the paper is to design a simple congestion control algorithm that makes flow completion time minimal and predictable. Our work is primarily inspired by the pFabric [4] transport protocol proposal, which is characterized by the use of priority queues in the network. In short, pFabric approximates a global shortest remaining processing time (SRPT) algorithm by setting a packet's priority to be equal to the remaining size of its flow. However, pFabric, in keeping with other contemporary congestion control mechanisms, also relies on rate control to prevent congestion collapse. Unfortunately, this reliance on rate control leads to starvation, which negatively impacts performance.

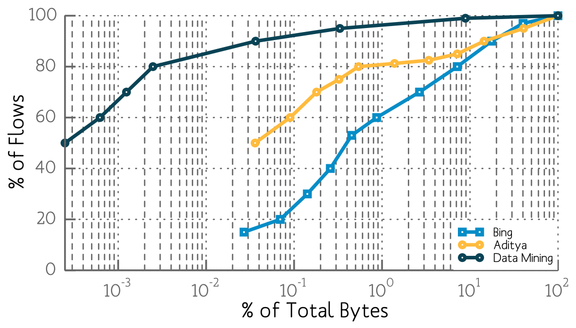
Turbo departs from pFabric's example a few important ways. Most importantly, rate based congestion control is abandoned entirely; all flows send at line rate. Congestion control is accomplished entirely using priorities. Like pFabric, packet drops are detected using timeouts alone. When a packet drop is detected, the priorities of subsequent packets of that flow are artificially inflated to make packets of that flow more likely to be dropped. The intuition is that some other flow of higher priority is consuming resources along the flow's path, so the flow should back off in order to avoid wasting resources that could be used by other flows whose paths are free.

We have performed extensive experimentation on both the pFabric and Turbo designs. The results support the hypothesis that pFabric leaves room for improvement in the low BDP scenarios we are concerned with. Meanwhile, the Turbo design has been plagued with some inefficiencies that we will discuss in section 4.

We provide background survey on our problem in Section 2 concluding that the state-of-the-art proposal to achieve small flow completion times is pFabric. Analysis on pFabric's design and performance drawbacks is provided in Section 3. Section 4 presents the discussion of the main Turbo implementation and design choices, while simulation detail and experimental results are discussed in Section 5.



(a) CDF of flow sizes in packets



(b) CDF of total bytes

Figure 1: **CDF of flow sizes and of total bytes from three different datacenter workloads**

2 Background

In order to highlight the differences and challenges that datacenter networks pose, in this section, we characterize them, both in terms for network topologies and traffic patterns. We highlight various research proposals to understand the different types of problems in this setting.

2.1 Datacenter Network Characterizations

2.1.1 Topology

Today, Fat-tree/Clos topology [5, 6] is the most common datacenter network topology [4]. These topologies are able to provide full-bisection bandwidth by using multiple roots for the tree (the core switches) which are connected to the top-of-rack switches either directly or hierarchically through aggregation layers. These networks are also able to push congestion to the edges, in that there are very few drops in the core of the network [7, 2]. This also makes the network amenable to be approximately modeled as a bipartite graph, where the congestion only happens at the edges.

2.1.2 Latency

Round-trip latency in data centers is going down at a fast paced. This is attributed both to increasing link capacities (10Gbps is now common in deployments at Bing, Facebook and Google) [8, 9], decreasing switching delays and the increasing popularity of cut-through switching [10], and improvements in the end-host stack [11, 12, 13]. It is envisioned that the next 5-10 years will reduce the round-trip time in data centers to 1-2 us [3]. At 10Gbps, this amounts to a bandwidth-delay product of just one or two 1500-byte packets!

2.1.3 Flow Size Distributions

Numerous studies [2, 14, 1] have characterized traffic patterns inside the datacenter network. Flow-size distributions are found heavy-tailed in almost all the studies that have been undertaken to characterize such networks. To illustrate, we present the CDF of flow sizes as well as the CDF of total bytes in three datacenter workloads, namely from the **Bing** web-search cluster [1], **Datamining** cluster at Microsoft [4] and another workload from a Microsoft cluster [2], called **Aditya** henceforth. Figure 1 illustrates the CDF of flow sizes and of total bytes across these three datacenter workloads. All three workloads exhibit a common trend, where most of the flows are small, e.g., 80% of the flows in the Aditya workload are smaller than 6 packets ($\sim$ 9KB), whereas most of the bytes lie in the long flows, e.g., 90% of the bytes are attributed to the 10% longest flows. This entails that most of the congestion are caused by these long flows and the short flows, if allowed to run by themselves in the absence of large flows, would perform near-optimal.

2.2 Traffic Engineering inside the Datacenter

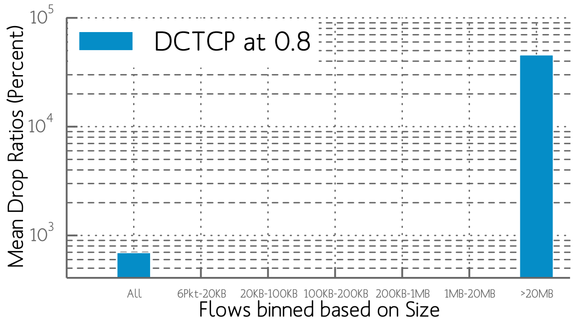
Multiple proposals have called the need to engineer traffic to take paths in order to alleviate network hotspots [15, 14]. This is facilitated by having multiple paths between a source and destination owing to the prevalent clos-topologies in today's datacenter networks. However, such schemes are only able to work at longer time-scales and do not safeguard against burst congestion. Further, when the bandwidth-delay product of the network becomes just a couple of packets, such scenarios are bound to become more common.

2.3 Better Transport Mechanisms

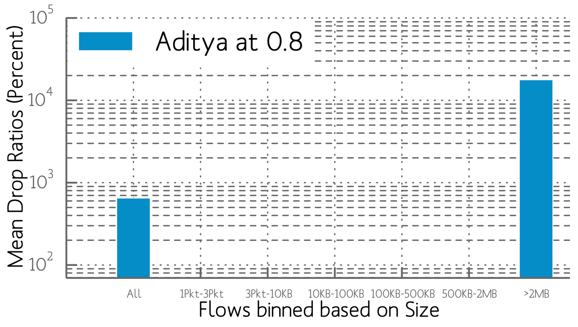
To achieve low-latency, another thread of proposals has been to optimize at the transport layer. Datacenter TCP (or DCTCP) [1] is a transport-layer protocol that offers TCP-style fairness whilst limiting the inherent burstiness of TCP. TCP's burstiness is owed to drastic adjustments to a flow's rate on timeouts and duplicate acks. DCTCP uses ECN marking and does gentler adjustments to the rate which helps keeping the queues small. Further, lower queueing delays enable short flows to finish quickly. To optimize on flow completion times and facilitate flows from interactive services to meet the tight deadlines imposed on them, numerous proposals have propagandandized the need for deadline-aware scheduling in the network. D^{2}TCP [16] is a modification of DCTCP that adjusts congestion window based on the flow's deadline.

2.4 Better Switch Fabrics

Switch architecture have received considerable attention both in the research community and the industry. While cut-through switching has started becoming very popular [10], a number of research articles have called to do better flow scheduling in the switch fabrics. D3 [17] tackles the problem of meeting flow deadlines by making switches explicitly reserve bandwidth for flows so that they can meet their deadlines. PDQ [18] tracks the active flows at a switch and schedules the ones with the earliest deadline (or smallest remaining size) while stopping the remaining flows to meet deadlines (or minimize flow completion time). The state-of-the-art in such protocol has been pFabric [4] which makes the switches do priority scheduling and dropping and using a minimal TCP-based transport protocol at the edge to achieve low flow-completion times. The priority based switching fabric priorities flows in the order of remaining flow size. As we later show, this can result in extremely high drop rates for long flows, which along with TCP's aggressive handling of losses can lead to potential starvation scenarios.



(a) Bing



(b) Aditya

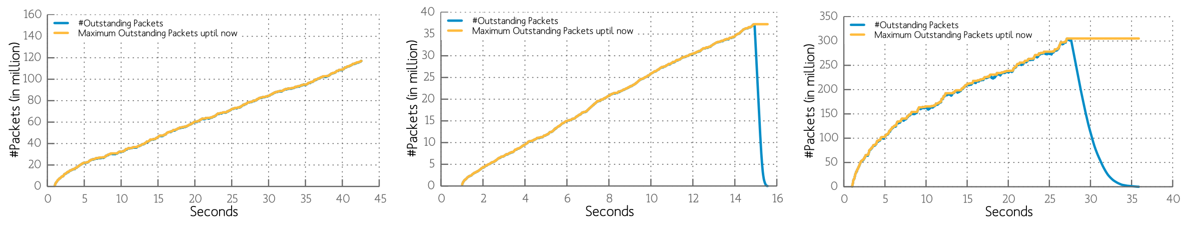
Figure 2: **At 0.8 load, drop rates are astronomical for long flows in both the Bing and Aditya workloads. The y axis shows the percentage drop rate on a log scale.**

2.5 End to End Inefficiencies

Irrespective of how well the network does flow scheduling, end-hosts can put a heavy burden on network latency [13, 19]. Much work has been done to optimize the end-host stack so as to rid it of the numerous inefficiencies that current stacks are plagued with [20, 21].

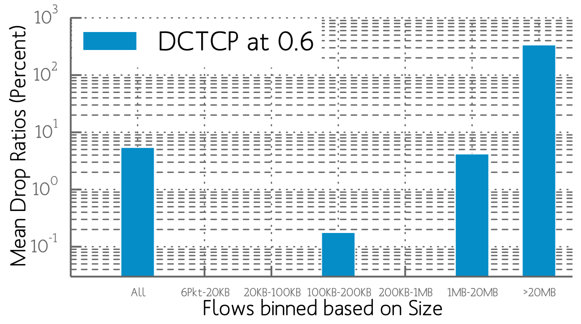
3 Potential For Improvement

We have found that the pFabric protocol, the current state of the art, performs poorly under the low BDP conditions with which this paper is concerned. The pFabric protocol sets packet priorities to be equal to the number of bytes remaining in the flow; that is, the number of bytes in the flow that have not yet been acked. The size of a switch queue is set to twice the BDP number of packets. When a packet arrives at a switch queue, it is added to the queue if the buffer is not full. Otherwise, the packet

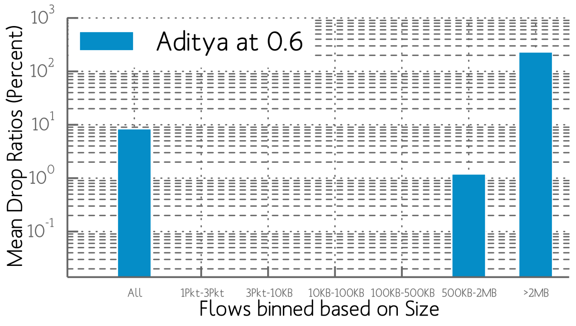


(a) Bing (b) Aditya (c) DataMining

Figure 4: **Under 0.8 load, the network is flooded by packets from long flows which are starved by higher priority short flows. These results are for a 1 million flow simulation.**



(a) Bing



(b) Aditya

Figure 3: **Even at 0.6 load, both the Bing and Aditya workloads experience high drop rates for the long flows. The y axis shows the percentage drop rate on a log scale.**

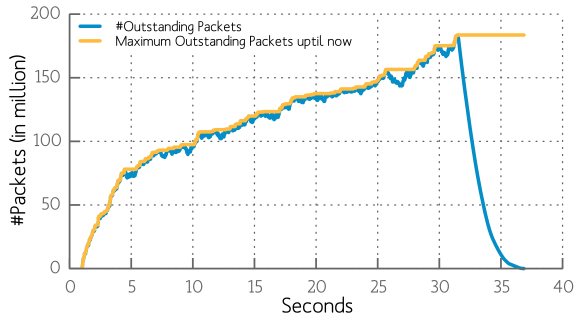


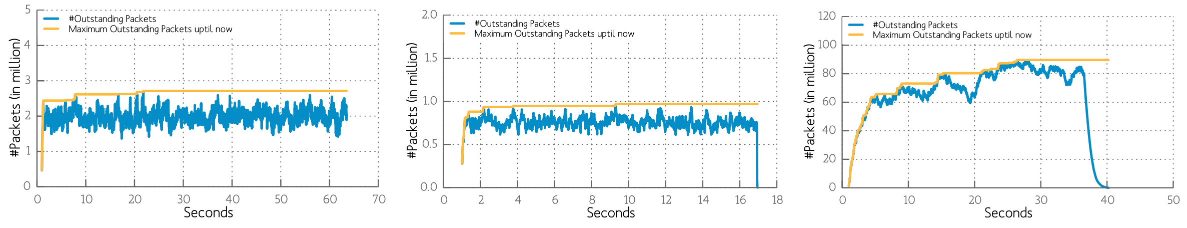
Figure 6: **Even at 0.7 load, the datamining workload creates a starvation scenario.**

with the least priority in the queue - in other words, the packet from the flow with the most bytes remaining - is dropped. When the queue is dequeuing, the packet with the greatest priority - in this case, the earliest packet from the flow with the least number of remaining packets - is sent. In this way, pFabric queues perform an approximation of SRPT scheduling.

pFabric's sole recovery mechanism is timeout-based. The timeout for a given packet is set to 3*RTT. While the initial CWND is set to be equal to the BDP so that flows start out sending at line rate, after a timeout this is abandoned and flows fall back to normal TCP slow start. Ergo, pFabric relies on rate control as its congestion control mechanism.

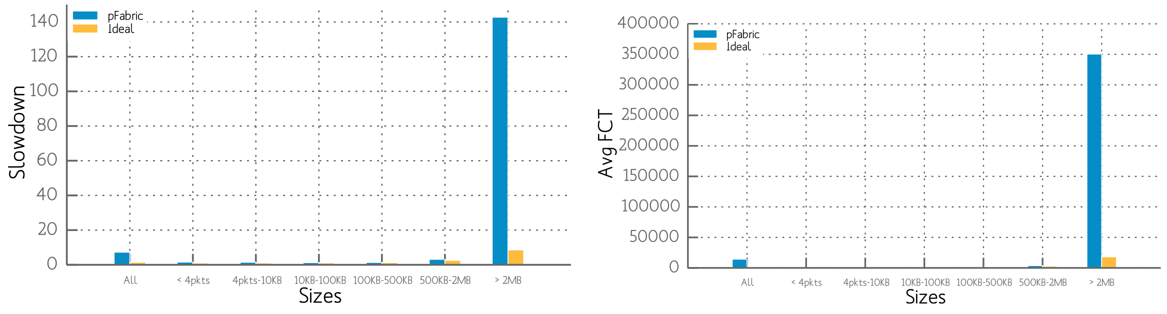
Since pFabric sets its queue size to be twice the BDP, in the low BDP scenarios with which we are concerned the queue size used by pFabric becomes quite small. In such situations, we have found that pFabric performs poorly because it starves long flows while servicing short flows. In distributions that are more heavy-tailed - that is, most flows are short but most bytes are part of long flows - pFabric performs especially poorly as the large number of short flows continually interrupt the long flows from making significant progress. In experiments we ran, long flows tended to only finish once the short flows had been drained from the network. This draining of the short flows probably would never occur in real-world conditions since a short flow that finished could trigger the formation of other short flows. The relentless onslaught of short flows would cause the long flow to be completely starved.

This finding can be seen in figures 2 and 4. Figure 2 shows the drop rates of large flows under a workload gathered from the Bing Datacenter. Note that the drop rate for the largest group of flows surpasses an astounding 45,000 percent. Figure 4 shows that pFabric is indeed starving the long flows by charting the number of outstanding packets - that is, the sum of the amount of unacked bytes from all flows that have already started but have not yet finished. As can be seen, the number of



(a) Bing (b) Aditya (c) DataMining

Figure 5: **Even under a lower 0.6 load, the most heavy-tailed workload creates a starvation scenario for the long flows. Under the Bing and Aditya workloads, the simulation is not a starvation scenario. These results are for a 1 million flow simulation.**



(a) Slowdowns (b) Average FCT

Figure 7: **pFabric performs quite far from the ideal for long flows.**

outstanding bytes steadily increases until the the very end of the simulation. This supports the conclusion that the short flows are starving the long flows, since bytes from teh long flows are building up as outstanding packets in the network.

As a result of this starvation, performance on both the average flow completion time metric as well as the mean slowdown metric were significantly worse than ideal for long flows. This is because the high drop rates for long flows cause them to continually re-enter Slow Start, and each timeout incurred will decrease SSTHRESH. As SSTHRESH decreases, the CWND size also decreases. Therefore, when the long flow finally encounters a situation in which the network is temporarily clear of short flows that would interrupt it, the long flow is unable to take advantage of the available network resources and must increase its CWND size through AIMD, which is quite slow. In other words, inefficient rate control policy prevents pFabric from achieving better performance. Therefore, a key insight of this paper is that rate control is completely unnecessary in order to achieve congestion control, and in fact amplifies the deleterious effects of starvation on long flows. Rather, Turbo abandons rate control entirely and simply performs congestion control using priorities alone.

4 Turbo Design

The basic principle of Turbo is that all flows should send at line rate all the time, setting priorities in such a way that priority queueing will be sufficient to prevent congestion collapse. Turbo has the following characteristics:

(1) Flows always send at line rate while they have data to send. This trivially guarantees perfect utilization.

(2) The only recovery is through timeouts. Duplicate ACKs are ignored (although TCP SACK is used), and there is no Fast Recovery or Slow Start.

(3) Packet priorities are set to the remaining flow size in order to approximate SRPT scheduling. Queues in the network implement flow-level priority dequeueing: after determining the packet with the highest priority, the packet in that flow with the least sequence number is dequeued. Queues also implement priority dropping, meaning that if a packet being enqueued causes the queue to overflow, the least priority packet is dropped.

(4) Since ACKs are so small (only 40 bytes), their interaction with data packets is minimal. It is however important to ensure that the signal that the network is clear is returned to a flow's source posthaste. As

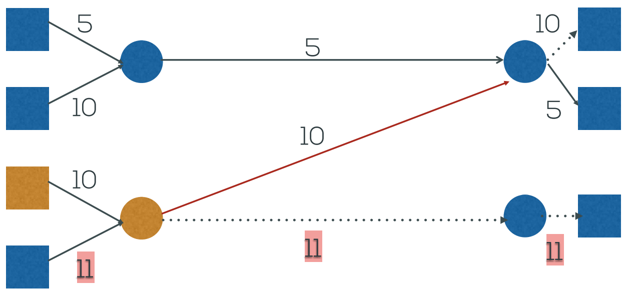


Figure 8: **Dead Packets scenario. The priority 11 flow is prevented from doing useful work, even though that is the optimal scheduling.**

a result, ACK packets are always sent at the highest priority.

For Turbo's predictability goals, we take advantage of the fact that using priority queueing means that an important packet with higher priority will suffer almost no queueing delay, since as soon as it is enqueued it will be the packet in the queue with the highest priority, and will thus be immediately dequeued. Therefore, this lack of queueing delay for high priority packets leads to both lower latency for such packets as well as more predictable performance. More predictable performance allows us to use tight timeouts, which speeds recovery. These benefits apply not only short flows with high priority but also long flows with low priority. Since packets from low priority flows will be dropped in encounters with high priority packets, tight timeouts can apply to low priority packets as well. Since queue sizes are small, the packet will only have gotten through if the network was completely clear, in which case it would suffer no queueing delay. In other words, it is far more likely for a given low priority packet to be dropped or delivered with little or no queueing delay than it is for such a packet to suffer large amounts of queueing delay before being dropped. This is especially true in the low-buffer scenarios for which Turbo is designed.

It is important to note that this guarantee of predictability requires priority queueing, since FIFO queues are highly unpredictable. Especially since most of the bytes in the network belong to the low priority long flows, it is probable that high priority packets from a short flow are enqueued behind low priority packets from a long flow on a regular basis. This would disastrous for the mean slowdown metric, which is especially sensitive to short flows since short flows have extremely low oracle flow completion times. Therefore, FIFO queues are unusable for our purposes.

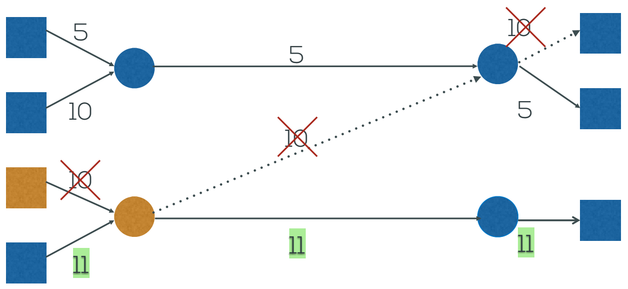


Figure 9: **Artificially inflating the priority of a flow on a timeout alleviates the problem.**

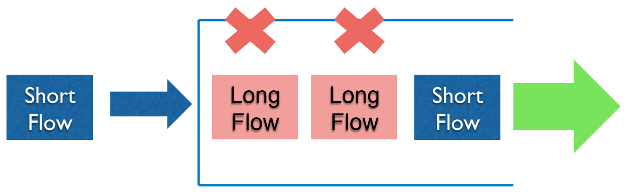


Figure 10: **The duplicate packets problem: packets from a lower priority long flow time out while stuck behind a higher priority short flow.**

4.1 Sources Of Inefficiency

There are two primary sources of inefficiencies in the use of Turbo. The first is zombie packets, which are a type of dead packet. Dead packets are packets that have consumed resources along the network up to a point, but are then dropped, letting the already-consumed resources go to waste. Zombie packets are dead packets that cause harm to other flows by blocking resources that they would have used. This is illustrated in figure 8. Initially, there are two flows of priority 10, A and B, and one flow of priority 11, C. The flow C's packets are dropped since they collide with packets from flow B. Then, a flow of priority 5, D, interrupts the transmission of both of the priority 10 flows A and B. One flow A is dropped near its source - this flow comprises of dead packets since its packets do not interfere with the transmission of any other flow. However, B, the other priority 10 flow, is dropped near its destination. Therefore, the flow C is still dropped, even though the flow blocking it is also being dropped. So, the packets of flow B are called zombie packets because they result in an inefficiency in flow scheduling.

To alleviate this situation, Turbo adjusts the priority of a flow that incurs a timeout to be inflated [1] above the usual remaining flow size based priority. To return to the example above now seen in figure 9, when flow D, with priority 5, preempts flows A and B, the end result

[1]Priority is set to the remaining flow size, so in this case a lower numerical priority is "better" - it will increase the packet's chances of being transmitted. Hence, an "inflated" priority is one that is worse than it would normally be.

for flow A is the same; it is dropped near its source and doesn't consume network resources. For flow B, however, the inflated priority will make flow C be preferable to flow B. Therefore, flow B will also be dropped near its source, and the network is able to make progress on both flows D and C. However, once flow D has finished blocking the paths of flows A and B, we want both these flows to rapidly resume sending at their real priority; this is unlikely if the flow is sending with artificially bad priority. Therefore, to aid in quick recovery, flows that have suffered a timeout and sending at inflated priority will periodically send out probe packets, which are very small so as to barely consume any network bandwidth, and which carry the real priority of the flow to determine whether the path is clear. If a probe successfully gets through, a flow immediately resets its priority to its true value.

The second primary source of inefficiency is the duplicated packets problem, as illustrated in 10. This stems from the fact that in certain cases, it is possible for a small number of packets to get stuck in a queue while packets from higher priority flows are dequeued. Due to the tight timeouts used, this leads to a timeout for that flow, even though its packets have not been dropped. This timeout leads to the flow needlessly retransmitting the stuck packets and thus wasting network bandwidth. This problem is both rare enough that simply increasing the timeout value would have negative effects, and harmful enough to flow performance that it should be addressed. In a FIFO queue, this problem is not present because there is inherently a guarantee that packets will move forward in the queue and eventually be dequeued, which makes it possible to reason about queueing delays and set overall timeouts appropriately. Unfortunately, FIFO queues are undesirable for our purposes as discussed above; in order to guarantee the highest level of performance for short flows, priority queues must be used. A solution to the duplicate packets problem in priority queues remains an open problem.

4.2 Design Options

There are three main parameters available in the Turbo protocol for addressing these inefficiencies. These are the amount of inflation a flow incurs on a timeout, the sending frequency of probe packets while in the inflated state, and finally the policy of how to set timeouts. There are a number of logical choices available for these parameters, and we discuss them below. Discussion of their performance is left to Section 5.

In the first option, timeouts are set for groups of packets. If a new ack is received, the timeout is updated to be calculated from the sending time of the next packet sent (which the same as the time at which the ack was received assuming no host delay). If a timeout occurs, the timeout is updated similarly. This scheme is called "**simple-timeouts**". Meanwhile, probes are sent whenever a timeout occurs. On a timeout, a flow inflates its priority under the scheme called "**cliff-inflation**". A flow using cliff inflation will, after a timeout occurs, increase its priority to be (INT_MAX/2 + remaining flow size). This way, two classes of flow priorities are created: the first consists of flows that have not incurred timeouts and are sending at their true priority: their remaining flow size. The other class of flows has a much higher priority, since INT_MAX/2 is used as the base priority. Therefore, this second class of flows does not cause any drops to flows in the first class. Furthermore, flows within the second class can also be differentiated since their priority is still a function of the same remaining flow size variable.

As an extension of cliff inflation, we also found it worthwhile to experiment with an extension of this concept by implementing a version of Turbo that ceases all sending of data packets upon a timeout. This extra conservative "**stop-on-timeout**" is our second option which still uses probes on each timeout to determine whether to start up again.

An alternative to the simple timeouts scheme is to use the cliff inflation and probing schemes with "Per Packet Timeouts" instead. In the per packet timeout scheme, a timeout is maintained for every packet that is sent. If an ack for that packet is received, the timeout is cancelled and the timeout on the next sent packet becomes active. If a timeout occurs, the packet in question is retransmitted, along with any other packets that are not acked and not already in flight, or in other words sent but not yet timed out or acked. This per packet timeout scheme is meant to take advantage of the tight timeouts made possible by Turbo. This may result in every dropped packet resulting in a probe being sent, resulting in a large number of probes being sent by flows in the inflated state. Even though probe packets are extremely small and consume minuscule amounts of network bandwidth, the possibility remains that they can contribute to the dad packets problem. Therefore, we add a guard that the probes are sent at least one retransmission timeout apart. This becomes our third option.

5 Evaluation

Our goal in the evaluation is to understand how the different approaches outlined in Section 4 perform. The metrics we are interested in are mean slowdown, which is a measure of latency since it is dominated by the performance of short flows, as well as average FCT, which is a measure of throughput since it is dominated by the performance of the long flows. This is because of the inherent heavy-tailed nature of the datacenter network work-

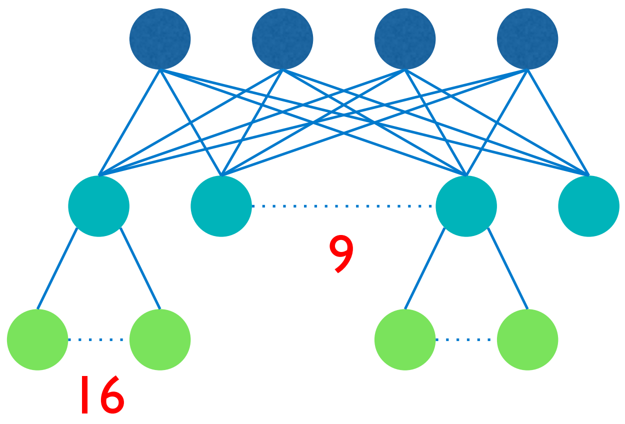


Figure 11: **The topoplogy used in our experiments.**

load, where most of the flows are small while most of the bytes reside in the long flows.

5.1 Methodology

5.1.1 Topology

Our experiments were performed on the same topology used in the evaluation of pFabric (seen in figure 11 - there are 144 hosts connected in groups of 16 to 9 top of rack (tor) switches and 4 core switches in a full mesh topology as shown in Figure 11. The bandwidth between the hosts and the tor switches was 10 Gbps, and the bandwidth in the core was 40 Gbps. Finally, the link latency was 0.2 microseconds. This means that the BDP of the network is $(0.2 \times 10^{-6} s \times 10 \times 10^{9} \frac{\text{bits}}{s}) \approx 2$ packets. The RTT ≈ 4 microseconds: $(0.2) \times 4 + \frac{1460*8 \text{ bits}}{10 \times 10^{3} \text{ bits per } \mu s} \times 2 + \frac{1460*8 \text{ bits}}{40 \times 10^{3} \text{ bits per } \mu s} \times 2 \approx 4 \mu s$.

5.1.2 Simulators

We ran our experiments in both ns2 [22] as well as our own simulator designed to be faster and more flexible than ns2 as well as handle experiments ns2 could not, for example very large experiments which took prohibitively lengthy amounts of time to complete in ns2. Our packet simulator is event based, and is implemented in approximately 2500 lines of C++ code.

5.1.3 Workloads

As mentioned before, we use three datacenter workloads namely **Bing, Aditya, and DataMining** for our evaluations. We assume that the inter-arrival times are distributed as per a poisson distributed whose parameter is set to achieve a desired network load. This arrival process is per source-destination pair. The **DataMining** workload is the most heavy tailed of the three workloads.

5.2 Performance

We were interested in the two metrics: average flow completion time and mean slowdown. To better understand why one proposal works better than the other, we seek to also understand the causes for inefficiency – fraction of dead packets and duplicate packets in the network. The latter two metrics give insight into starvation of flows and the stability of the network. The measurement of average flow completion time and mean slowdown were computed naturally as part of the simulation using the flow completion time of each flow in the simulation.

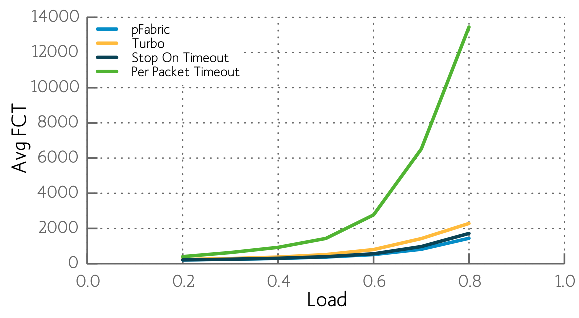
The dead packets percentage was calculated by measuring the difference between the number of bytes sent by hosts to the top-of-racks and the number of bytes received by hosts from top-of-racks. If there were no dead packets in the network, these two measurements would be identical, since all packets sent by hosts would be received by other hosts. However, in a dead packets scenario, the number of bytes received by hosts is less than the number of bytes sent. The dead packets percentage is computed by dividing this difference by the total amount of goodput, or, in other words, the sum of the size of all the flows.

The duplicated packets number was counted at each host. Whenever a host received a packet that it had already received, it incremented its duplicated packet counter. The overall duplicated packets percentage is taken to be the sum of the duplicated packet counts across all hosts divided by the goodput of the simulation. Since ACK packets are sent at the highest priority, it is extremely unlikely that an ACK is lost in the network; therefore, any duplicated packet counted by our methodology is almost certainly caused by a failure of the network as discussed in section 4.

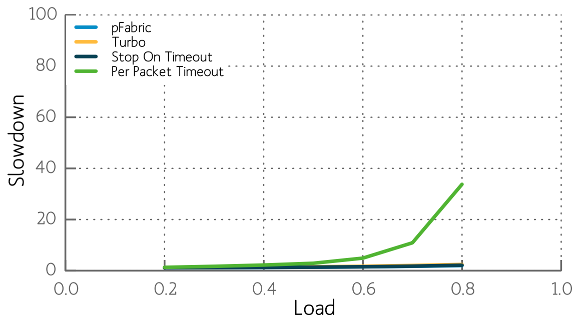
On the Aditya and Bing workloads, we ran experiments using 200,000 flows at 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, and 0.8 load for each of the 4 options presented above, as well as with pFabric for comparison. For the parameters in the pFabric experiment, we used the values suggested in the paper give our topology. Specifically, we used an initial CWND of one BDP number of packets, or 4. We used a maximum CWND of 7 packets, equivalent to twice the BDP. The buffer size was set to 12000 bytes, or 8 packets, which is pFabric's suggested 2× BDP, and the timeout was 15 μ s, or approximately 3× RTT.

For the parameters for the Turbo options, we used a timeout of 9.5 μ s and a buffer size of 6200 bytes, or approximately 4 packets (equal to the BDP). Since rate control is abandoned in Turbo, the CWND value was irrelevant and therefore not set.

When evaluating pFabric, we found that pFabric does indeed starve long flows, especially when the length of the simulation is increased. In fact, for very long simu-

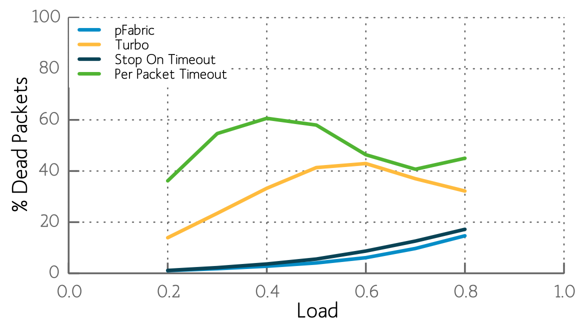


(a) CDF of flow sizes in packets

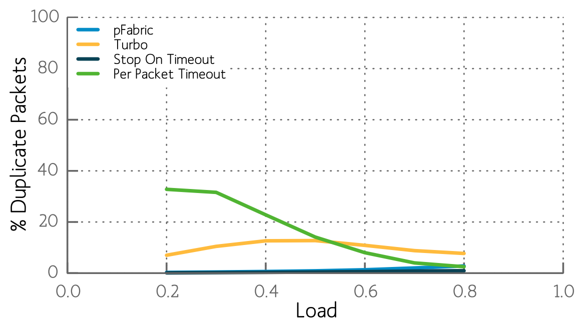


(b) CDF of total bytes

Figure 12: **CDF of flow sizes and of total bytes from three different datacenter workloads**



(a) CDF of flow sizes in packets



(b) CDF of total bytes

Figure 13: **CDF of flow sizes and of total bytes from three different datacenter workloads**

lations (more than 10 million flows), the pFabric simulations do not even finish due to the huge amount of starvation for the long flows. In other words, the abysmal performance of the long flows is not isolated to those flows alone, but rather causes the network as a whole to grind to a halt. Note that we observed that the edge-utilization in such cases was indeed 80%, so it was clearly the excessive backing off of large flows that was causing the problem.

Figures 12a and 13 illustrate the performance of the three schemes compared against pFabric.

Turbo still experiences a large number of dead-packets as shown in Figure 13a. Our priority back-off scheme should ideally lower the dead-packet number substantially, but we are still investigating as to why the numbers look so worse. Further, as discussed before, the timeout-setting becomes a challenging problem when the network does priority scheduling and improper values can result in a large number of duplicated packets. We observed that the number of duplicated packets was substantially high in the per-packet timeout scheme as shown in Figure 13b. We are currently investigating two options to alleviate this problem, one where the timeouts themselves are dependent on the priority of the flow, and second where the the switch maintains a TTL per packet and drops it if it is present too long in the queue.

However, the good news is that we were able to observe the trend that if we are able to lower the dead-packet and duplicated packet number, and thus, we believe that through a careful protocol design, we should be able to achieve extremely good performance.

6 Conclusion

While majority of the datacenter workloads are heavy-tailed , current datacenter network protocols, using primarily rate based congestion control algorithms, perform poorly with significant starvation problem and long queueing delays. We therefore proposed Turbo, a congestion control algorithm that is designed under low BDP scenarios and solely depends on packets priority information instead of flow rate assignments. Currently, there are two inefficiencies of Turbo coming from the dead packet and the duplicated problems that both contributes to wasted resource in the network. Our future goal for Turbo is to effectively and efficiently eliminate these inefficiencies.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prab-

hakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In ACM SIGCOMM, 2010.

[2] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In ACM IMC, 2010.

[3] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In Usenix HotOS, 2011.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In ACM SIGCOMM, 2013.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In ACM SIGCOMM, 2008.

[6] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In ACM SIGCOMM, 2009.

[7] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. Eyeq: Practical network performance isolation at the edge. In Usenix NSDI, 2013.

[8] Nathan Farrington and Andreyev Alexey. Facebook's data center network architecture. IEEE Optical Interconnects Conference, 2013.

[9] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In ACM SIGCOMM, 2013.

[10] Arista 7300 series switches. `http://www.arista.com/en/products/7300-series/7300-specifications`.

[11] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In Usenix OSDI, 2012.

[12] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In ACM SIGCOMM, 2009.

[13] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. NSDI, 2013.

[14] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, In ACM CoNEXT, 2011.

[15] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In Usenix NSDI, 2010.

[16] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In ACM SIGCOMM, 2012.

[17] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: meeting deadlines in datacenter networks. In ACM SIGCOMM, 2011.

[18] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In ACM SIGCOMM, 2012.

[19] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In ACM SOCC, 2013.

[20] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. Sonic: Precise realtime software access and control of wired networks. In USENIX NSDI, 2013.

[21] Sivasankar Radhakrishnan, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. Nicpic: Scalable and accurate end-host rate limiting. In Usenix HotCloud, 2013.

[22] The Network Simulator - ns2. `http://www.isi.edu/nsnam/ns/`.