Better Never than Late: Meeting Deadlines in Datacenter Networks

Christopher Wilson, Hitesh Ballani, Thomas Karagiannis and Ant Rowstron
Microsoft Research, Cambridge

Technical Report
MSR-TR-2011-66

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1. INTRODUCTION

The proliferation of datacenters over the past few years has been primarily driven by the rapid emergence of user-facing online services. Web search, retail, advertisement, social networking and recommendation systems represent a few prominent examples of such services.

While very different in functionality, these services share a couple of common underlying themes. First is their *soft real time* nature resulting from the need to serve users in a timely fashion. Consequently, today's online services have service level agreements (SLAs) baked into their operation [1–3]. User requests are to be satisfied within a specified latency target; when the time expires, responses, irrespective of their completeness, are shipped out. However, the completeness of the responses directly governs their quality and in turn, operator revenue [4]. Second, the mantra of horizontal scalability entails that online services have a *partition-aggregate* workflow with user requests being partitioned amongst (multiple) layers of workers whose results are then aggregated to form the response [1].

The combination of latency targets for datacenter applications and their distributed workflow has implications for traffic inside the datacenter. Application latency targets cascade to targets for workers at each layer; targets in the region of 10 to 100ms are common [5], which in turn, yield targets for network communication between the workers. Specifically, for any network flow initiated by these workers, there is an associated *deadline*. The flow is useful and contributes to the *application throughput* if, and only if, it completes within the deadline.

Today's datacenter networks, given their Internet origins, are oblivious to any such implications of the application design. Specifically, the congestion control (TCP) and flow scheduling mechanisms (FIFO queuing) used in datacenters are unaware of flow deadlines and hence, strive to optimize network-level metrics: maximize network throughput while achieving fairness. This mismatch can severely impact application performance; this is best illustrated through a couple of simple examples:

– *Case for unfair sharing:* Consider two flows that share a bottleneck link; one flow has a tighter deadline than the other. As shown in figure 1, with today's setup, TCP strives for fairness and the flows finish at similar times.[1] However, only one flow makes its deadline and is included in the user response. Apart from hurting application performance, this wastes valuable network resources on a non-contributing flow. Alternatively, given explicit information about flow deadlines, the network can distribute bandwidth unequally to meet the deadlines.

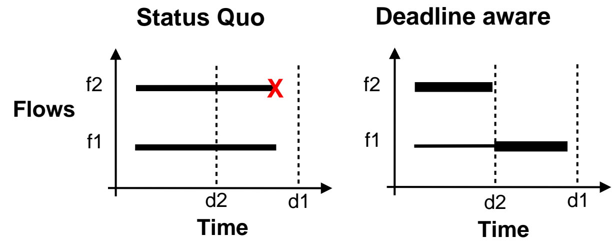


Figure 1: Two flows (f1, f2) with different deadlines (d1, d2). The thickness of a flow line represents the rate allocated to it. Awareness of deadlines can be used to ensure they are met.

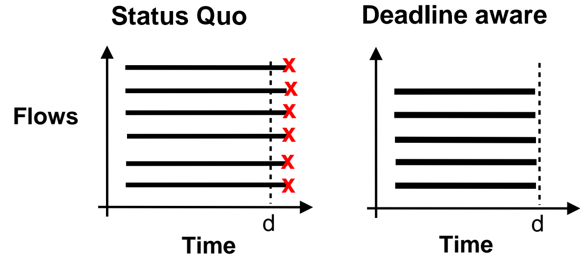


Figure 2: Multiple flows with the same deadline (d). The bottleneck capacity cannot satisfy the deadline if all six flows proceed. Quenching one flow ensures that the others finish before the deadline.

– *Case for flow quenching:* Consider a common application setting involving multiple servers responding to an aggregator simultaneously. The resulting network flows share a bottleneck link and have the same deadline. Further, assume that congestion on the bottleneck link is such that the aggregate capacity available to these flows is not enough to finish all the flows before the deadline. Figure 2 shows that with today's setup, all flows will receive their fair share of the bandwidth, finish at similar times and hence, miss the deadline. This, in turn, results in an empty response to the end user. Given flow deadlines, it may be possible to determine that the network is congested and quench some flows to ensure that the remaining flows do meet the deadline.

Both these examples reflect a tension between the functionality offered by a deadline agnostic network and application goals. This tension is above and beyond the known deficiencies of TCP in the datacenter environment. These include the incast problem resulting from bursts of flows [7,8], and the queuing and buffer pressure induced by a traffic mix that includes long flows [5].

However, the goal of a deadline-aware datacenter network poses unique challenges:

1. Deadlines are associated with flows, not packets. All

[1] We use TCP as a running example since it is used in datacenters today. However, our arguments apply to other TCP variants and proposals like DCTCP [5], XCP [6], etc.

packets of a flow need to arrive before the deadline.

2. Deadlines for flows can vary significantly. For example, online services like Bing & Google include flows with a continuum of deadlines (including some that do not have a deadline). Further, datacenters host multiple services with diverse traffic patterns. This, combined with the previous challenge, *rules out traditional scheduling solutions, such as simple prioritization of flows based on their length and deadlines (EDF scheduling [9])*.

3. Most flows are very short (<50KB), and, at the same time, RTTs minimal ($\approx 300\mu sec$). Consequently, reaction time-scales are short, and centralized, or heavy weight mechanisms to reserve bandwidth for flows are impractical.

In this paper, we present D^3, a Deadline-Driven Delivery control protocol, that addresses the aforementioned challenges. Inspired by proposals to manage network congestion through explicit rate control [6,10], D^3 explores the feasibility of exploiting deadline information to control the rate at which endhosts introduce traffic in the network. Specifically, applications expose the flow deadline and size information at flow initiation time. Endhosts use this information to request rates from routers along the data path to the destination. Routers thus allocate sending rates to flows to greedily satisfy as many deadlines as possible.

Despite the fact that flows get assigned different rates, instead of fair share, D^3 *does not* require routers to maintain per-flow state. By capitalizing on the trusted nature of the datacenter environment, both the state regarding flow sending rates and rate policing are delegated to endhosts. Routers only maintain simple aggregate counters. Further, our design ensures that rate assignments behave like "leases" instead of reservations, and are thus unaffected by host or router failures.

To this effect, this paper makes three main contributions:

- We present the case for utilizing flow deadline information to apportion bandwidth in datacenters.

- We present the design, implementation and evaluation of D^3, a congestion control protocol that makes datacenter networks deadline aware. Results from our testbed deployment show that D^3 can effectively double the peak load that a datacenter can support.

- We show that apart from being deadline-aware, D^3 performs well as a congestion control protocol for datacenter networks in its own right. Even without any deadline information, D^3 outperforms TCP in supporting the mix of short and long flows observed in datacenter networks.

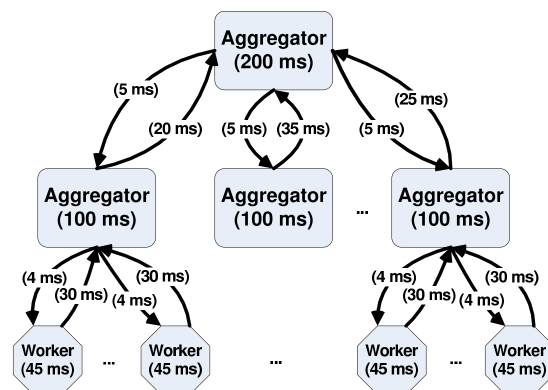


Figure 3: An example of the partition aggregate model with the associated component deadlines in the parentheses.

While we are convinced of the benefits of tailoring datacenter network design to the soft real time nature of datacenter applications, we also realize that the design space for such a deadline-aware network is vast. There exists a large body of work for satisfying application demands in the Internet. While we discuss these proposals in Section 3, the D^3 design presented here is heavily shaped by the peculiarities of the datacenter environment; its challenges (lots of very short flows, tiny RTTs, etc.) and luxuries (trusted environment, limited legacy concerns, etc.). We believe that D^3 represents a good first step towards a datacenter network stack that is optimized for application requirements and not a retrofitted Internet design.

2. BACKGROUND: TODAY'S DATACENTERS

otherIn this section, we provide a characterization of today's datacenters, highlighting specific features that influence D^3 design.

2.1 Datacenter applications

Partition-aggregate. Today's large-scale, user facing web applications achieve horizontal scalability by partitioning the task of responding to users amongst worker machines (possibly at multiple layers). This partition-aggregate structure is shown in Figure 3, and applies to many web applications like search [5], social networks [11], and recommendation systems [1]. Even data processing services like MapReduce [12] and Dryad [13] follow this model.

Application deadlines. The interactive nature of web applications means that latency is key. Customer studies guide the time in which users need to be responded to [14], and after accounting for wide-area network and rendering delays, applications typically have an SLA of 200-300ms to complete their operation and ship out the

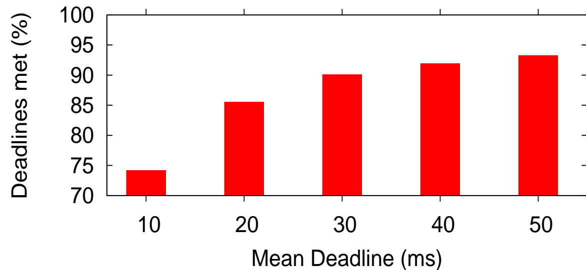


Figure 4: Deadlines met (%) based on flow completion times from a production datacenter.

user reply [1]. These application SLAs lead to SLAs for workers at each layer of the partition aggregate hierarchy. The worker SLAs mean that network flows carrying queries and responses to and from the workers have deadlines (see Figure 3). Any network flow that does not complete by its deadline is not included in the response and typically hurts the response quality, not to mention the wasted network bandwidth. Ultimately, this affects operator revenue; for example, an added latency of 100ms costs Amazon 1% of sales [4]. This leads to **Observation 1:** *Network flows generated by applications have deadlines and these deadlines are (implicitly) baked into all stages of the application operation.*

Deadline variability. Worker processing times can vary significantly. For instance, with search, workers operate on different parts of the index, execute different search algorithms, return varying number of response items, etc. This translates to varying flow deadlines. Datacenter traffic itself further contributes to deadline variability. We mentioned the short *query* and *response* flows between workers. Beyond this, datacenter traffic comprises of time sensitive, *short messages* (50KB to 1MB) that update the control state at the workers and long *background* flows (5KB to 50MB) that move fresh data to the workers [5]. Of course, the precise traffic pattern varies across datacenters. A Google datacenter running MapReduce jobs will have primarily long flows, some of which may carry deadlines. Hence, **Observation 2:** *Today's datacenters have a diverse mix of flows with widely varying deadlines. Some flows, like background flows, may not even have deadlines.*

Missed deadlines. To quantify the prevelance of missed deadlines in today's datacenters, we use measurements of flow completion times in datacenters [5]. We were unable to obtain flow deadline information, and resort to capturing their high variability by assuming that deadlines are exponentially distributed. Figure 4 shows the percentage of flows that meet their deadlines with varying mean for the deadline distribution. We find that even with lax deadlines (mean=40ms), more than 7% of flows miss their deadline. Our evaluation (section 6) shows that this results from network inefficiencies. Application designers struggle to cope with the impact of such inefficiencies [5], and this might mean artificially limiting the peak load a datacenter can support so that application SLAs are met. Hence, **Observation 3:** *A significant fraction of flow deadlines are missed in today's datacenters.*

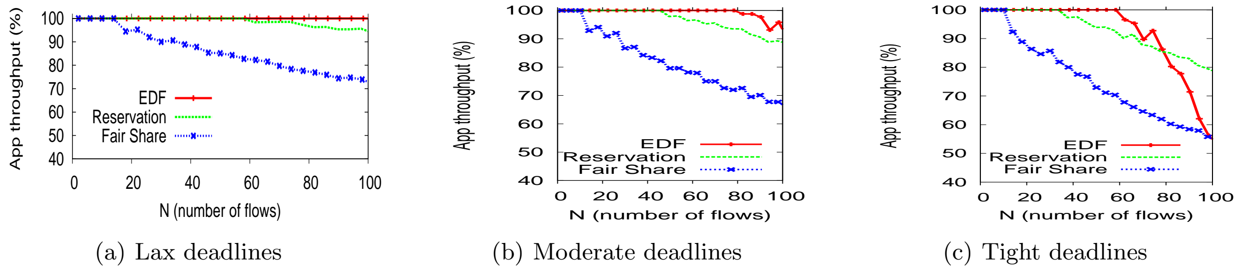
Flow sizes. Since deadlines are associated with flows, all packets of a flow need to arrive before its deadline. Thus, a-priori knowledge of the flow size is important. Indeed, for almost all the interactive web applications today, the size of network flows initiated by workers and aggregators is known in advance. As a specific example, in web search, queries to workers are fixed in size while responses essentially include the top-k matching index records (where k is specified in the query). Thus, the size of the response flow is known to the application code at the worker even before it begins processing. The same holds for many other building block services like key-value stores [1,11], data processing [12,13], etc. Even for applications where this condition does not hold, the application designer can typically provide a good estimate of the expected flow sizes. Hence, this leads to **Observation 4:** *Web applications have knowledge of the flow size at flow initiation time.*

2.2 TCP in Datacenters

The problems resulting from the use of TCP in datacenters are well documented [7,8]. Bursts of concurrent flows that are all too common with the partition aggregate application structure can cause a severe drop in network throughput (incast). Beyond this, the presence of long flows, when combined with TCP's tendency to drive queues to losses, hurts the latency of query-response flows. These problems have placed artificial limitations on application design, with designers resorting to modifying application workflow to address the problems [5].[2]

Motivated by these issues, recent proposals have developed novel congestion control protocols or even moved to UDP to address the problems [5,15]. These protocols aim to ensure: (i) low latency for short flows, even in the face of bursts, and (ii) good utilization for long flows. We concur that this should be the baseline for any new datacenter transport protocol and aim to achieve these goals. *We further argue that these minimum requirements should ideally be combined with the ability to ensure that the largest possible fraction of flows meet their deadlines.* Finally, many application level solutions (like SEDA [16]) deal with variable application load. However, they are not well suited for network/transport problems since the datacenter network is shared amongst multiple applications. Further, ap-

[2]Restrictions on the fan-out factor for aggregators and the size of the response flows from workers to aggregator are a couple of examples of the limitations imposed.



(a) Lax deadlines (b) Moderate deadlines (c) Tight deadlines

Figure 5: Application throughput with varying number of flows across a single bottleneck link. Confidence intervals are within 1% of the presented values.

plications do not have visibility into the network state (congestion, failures, etc.). On the contrary, a deadline-aware network can complement load-aware applications by explicitly honoring their demands.

3. DESIGN SPACE AND MOTIVATION

The past work for satisfying application deadlines in the Internet can be categorized in two broad classes. The first class of solutions involve packet scheduling in the network based on deadlines. An example of such scheduling is Earliest Deadline First (EDF) [9] wherein routers prioritize packets based on their per-hop deadlines. EDF is an optimal scheduling discipline in that if a set of flow deadlines can be satisfied under any discipline, EDF can satisfy them too. The second solution category involves rate reservations. A deadline flow with size s and deadline d can be satisfied by reserving rate $r = \frac{s}{d}$. Rate reservation mechanisms have been extensively studied. For example, ATM supported Constant Bit Rate (CBR) traffic. In packet switched networks, efforts in both industry (IntServ, DiffServ) and academia [17,18] have explored mechanisms to reserve bandwidth or at least, guarantee performance.

Value of deadline awareness. Given this existing body of work, we attempt here through simple Monte Carlo simulations, to build some intuition regarding the (possible) benefits of these approaches over fair sharing used in datacenters today.

Consider a 1Gbps link carrying several flows with varying deadlines. Flow parameters (such as the size and the fraction of short and long flows) are chosen to be consistent with typical datacenters [5]. For the flows with deadlines, the deadlines are chosen to be exponentially distributed around 20ms (tight), 30ms (moderate) and 40ms (lax). To capture the lack of application value of flows that miss their deadline, we use *application throughput* or the number of flows that meet their deadline as the performance metric of interest.

Using this simple simulation setup, we evaluate three "ideal" bandwidth allocation schemes: (i) *Fair-share*, where the link bandwidth is allocated evenly amongst all current flows and represents the best-case scenario for today's deadline agnostic protocols like TCP, DCTCP [5], XCP [6], etc. (ii) *EDF*, representing the first broad class of deadline aware solutions, where the flow with the earliest deadline receives all the bandwidth until it finishes, and (iii) *Rate reservation* (i.e., the second category), where flows, in the order of their arrival, reserve the rate needed to meet their deadline. In contrast to fair-share, the latter two approaches are deadline aware.

Figure 5 shows the application throughput for the three approaches. *As the number of flows increases, deadline-aware approaches significantly outperform fair sharing*. Perhaps most important is the fact that they can support *three to five times* as many flows as fair share without missing any deadline (application throughput=100%). This, in effect, redefines the peak loads at which a datacenter can operate without impacting the user experience.

Hence, deadline-aware approaches have a lot to offer towards datacenter performance. However, practical challenges remain for both types of solutions, scheduling as well as reservations.

For the former class, we use EDF as an example to explain its limitations, though our arguments are general. EDF is packet based. It works on per-hop packet deadlines while datacenter applications have end-to-end flow deadlines. As a result, even though EDF is optimal when the deadlines can be satisfied, when there is congestion, EDF can actually drive the network towards congestive collapse (see figure 5(c)). Second and perhaps more importantly, EDF still needs to be complemented by an endhost rate control design that will ensure that routers have the right packets to schedule. Designing such a distributed rate control scheme is far from trivial. Finally, EDF requires priority queuing at routers. Our testbed experiments in Section 6 illustrate some of these limitations for a simple priority scheme.

For the latter class, reservation schemes are too heavy weight for the datacenter environment where most flows are short. Further, unlike real-time traffic on the Internet, datacenter flows do not require a "constant" rate. Reservation schemes ignore this flexibility and reduce network efficiency, especially given the dynamics on dat-

acenter networks, where network conditions change very fast (e.g., tiny RTTs, large bursts of short flows).

Overall, these limitations motivate the need for a practical datacenter congestion control protocol that, on the one hand, ensures flows meet their deadlines, but, on the other, avoids packet scheduling and explicit reservations.

4. D^3 DESIGN

The discussion in the two previous sections leads to the following goals for datacenter congestion control:

1. *Maximize application throughput*: The protocol should strive to maximize the number of flows that satisfy their deadlines and hence, contribute to application throughput.
2. *Burst tolerance*: Application workflows often lead to flow bursts, and the network should be able to accommodate these.
3. *High utilization*: For flows without deadlines, the protocol should maximize network throughput and achieve high utilization.

D^3 is designed to achieve these goals. Beyond these explicit goals, D^3 accounts for the luxuries and challenges of the datacenter environment. For instance, an important luxury is the fact that the datacenter is a homogenous environment owned by a single entity. Consequently, incremental deployment, backwards compatibility, and being friendly to legacy protocols are *non-goals*.

The key insight guiding D^3 design is the following: given a flow's size and deadline, one can determine the rate needed to satisfy the flow deadline. Endhosts can thus ask the network for the required rate. There already exist protocols for explicit rate control wherein routers assign sending rates to endhosts [6,10]. With D^3, we extend these schemes to assign flows with rates based on their deadlines, instead of the fair share.

4.1 Rate control

With D^3, applications expose the size and deadline information when initiating a deadline flow. The source endhost uses this to request a *desired rate*, r. Given a flow of size s and deadline d, the initial desired rate is given by $r = \frac{s}{d}$. This rate request, carried in the packet header, traverses the routers along the path to the destination. Each router assigns an *allocated rate* that is fed back to the source through the acknowledgement packet on the reverse path. The source thus receives a vector of allocated rates, one for each router along the path. The sending rate is the minimum of the allocated rates. The source sends data at this rate for a RTT while piggybacking a rate request for the next RTT on one of the data packets.

Note however that neither does a flow need, nor does it obtain a reservation for a specific sending rate throughout its duration. The rate that the network can offer varies with traffic load and hence, each source must periodically (in our case, every RTT) ask the network for a new allocation. Since the actual rate allocated by the network can be more or less than the desired rate, endhosts update the desired rate as the flow progresses based on the deadline and the remaining flow size.

4.2 Rate allocation

For each of their outgoing interfaces, routers receive rate requests from flows with deadlines. Beyond this, there are flows without deadlines, where $r = 0$. Hence, the *rate allocation problem* is defined as: Given rate requests, a router needs to allocate rates to flows so as to maximize the number of deadlines satisfied (goal 1) and fully utilize the network capacity (goal 3). In a dynamic setting, this rate allocation problem is NP-complete [19].

We adopt a greedy approach to allocate rates. When a router receives a rate request packet with desired rate r, it strives to assign at least r. If the router has spare capacity after satisfying rate requests for all deadline flows, it distributes the spare capacity fairly amongst all current flows. Hence, when the router capacity is more than the capacity needed to satisfy all deadline flows, *the allocated rate* a given to a flow is:

- For a deadline flow with desired rate r, $a = (r + fs)$, where fs is the fair share of the spare capacity after satisfying deadline flow requests.
- For a non-deadline flow, $a = fs$.

We note that distributing the spare capacity between deadline and non-deadline flows allows us to balance the competing goals 1 and 3. Assigning deadline flows with a rate greater than their desired rate ensures that their subsequent rate requests will be lower and the network will be able to satisfy future deadline flows. At the same time, assigning non-deadline flows with a share of the spare capacity ensures that they make progress and network utilization remains high.

However, in case the router does not have enough capacity to satisfy all deadline flows, it greedily tries to satisfy the rate requests for as many deadline flows as possible. The remaining flows, deadline and non-deadline, are assigned a ***base rate*** that allows them to send a header-only packet per RTT and hence, request rates in the future. For deadline flows, such low assignments will cause the desired rate to increase. The endhosts can thus decide whether to give up on flows based on an ever increasing desired rate. This is further discussed in Section 6.1.3.

4.3 Router operation

The rate allocation description above assumes the router has the rate requests for all flows at the same point in time. In reality, the router needs to make allocation decisions in an online, dynamic setting, i.e., rate requests are spread over time, and flows start and finish. To achieve this, the rate allocation operates in a slotted fashion (from the perspective of the endhosts). The rate allocated to a flow is valid for the next RTT, after which the flow must request again. A rate request at time t serves two purposes: (1). It requires the router to assign a_{t+1}, the allocated rate for the next RTT, and (2). It returns a_t, the allocation for the current RTT.

To achieve (1), the router needs to track its existing allocations. Consequently, routers maintain three simple, aggregate counters for each interface:

- N: number of flows traversing the interface. Routers use flow initiation and termination packets (TCP SYN/FIN) to increment and decrement N respectively.[3]
- Demand counter (D): sum of the desired rates for deadline flows. This represents the total demand imposed by flows with deadlines.
- Allocation counter (A): sum of allocated rates. This is the current total allocation.

To achieve (2), the router must know the current rate allocated to the flow. In a naive design, a router could maintain rate allocations for each active flow through it. However, since most deadline flows are very short, such an approach is too heavy-weight, not to mention router memory intensive. We avoid the need for per-flow state on routers by relying on endhosts to convey rate allocations for each flow. Specifically, each rate request packet, apart from the desired rate r_{t+1}, contains the rate requested in the previous interval (r_t) and a vector of the rates allocated in the previous interval ($[a_t]$). Each element in the vector corresponds to the rate allocated by a router along the path in the previous interval. The encoding of these in the rate request packet header is described in Section 5. For topologies with multiple paths between endhosts [20–23], we rely on ECMP and other existing mechanisms used with TCP to ensure that a given flow follows a single path.

Given this, we can now describe how packets are processed by routers. *Routers have no notion of flow RTTs.* Packets without rate request headers are forwarded just as today. Snippet 1 shows how a router processes a rate request packet. It applies to both deadline and non-deadline flows (for the latter, desired rate r_{t+1} is 0). The router first uses the packet header information to perform bookkeeping. This includes the flow returning its current allocation (line 3). Lines 7-13 implement the rate allocation scheme (Section 4.2) where the router calculates a_{t+1}, the rate to be allocated to the flow. The router adds this to the packet header and forwards the packet.

Snippet 1 Rate request processing at interval t

Packet contains: Desired rate r_{t+1}, and past information r_t and a_t. Link capacity is C.
Router calculates: Rate allocated to flow (a_{t+1}).

```
1: //For new flows only
2: if (new_flow_flag_set) N = N + 1
3: A = A−a_t //Return current allocation
4: D = D−r_t + r_{t+1} //Update demand counter
   //Calculate left capacity
5: left_capacity = C − A
   //Calculate fair share
6: fs = (C − D)/N

7: if left_capacity > r_{t+1} then
8:    //Enough capacity to satisfy request
9:    a_{t+1} = r_{t+1} + fs
10: else
11:    //Not enough capacity to satisfy request
12:    a_{t+1} = left_capacity
13: end if
   //Flows get at least base rate
14: a_{t+1} = max(a_{t+1}, base_rate)
   //Update allocation counter
15: A = A+ a_{t+1}
```

Of particular interest is the scenario where the router does not have enough capacity to satisfy a rate request (lines 11-12). This can occur in a couple of scenarios. First, the cumulative rate required by existing deadline flows, represented by the demand counter, may exceed the router capacity. In this case, the router simply satisfies as many requests as possible in the order of their arrival. In the second scenario, the demand does not exceed the capacity but fair share allocations to existing flows imply that when the rate request arrives, there is not enough spare capacity. However, the increased demand causes the fair share assigned to the subsequent rate requests to be reduced (line 6). Consequently, when the deadline flow in question requests for a rate in the next interval, the router should be able to satisfy the request.

4.4 Good utilization and low queuing

The rate given by a router to a flow is based on the assumption that the flow is bottlenecked at the router. In a multihop network, this may not be true. To account for bottlenecks that occur earlier along the path,

[3]Note that past rate control schemes [6,10] approximate N as C/R, where C is the interface capacity and R is the current rate being assigned to flows. Yet, D[3] does not assign the same rate to each flow and this approximation is not applicable.

a router ensures that its allocation is never more than that of the previous router. This information is available in the rate allocation vector being carried in the packet header. However, the flow may still be bottlenecked downstream from the router and may not be able to utilize its allocation.

Further, the veracity of the allocation counter maintained by a router depends on endhosts returning their allocations. When a flow ends, the final rate request packet carrying the FIN flag returns the flow's allocated rate. While endhosts are trusted to follow the protocol properly, failures and bugs do happen. This will cause the router to over-estimate the allocated rate, and, as a result, penalize the performance of active flows.

The aforementioned problems impact router utilization. On the other hand, a burst of new flows can cause the router to temporarily allocate more bandwidth than its capacity, which results in queuing. To account for all these cases, we borrow from [6,10] and periodically adjust the router capacity based on observed utilization and queuing as follows:

$$C(t+1) = C(t) + \alpha(C(t) - \frac{u(t)}{T}) - \beta(\frac{q}{T})$$

where, C is router capacity at time t, T is the update interval, u is bytes sent over the past interval, q is instantaneous queue size and α, β are chosen for stability and performance.[4]

Consequently, when there is under-utilization ($u/T < C$), the router compensates by allocating more total capacity in the next interval, while when there is queuing ($q(t) > 0$), the allocations reduce. Apart from addressing the downstream bottleneck problem, this ensures that the counters maintained by routers are soft state and divergence from reality does not impact correctness. The failure of endhosts and routers may cause flows to not return their allocation. However, the resulting drop in utilization drives up the capacity and hence, the allocation counters do not have to be consistent with reality. The router, during periods of low load, resets its counters to return to a consistent state. Even in an extreme worst case scenario, where bugs at endhosts lead to incorrect rate requests, this will only cause a degradation in the performance of the application in question, but will have no further effects in the operation of the router or the network.

4.5 Burst tolerance

Bursts of flows are common in datacenters. Such bursts are particularly challenging because of tiny RTTs in the datacenter. With a typical RTT of $300\mu s$, a new flow sending even just one 1500-byte packet per RTT equates to a send rate of 40Mbps! Most TCP implementations shipping today start with a send window of two packets and hence, a mere 12-13 new flows can cause queuing and packet loss on a 1Gbps link. This has been observed in past work [7,8] and is also present our experiments.

With D^3, a new flow starts with a rate request packet with the SYN flag set. Such a request causes N to be incremented and reduces the fair share for flows. However, pre-existing flows may already have been allocated a larger fair share rate (N was less when the earlier rate requests were processed). Hence, allocating each new flow with its proper fair share can cause the router to allocate an aggregate rate larger than its capacity, especially when a burst of new flows arrives suddenly.

We rely on D^3's ability to "pause" flows by assigning them the base rate to alleviate bursts. When a new flow starts, the fair share assigned to its rate request is set to base rate. For non-deadline flows, this effectively asks them to pause for a RTT and not send any data packets. The sender however does send a packet with only the rate request (i.e., a header-only packet) in the next RTT and the router assigns it with a fair share as normal. This implies that a new non-deadline flow does not make progress for an extra RTT at startup. However, such flows are typically long. Further, RTTs are minimal, and this approach trades-off a minor overhead in bandwidth and latency (one RTT $\sim 300\mu s$) for a lot of burst tolerance. Our evaluation shows that this vastly improves D^3's ability to cope with flow bursts over the state of the art. Additionally, this does not impact deadline flows much because the router still tries to honor their desired rate.

5. IMPLEMENTATION

We have created an endhost-based stack and a proof-of-concept router that support the D^3 protocol. This paper focuses on the congestion control aspects of D^3 but our implementation provides a complete transport protocol that provides reliable, in-order delivery of packets. As with TCP, reliability is achieved through sequence numbers for data packets, acknowledgements from receivers, timer-based retransmissions and flow control.

On endhosts, D^3 is exposed to applications through an extended Sockets-like API. The extensions allow applications to specify the flow length and deadline when a socket is created. The core logic for D^3, including the rate control scheme, runs in user space. We have a kernel driver that is bound to the Ethernet interface exposed by the NIC driver. The kernel driver efficiently marshals packets between NIC and the user-level stack.

The router is implemented on a server-grade PC and

[4]The α, β values are chosen according to the discussion in [10], where the stability of this controller was shown, and are set to 0.1 and 1 in our implementation. The update interval T should be larger than the datacenter propagation RTT; in our implementation it is set to $800\mu s$.

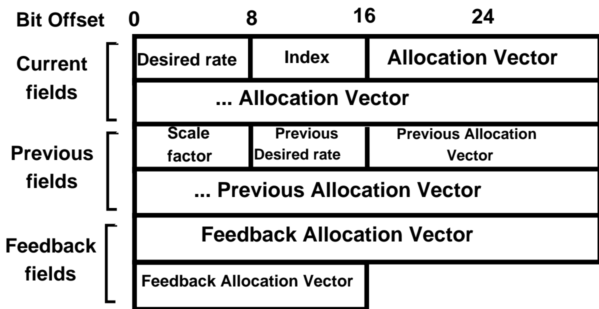


Figure 6: Congestion header for rate request and feedback packets.

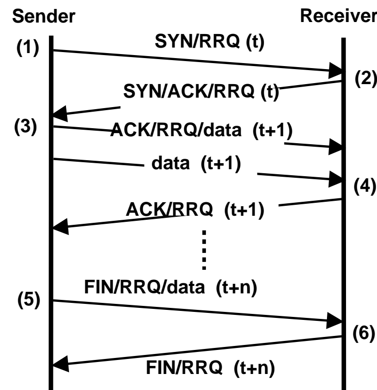


Figure 7: Packet exchange with D^3. RRQ is Rate Request. Text in parenthesis is the current RTT interval.

implements a shared buffer, store and forward architecture. To be consistent with shallow buffers in today's datacenters, the router has a buffer of 128KB per NIC. The router uses the same kernel driver as the endhosts, except the driver is bound to multiple Ethernet interfaces. All incoming packets pass to a user space process, which processes and forwards them on the appropriate interface. The design of the kernel driver and user-space application support zero-copy of packets.

Router overhead. To keep per-packet overhead low in the router, we use integer arithmetic for all rate calculations. Although each packet traverses the user-kernel space boundary, we are able to sustain four links at full duplex line rate. *Specifically, the average packet processing time was less than 1μs (0.208μs), and was indistinguishable from normal packet forwarding in user-space.* Thus, D^3 imposes minimal overhead on the forwarding path and the performance of our prototype leads us to believe that it is feasible to implement D^3 in a commodity router.

Packet header. The D^3 request and rate feedback packet header is shown in Figure 6. The congestion header includes the desired rate r_{t+1}, an index into the allocation vector and the current allocation vector ($[a_{t+1}]$). The header also includes the allocations for the previous RTT so that the routers can update their relevant counters - the desired rate r_t and the vector of rates allocated by the routers ($[a_t]$). Finally, the header carries rate feedback to the destination - a vector of rates allocated by the routers for reverse traffic from the destination to the source.

All rates are in Bytes/μs and hence, can be encoded in one byte; 1Gbps equates to a value of 125. The scale factor byte can be used to scale this and would allow encoding of much higher rates. The allocation vectors are 6 bytes long, allowing for a maximum network diameter of 6 routers (or switches). We note that current datacenters have three-tiered, tree-like topologies [20] with a maximum diameter of 5 switches. The allocation vectors could be made variable length fields to achieve extensibility. Overall, our current implementation imposes an overhead of 22 bytes for every packet carrying rate requests.

Protocol description. The protocol operation is illustrated in Figure 7. (1). The sender initiates the flow by sending a SYN packet with a rate request. Routers along the path allocate rate and add it to the current allocation vector in the congestion header. (2). Receivers respond with a SYN/ACK and a rate request of their own. The congestion header of the response includes the allocated rate vector for the sender. (3). The sender uses this vector to determine its sending rate and sends data packets. One of these includes a rate request for the next RTT. (5). Once the sender is done, it sends a FIN packet and returns its existing allocation.

Calculating desired rate. The sender uses information about flow deadline and remaining flow length to determine the desired rate that would allow the flow to meet its deadline. At interval t, the desired rate for the next RTT is given by

$$rt + 1 = \frac{remaining_flow_length - s_t * rtt}{deadline - 2 * rtt}$$

where s_t is the current sending rate, and rtt is the sender's current estimate of the RTT for the flow, which is based on an exponential moving average of the instantaneous RTT values. The numerator accounts for the fact that by the next RTT, the sender would have sent $s_t * rtt$ bytes worth of more data. The denominator is the remaining time to achieve the deadline: one rtt is subtracted since the rate will be received in the next RTT, while the second rtt accounts for the FIN exchange to terminate the flow.

Discretization overhead. D^3's endhost stack is ACK-clocked. On receiving an ACK with rate feedback, the sender calculates its new send rate and sends a burst of packets. However, the fact that it can only send an integer number of 1500-byte packets leads to discretization overhead. For instance, with a 300μs RTT and a send

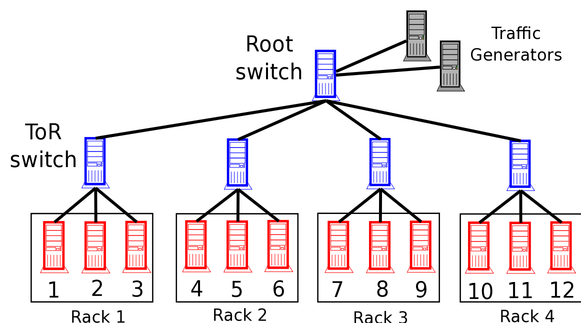


Figure 8: Testbed topology: red servers are endhosts, blue are switches, grey are traffic generators.

rate of 60Mbps, only one packet can be sent per RTT. This amounts to 40Mbps, so the sender does not use its complete allocation. To counter this, the sender keeps a running counter of unused allocation that is added to the next interval.

6. EVALUATION

We deployed D^3 across a small testbed structured like the multi-tier tree topologies used in today's datacenters. The testbed (Figure 8) includes twelve endhosts arranged across four racks. Each rack has a top-of-rack (ToR) switch, and the ToR switches are connected through a root switch. All endhosts and switches are Dell Precision T3500 servers with a quad core Intel Xeon 2.27GHz processor, 4GB RAM and 1 Gbps interfaces, running Windows Server 2008 R2. The root switch is further connected to two other servers that are used as traffic generators to model traffic from other parts of the datacenter. For two endhosts in the same rack communicating through the ToR switch, the propagation RTT, measured when there is no queuing, is roughly $500\mu s$. The endhosts are also connected to a dedicated 48-port 1Gbps NetGear GS748Tv3 switch (not shown in the figure). We use this for TCP experiments.

Our evaluation has two primary goals: (i). To determine the value of using flow deadline information to apportion network bandwidth. (ii). To evaluate the performance of D^3 just as congestion control protocol, without deadline information. This includes its queuing and utilization behavior, and performance in a multi-bottleneck, multi-hop setting.

6.1 Exploiting deadlines through D^3

To evaluate the benefit of exploiting deadline information, we compare D^3 against TCP. However, TCP is well known to not be amenable to datacenter traffic patterns. To capture the true value of deadline awareness, we also operate D^3 in fair share mode only, i.e., without any deadline information and all flows treated as non-deadline flows. We term this RCP_{dc} since it is effectively RCP optimized for the datacenter.[5] With RCP_{dc}, the fair share is explicitly communicated to the hosts (i.e., no probe-based exploration is required) and it has been shown to be optimal in terms minimizing flow completion times [10]. Hence, it represents the limit for any fair share protocol. We were unable to obtain the set of Windows kernel patches for implementing DCTCP; yet, as explained above, our RCP_{dc} implementation represents an upper limit for DCTCP's performance. We further contrast D^3 against deadline-based priority queuing of TCP flows. Priority queuing was implemented by replacing the Netgear switch with a CISCO router that offers port-based priority capabilities. Flows with short deadlines are mapped to high priority ports. Our evaluation covers the following scenarios:

- *Flow burst microbenchmarks.* This scenario reflects the case where a number of workers start flows at the same time towards the same destination. It provides a lower-bound on the expected performance gain as all flows compete at the same bottleneck at the same time. Our results show that D^3 can *support almost twice the number of workers*, without compromising deadlines, compared to RCP_{dc}, TCP and TCP with priority queuing (henceforth referred to as TCP_{pr}).

- *Benchmark traffic.* This scenario represents typical datacenter traffic patterns (e.g., flow arrivals, flow sizes) and is indicative of the expected D^3 performance with current datacenter settings. The evaluation highlights that D^3 offers an order of magnitude improvement over out of the box TCP and a factor of two improvement over an optimized TCP version and RCP_{dc}.

- *Flow quenching.* We evaluate the value of terminating “useless” flows that do not contribute to application throughput. Our results show that flow quenching ensures the D^3 performance degrades gracefully under extreme load.

6.1.1 Flow burst microbenchmarks

In this scenario, a host in each rack serves as an aggregator (see Figure 3) while other hosts in the rack represent workers responding to an aggregator query. This is a common application scenario for online services. All workers respond at the same time and all response flows are bottlenecked at the link from the rack's ToR switch to the aggregator. Since there are only three hosts in

[5]While the core functionality of RCP_{dc} mimics RCP, we have introduced several optimizations to exploit the trusted nature of datacenters. The most important of these include: exact estimates of the number of flows at the router (RCP uses algorithms to approximate this), the introduction of the *base rate*, the pause for one RTT to alleviate bursts, etc.

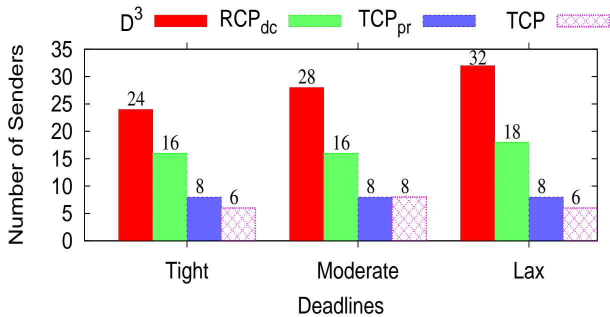


Figure 9: Number of concurrent senders that can be supported while ensuring more than 99% application throughput.

each rack, we use multiple workers per host. We also repeated the experiment on a restructured testbed with more hosts per rack and the results remained qualitatively the same.

The response flow lengths are uniformly distributed across [2KB, 50KB] and the flow deadlines are distributed exponentially around 20ms (tight), 30ms (moderate) and 40ms (lax). As described earlier, the primary performance metric is *application throughput*, that is, the number of flows finishing before their deadline. *This metric was intentionally chosen as datacenter operators are primarily interested in the "operating regime" where the network can satisfy almost all flow deadlines.* Hence, we vary the number of workers sending flows and across 200 runs of this experiment, determine the maximum number of concurrent senders a given congestion control scheme supports while ensuring at least 99% application throughput. This is shown in Figure 9.

As compared to RCP_{dc}, *D^3 can support almost twice as many concurrent senders while satisfying flow deadlines* (3-4 times as compared to TCP and TCP_{pr}). This is because D^3 uses flow deadline information to guide bandwidth apportioning. This is further pronounced for relaxed deadlines where D^3 has more flexibility and hence, the increase in the number of senders supported compared to tight deadlines is greater than for other approaches. Note that since congestion occurs at the aggregator's downlink, richer datacenter topologies like VL2 [20] or FatTree [21] cannot solve the problem.

For completeness, Figure 10 shows the application throughput as we vary the number of concurrent senders to 40. While the performance of these protocols beyond the regime of high application throughput may not be of primary operator importance, the figure does help us understand application throughput trends under severe (unplanned) load spikes. Apart from being able to support more senders while ensuring no deadlines are missed, when the number of senders does become too high for D^3 to satisfy all deadlines, it still improves application throughput by roughly 20% over TCP, and

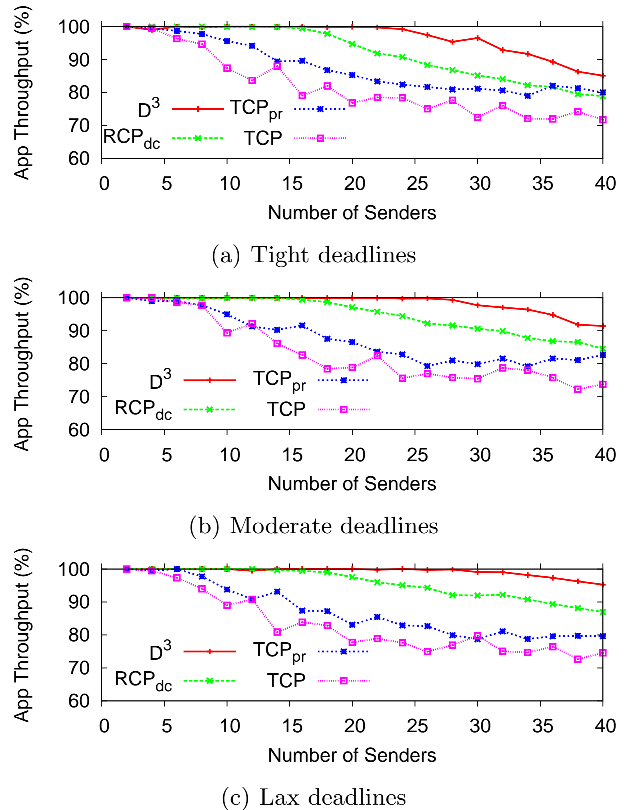


Figure 10: Application throughput for varying concurrent senders and tight, moderate and lax deadlines (the Y-axis starts at 60%).

10% or more, over RCP_{dc} and TCP_{pr}. Hence, even if we relax our "operating-regime" metric to be less demanding, for example, to 95% of application throughput, D^3 can support 36 senders with moderate deadlines compared to the 22, 10 and 8 senders of RCP_{dc}, TCP_{pr} and TCP respectively.

The figure also illustrates that RCP_{dc} outperforms TCP at high loads. This is because probe-based protocols like TCP attempt to discover their fair share by causing queues to build up and, eventually, losses, ergo increasing latency for a number of flows. Instead, RCP_{dc} avoids queueing by equally dividing the available capacity. Figure 11 highlights this point by displaying the scatter plots of flow completion times versus flow deadlines for TCP, RCP_{dc} and D^3 for one of the experiments (moderate deadlines, 14-30 senders). For TCP, it is evident that packet losses result in TCP timeouts and very high completion times for some flows. Since the hardware switch has a buffer of 100KB, a mere eight simultaneous senders with a send-window of eight full sized packets can lead to a loss; this is why TCP performance starts degrading around eight senders in Figure 10.[6]

[6]The mean flow size in our experiments is 26KB. Hence, a majority of flows will require more than 3 RTTs to complete,

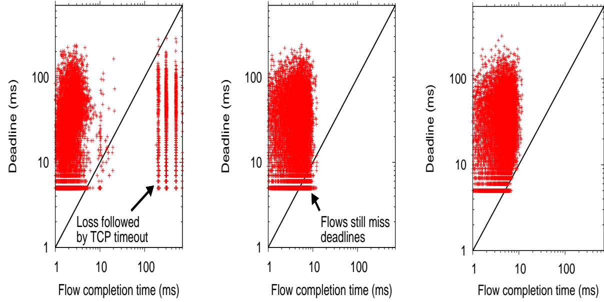


Figure 11: Scatter plot of flow completion times vs. deadlines for TCP (left), $\mathrm{RCP_{dc}}$ (middle), and D^3 (right). Points above the diagonal reflect flows that have met their deadlines.

Fair share protocols, like $\mathrm{RCP_{dc}}$ and DCTCP, address precisely this issue by ensuring no losses and short queues in the network; yet, as they are unaware of deadlines, a large number of flows still miss their associated deadlines (see middle Figure 11). For these particular experiments, $\mathrm{RCP_{dc}}$ misses 8.1% of flow deadlines as compared to 1% for D^3. Further, looking at flows that do miss deadlines, $\mathrm{RCP_{dc}}$ causes their completion time to exceed their deadline by 75% at the 95th percentile (30% on average). With D^3, completion times are extended by 27.7% at the 95th percentile (9% on average). This implies that *even if deadlines were "soft", fair share protocols still suffer, as a non-negligible fraction of flows exceed their deadline significantly.* This is not acceptable for datacenters where good performance has to be ensured for a very high percentile of flows.

For $\mathrm{TCP_{pr}}$, we use two-level priorities. Flows with short deadlines are mapped to the higher priority class. Such "deadline awareness" improves its performance over TCP. However, it remains hampered by TCP's congestion control algorithm and suffers from losses. Increasing the priority classes to four (maximum supported by the switch) does not improve performance significantly. Simply using priorities with $\mathrm{RCP_{dc}}$ will not help either. This is because deadlines can vary a lot and require a high number of priority classes while today's switches only support O(10) classes. Further, as discussed in Section 3, switch prioritization is packet-based while deadlines are associated with flows. Finally, bursty datacenter traffic implies that instead of statically mapping flows to priority classes, what is needed is to dynamically prioritize flows based on deadlines and network conditions. D^3 tackles precisely all these issues!

With background flows. We repeat the above flow burst experiment by adding long, background flows. Such flows are used to update the workers with the latest data, and typically don't have deadlines associated with them. For each run, we start a long flow from a traffic generator to the receiver. This flow is bottlenecked at the link between the receiver and its ToR switch. After five seconds, all senders send their responses ([2KB, 50KB]).

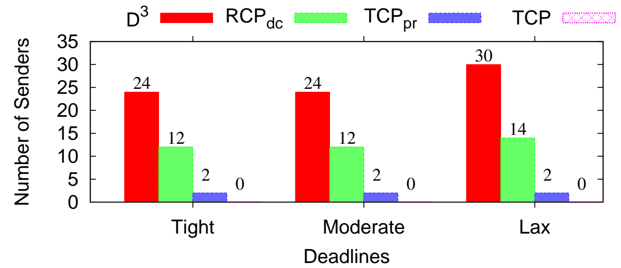


Figure 12: Number of concurrent senders supported while ensuring more than 99% application throughput in the presence of long background flows.

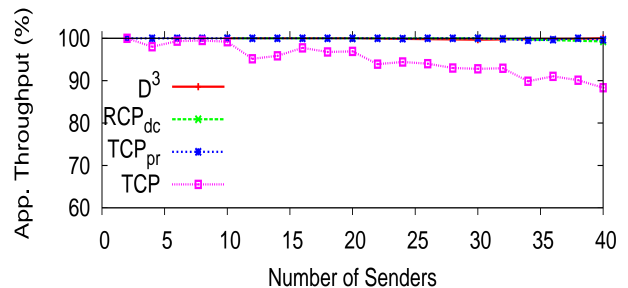


Figure 13: Application throughput with varying number of senders, 2.9KB response flows and one long background flow.

Figure 12 shows the high application throughput operating regime for the three protocols. When compared to the response-flows-only scenario, the relative drop in the maximum number of senders supported is less for D^3 than for TCP, $\mathrm{TCP_{pr}}$ and $\mathrm{RCP_{dc}}$. Performance significantly degrades for TCP and $\mathrm{TCP_{pr}}$ in the presence of queuing due to background flows, which has been well documented in the past [5]. It is noteworthy that even with only two senders, TCP cannot achieve 99% of application throughput. $\mathrm{TCP_{pr}}$ implements a 3-level priority in this scenario, where background flows are assigned to the lowest priority class, and deadline flows are assigned according to their deadline value to the other two priority classes. However, the background flows still consume buffer space and hurt higher priority response flows. Hence, D^3 is even better at satisfying flow deadlines in the presence of background traffic.

With tiny response flows. TCP's travails with flows bursts seen in datacenters have forced application designers to use various "hacks" as workarounds. This includes restricting the response flow size [5]. Here, we re-

at which point the TCP send window will exceed 8 packets.

	TCP (reduced RTO)	RCP_{dc}	D^3
Flows/s	100 (1100)	1300	2000

Table 1: Flow arrival rate supported while maintaining >99% application throughput.

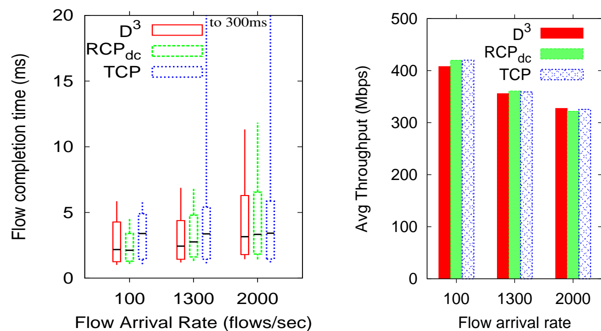
peat the flow burst experiment with a uniform response size of 2.9KB such that response flows are 2 packets long. Further, there exists one long background flow. Figure 13 shows the application throughput with moderate deadlines (mean=30ms). With TCP, the long flow fills up the queue, causing some of the response flows to suffer losses. Consequently, application throughput starts dropping at only 12 senders. As a contrast, the other three approaches satisfy all deadlines untill 40 senders. Since response flows are tiny, there is no room for D^3 to improve upon RCP_{dc} and TCP_{pr}. However, our earlier results show that if designers were to be freed of the constraints imposed by TCP inefficiencies, D^3 can provide significant gains over fair sharing approaches and priorities.

Since RCP_{dc} presents an upper limit for fair share protocols and further performs better than priorities, in the following sections, we will focus on comparing D^3 against RCP_{dc} only; TCP will also be presented as a reference for today's status quo.

6.1.2 Benchmark traffic

In this section, we evaluate the performance of D^3 under realistic datacenter traffic patterns and flow dynamics. As before, we choose one endhost in each rack as the aggregator that receives response flows from the two other endhosts in the rack. The response flow size is uniformly distributed between [2KB, 50KB], and we focus on results with moderate deadlines (mean=30ms). Further, each sender also has one long flow to the aggregator. Both long flows are bottlenecked at the aggregator's link to its ToR switch. This represents the 75th percentile of long flow multiplexing observed in datacenter traffic [5]. We model the response flow arrival as a Poisson process and vary the mean arrival rate. As before, our primary focus is on the flow arrival rate that can be supported while maintaining more than 99% application throughput. This is shown in Table 1.

We find that under realistic traffic dynamics, both D^3 and RCP_{dc} outperform TCP by more than an order of magnitude. These results are best understood by looking at the underlying network level metrics. Figure 14(a) plots the percentiles for flow completion times (1st-5th-50th-95th-99th) at different flow arrival rates. At very low load, the flow completion times for the three protocols are comparable (TCP's median flow completion time is 50% higher than D^3). However, as the flow arrival rate increases, the presence of long flows and the resulting queuing with TCP inevitably causes some re-



(a) 1st-5th-50th-95th-99th percentiles (b) Long flow throughput

Figure 14: Benchmark traffic including short response flows (poisson arrivals with varying arrival rate) and two long flows.

sponse flows to drop packets. Even at 200 flows/s, more than 1% flows suffer drops and miss their deadlines. Given the retransmission timeout value (RTO=300ms), the 99th percentile flow completion time for TCP is more than 300ms. Reducing the RTO to a lower value like 10ms [7] does help TCP to improve its performance by being able to support roughly 1100 flows (Table 1). Yet, even with this optimization, TCP can support less than half the flows supported by D^3.

The flow completion times for D^3 and RCP_{dc} are similar throughout. For instance, even at 2000 flows/s, the 99th percentile completion time is almost the same even though D^3 can satisfy 99% of flow deadlines while RCP_{dc} can only satisfy 96.5% flow deadlines. We reiterate the fact that deadline awareness does not improve flow completion times over RCP_{dc} (which minimizes them already). However, by being cognizant of the flow deadlines, it ensures a greater fraction of flows satisfy them.

We also look at the performance of the long flows to examine whether the gains offered by D^3 come at the expense of (non-deadline) long background flows. Figure 14(b) shows the average throughput achieved by each long flow during the experiment. The figure shows that long flows do not suffer with D^3. Instead, it achieves its gains by smarter allocation of resources amongst the deadline flows.

6.1.3 Flow quenching

The results above illustrate the benefits of unfair sharing. Beyond this, deadline awareness can also guide "flow quenching" to cope with severe congestion. As described in Section 1, under extreme duress, it may be better to shed some load and ensure that a good fraction of flows meet their deadlines, rather than to run all flows to completion even though most will miss their

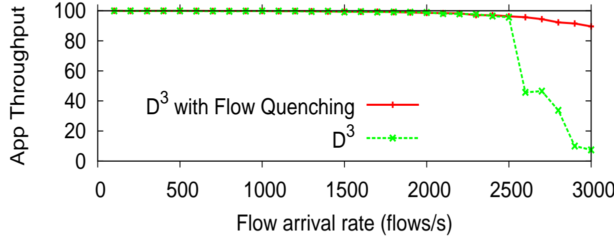


Figure 15: Application throughput with flow quenching

deadlines and not contribute at all. D^3 design is particularly suited to such flow quenching. Since endhosts know the rate needed to satisfy their deadlines and the rate the network can offer, they can independently decide whether to continue with the flow or not.

We implemented a straightforward flow quenching mechanism wherein endhosts prematurely terminate flows (by sending a FIN) when: (i). desired rate exceeds their uplink capacity, or (ii). the deadline has already expired. Figure 15 shows the application throughput with such flow quenching for the benchmark traffic experiment.

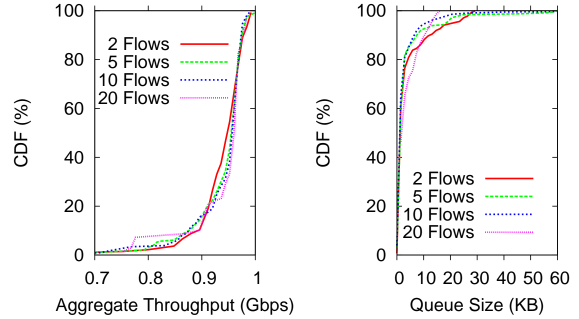
Flow quenching leads to a smoother decline in performance at extreme loads. From the application perspective, fewer end users get empty responses. Beyond 2500 flows/s, D^3 cannot cope with the network load since the flow arrival rate exceeds the flow departure rate. Consequently, the application throughput drops drastically as the network suffers congestive collapse. However, with flow quenching, endhosts do not pursue intractable deadlines which, in turn, spares bandwidth for other flows whose deadlines can be met.

6.2 D^3 as a congestion control protocol

We also evaluated the performance of D^3 as a congestion control protocol in its own right, operating without any deadline information. Results in earlier sections already show that D^3 (and RCP_{dc}) outperforms TCP in terms of short flow latency and tolerance of flow bursts. Here we look at other aspects of D^3 performance.

Throughput and Queuing. We first evaluate the behavior of D^3 with long flows to determine the network throughput achieved. To this effect, we start a varying number of flows (2-20) to an endhost and record the flow throughput at 1ms intervals. The CDF for the aggregate throughput achieved by the flows is shown in Figure 16(a). The figure shows that the aggregate throughput remains the same as we increase the number of long flows with a median and average network throughput of 0.95Gbps (95% of capacity). Overall, D^3 matches the performance offered by TCP for long flows.

We also measured the instantaneous queue length on the bottleneck link to determine D^3's queuing behavior.



(a) Aggregate throughput (b) Queue size

Figure 16: D^3 performance with long flows.

This is plotted in Figure 16(b). For all scenarios, the average queue size is 4-6KB, the median is 1.5KB while the 99th percentile is 16.5-48KB. As a contrast, TCP tends to drive long flows to fill switch buffers and results in much larger queues. This, in turn, impacts any short (deadline) flows that share the same bottleneck. Even comparing to DCTCP [5], D^3 achieves the same, if not better, throughput with queues that are shorter by a factor of five.

Multi-hop, multi-bottleneck setting. To evaluate D^3 performance in a multi-hop network, we set up a scenario with long flows as follows: $1 \to 3$, $2 \to 3$, $4 \to 3$, and $4 \to 5$ (endhost numbering in figure 8). The first three flows are bottlenecked at the downlink from the ToR switch to host 3. The last flow is bottlenecked at the uplink from host 4 to its ToR switch. Hence, this is a multi-hop, multi-bottleneck scenario. As expected, the first three flows achieved an average network throughput of 302 Mbps each, while the average throughput for the last flow was 595 Mbps. Other complex scenarios yielded expected results. Overall, our experiments show that D^3 performs well in scenarios with both multiple hops as well as multiple bottlenecks.

7. DISCUSSION

While we have evaluated many aspects of D^3 design and performance, a number of issues were not discussed in detail due to space limitations. We briefly comment on the most important of these here.

Deployability. D^3 takes a radical tact to align the datacenter network to application requirements. It mandates changes to almost every participant: applications, endhosts, and network elements. While in all cases these changes might be considered easy to implement, the choice of modifying the main components is quite intentional. Discussions with both datacenter operators and application designers have revealed that they are quite willing to adopt new designs that would free them from the artificial restrictions posed by existing retrofitted protocols. This is especially true if the added

benefit is significant as in the case of D^3. The use of a UDP-based transport protocol by Facebook is a good example of the extra miles designers are willing to go to overcome current limitations [15].

The biggest hurdle to D^3 deployment may be the changes necessitated to network elements. From a technical perspective, we have strived to restrict the state maintained and the processing done by a D^3 router to a bare minimum. This allowed our user-space software implementation to easily achieve line-rates and bodes well for a hardware implementation. For instance, like RCP [10], D^3 can be implemented on NetFPGA boards.
Soft vs. hard deadlines. Throughout the paper, D^3 operates under the assumption that deadlines are hard, and once missed, flows are useless. This decision was intentional to stress a, perhaps, extreme design point: *Being deadline aware provides significant value to the network*. On the other hand, one can imagine applications and flows that operate on soft deadlines. Such flows may, for example, gracefully degrade their performance once the deadline is missed without needing to be quenched. The D^3 model can accommodate soft deadlines in a number of ways. For example, since the host controls the requested rate, flows with soft requirements could extend their deadlines if these are missed, or fall back to fair share mode; alternatively, a two-stage allocation process in the router could be implemented, where demands are met in the order of importance depending on the network congestion. Yet, even with the presence of soft deadlines, the evaluation in Section 6 stresses the benefits of deadline-aware protocols over fair share and priorities.
Speculative flow quenching. Beyond the mechanisms described in Section 6.1.3, we are currently experimenting with more aggressive flow quenching heuristics, where hosts attempt to predict flows that will eventually miss their deadline. Such heuristics include, for example, successive increases of the desired rate over multiple RTTs. However, we believe that the aggressiveness of speculative flow quenching depends on the applications under consideration and we defer its discussion to future work.

8. RELATED WORK & CONCLUDING REMARKS

Sections 2 and 3 discuss various proposals relevant to D^3 design, while related work is cited throughout the paper. Here, we briefly comment on other efforts.

The use of RCP [10] as the basis for D^3 design implies that the fair share version of D^3 (RCP$_{dc}$) emulates Processor Sharing. Consequently, both D^3 and RCP$_{dc}$ can match, if not better, the flow completion times of other protocols that do fair sharing, including the ones proposed for datacenter environments. Examples include QCN [24], E-TCP [25], DCTCP [5], etc. However, as discussed in Section 6.1.2, even in the big space of protocols that minimize flow completion times, the resulting application throughput can vary significantly.

To conclude, D^3 is a control protocol that uses application deadline information to achieve informed allocation of network bandwidth. It explicitly deals with the challenges of the datacenter environment - small RTTs, and a bursty, diverse traffic mix with widely varying deadlines. Our evaluation shows that D^3 is practical and provides significant benefits over even optimized versions of existing solutions. This, we believe, is a good illustration of how datacenters that are tuned to application requirements and exploit inherent aspects of the underlying network can perform above and beyond the current state of the art. Emerging trends indicate that operators are willing to adopt new designs that address their problems and this bodes well for D^3 adoption.

9. REFERENCES

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *ACM SIGOPS*, vol. 41, no. 6, 2007.

[2] T. Hoff, “10 eBay Secrets for Planet Wide Scaling,” Nov. 2009, http://highscalability.com/blog/2009/11/17/10-ebay-secrets-for-planet-wide-scaling.html.

[3] W. Vogels, “Performance and Scalability,” Apr. 2009, http://www.allthingsdistributed.com/2006/04/performance_and_scalability.html.

[4] T. Hoff, “Latency is Everywhere and it Costs You Sales - How to Crush it,” Jul. 2009, http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html.

[5] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center TCP (DCTCP),” in *ACM SIGCOMM*, 2010.

[6] D. Katabi, M. Handley, and C. Rohrs, “Congestion Control for High Bandwidth-Delay Product Networks,” in *Proc. of ACM SIGCOMM*, Aug. 2002.

[7] 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.

[8] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, “Understanding TCP incast throughput collapse in datacenter networks,” in *WREN*, 2009.

[9] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of the ACM*, vol. 20, no. 1, 1973.

[10] N. Dukkipati, “Rate Control Protocol (RCP): Congestion control to make flows complete quickly,” Ph.D. dissertation, Stanford University, 2007.

[11] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, “Finding a Needle in Haystack: Facebook’s Photo Storage,” in *Proc. of OSDI*, 2010.

[12] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *USENIX OSDI*, 2004.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks,” in *Proc. of EuroSys*, Mar. 2007.

[14] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, “Controlled experiments on the web: survey and practical guide,” *Data Mining and Knowledge Discovery*, vol. 18, no. 1, 2009.

[15] P. Saab, “Scaling memcached at Facebook,” Dec. 2008, http://www.facebook.com/note.php?note_id=39391378919.

[16] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *Proc. of ACM SOSP*, 2001.

[17] D. Ferrari, A. Banerjea, and H. Zhang, "Network support for multimedia: A discussion of the tenet approach," in *Proc. of Computer Networks and ISDN Systems*, 1994.

[18] C. Aras, J. Kurose, D. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proc.of the IEEE*, vol. 82, no. 1, 1994.

[19] B. B. Chen and P.-B. Primet, "Scheduling deadline-constrained bulk data transfers to minimize network congestion," in *CCGRID*, May 2007.

[20] 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 *Proc. of ACM SIGCOMM*, 2009.

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

[22] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proc. of ACM SIGCOMM*, 2008.

[23] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," in *ACM SIGCOMM*, 2010.

[24] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Laksmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms: congestion control theory and IEEE standardization," in *Proc. of Allerton Conference on Communications, Control and Computing*, Sep. 2008.

[25] Y. Gu, C. V. Hollot, and H. Zhang, "Congestion Control for Small Buffer High Speed Networks," in *Proc. of IEEE INFOCOM*, 2007.