Finishing Flows Quickly with Preemptive Scheduling

Chi-Yao Hong
UIUC
cyhong@illinois.edu

Matthew Caesar
UIUC
caesar@illinois.edu

P. Brighten Godfrey
UIUC
pbg@illinois.edu

ABSTRACT

Today's data centers face extreme challenges in providing low latency. However, fair sharing, a principle commonly adopted in current congestion control protocols, is far from optimal for satisfying latency requirements.

We propose Preemptive Distributed Quick (**PDQ**) flow scheduling, a protocol designed to complete flows quickly and meet flow deadlines. PDQ enables flow preemption to approximate a range of scheduling disciplines. For example, PDQ can emulate a shortest job first algorithm to give priority to the short flows by pausing the contending flows. PDQ borrows ideas from centralized scheduling disciplines and implements them in a fully distributed manner, making it scalable to today's data centers. Further, we develop a multipath version of PDQ to exploit path diversity.

Through extensive packet-level and flow-level simulation, we demonstrate that PDQ significantly outperforms TCP, RCP and D^3 in data center environments. We further show that PDQ is stable, resilient to packet loss, and preserves nearly all its performance gains even given inaccurate flow information.

Categories and Subject Descriptors: C.2.2 [Computer-Communication Networks]: Network Protocols
General Terms: Algorithms, Design, Performance
Keywords: Data center, Flow scheduling, Deadline

1. INTRODUCTION

Data centers are now used as the underlying infrastructure of many modern commercial operations, including web services, cloud computing, and some of the world's largest databases and storage services. Data center applications including financial services, social networking, recommendation systems, and web search often have very demanding latency requirements. For example, even fractions of a second make a quantifiable difference in user experience for web services [7]. And a service that aggregates results from many back-end servers has even more stringent requirements on completion time of the back-end flows, since the service must often wait for the *last* of these flows to finish or else reduce the quality of the final results.[1] Minimizing delays from network congestion, or meeting soft-real-time deadlines with high probability, is therefore important.

Unfortunately, current transport protocols neither minimize flow completion time nor meet deadlines. TCP, RCP [10], ICTCP [22], and DCTCP [3] approximate *fair sharing*, dividing link bandwidth equally among flows. Fair sharing is known to be far from optimal in terms of minimizing flow completion time [4] and the number of deadline-missing flows [5]. As a result, a study of three production data centers [20] showed that a significant fraction $(7-25\%)$ of flow deadlines were missed, resulting in degradation of application response quality, waste of network bandwidth, and ultimately loss of operator revenue [3].

This paper introduces **Preemptive Distributed Quick (PDQ)** flow scheduling, a protocol designed to complete flows quickly and meet flow deadlines. PDQ builds on traditional real-time scheduling techniques: when processing a queue of tasks, scheduling in order of Earliest Deadline First (EDF) is known to minimize the number of late tasks, while Shortest Job First (SJF) minimizes mean flow completion time. However, applying these policies to scheduling data center flows introduces several new challenges.

First, EDF and SJF assume a centralized scheduler which knows the global state of the system; this would impede our goal of low latency in a large data center. To perform dynamic decentralized scheduling, PDQ provides a distributed algorithm to allow a set of switches to collaboratively gather information about flow workloads and converge to a stable agreement on allocation decisions. Second, unlike "fair sharing" protocols, EDF and SJF rely on the ability to *preempt* existing tasks, to ensure a newly arriving task with a smaller deadline can be completed before a currently-scheduled task. To support this functionality in distributed environments, PDQ provides the ability to perform distributed preemption of existing flow traffic, in a manner that enables fast switchover and is guaranteed to never deadlock.

Thus, PDQ provides a distributed flow scheduling layer which is *lightweight*, using only FIFO tail-drop queues, and *flexible*, in that it can approximate a range of scheduling disciplines based on relative priority of flows. We use this primitive to implement two scheduling disciplines: EDF to minimize mean flow completion time, and SJF to minimize the number of deadline-missing flows.

Through an extensive simulation study using real datacenter workloads, we find that PDQ provides strong benefits over existing datacenter transport mechanisms. PDQ is

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1419-0/12/08 ...$15.00.

[1] See discussion in [3], §2.1.

most closely related to D^3 [20], which also tries to meet flow deadlines. Unlike D^3, which is a "first-come first-reserve" algorithm, PDQ proactively and preemptively gives network resources to the most critical flows. For deadline-constrained flows, our evaluation shows PDQ supports 3 times more concurrent senders than [20] while satisfying their flow deadlines. When flows have no deadlines, we show PDQ can reduce mean flow completion times by ∼30% or more compared with TCP, RCP, and D^3.

The key contributions of this paper are:

- We design and implement PDQ, a distributed flow scheduling layer for data centers which can approximate a range of scheduling disciplines.
- We build on PDQ to implement flow scheduling disciplines that minimize mean flow completion time and the number of deadline-missing flows.
- We demonstrate PDQ can save ∼30% average flow completion time compared with TCP, RCP and D^3; and can support 3× as many concurrent senders as D^3 while meeting flow deadlines.
- We show that PDQ is stable, resilient to packet loss, and preserves nearly all its performance gains even given inaccurate flow information.
- We develop and evaluate a multipath version of PDQ, showing further performance and reliability gains.

2. OVERVIEW

We start by presenting an example to demonstrate potential benefits of PDQ over existing approaches (§2.1). We then give a description of key challenges that PDQ must address (§2.2).

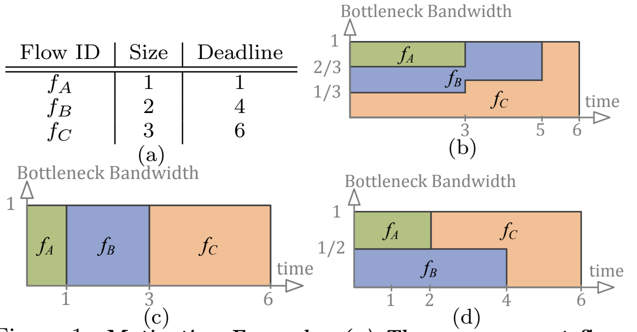
2.1 Example of Benefits

Consider the scenario shown in Figure 1, where three concurrent flows (f_A, f_B, and f_C) arrive simultaneously.

Deadline-unconstrained Case: Suppose that the flows have no deadlines, and our objective is to minimize the average flow completion time. Assuming a fluid traffic model (infinitesimal units of transmission), the result given by fair sharing is shown in Figure 1b: [f_A,f_B,f_C] finish at time [3,5,6], and the average flow completion time is $\frac{3+5+6}{3} = 4.67$. If we schedule the flows by SJF (one by one according to flow size), as shown in Figure 1c, the completion time becomes $\frac{1+3+6}{3} = 3.33$, a savings of ∼29% compared to fair sharing. Moreover, for every individual flow, the flow completion time in SJF is no larger than that given by fair sharing.

Deadline-constrained Case: Suppose now that the flows have deadlines, as specified in Figure 1a. The objective becomes minimizing the number of tardy flows, i.e., maximizing the number of flows that meet their deadlines. For fair sharing, both flow f_A and f_B fail to meet their deadlines, as shown in Figure 1b. If we schedule the flows by EDF (one by one according to deadline), as shown in Figure 1c, every flow can finish before its deadline.

Now we consider D^3, a recently proposed deadline-aware protocol for data center networks [20]. When the network is congested, D^3 satisfies as many flows as possible according to the flow request rate in the order of their arrival. In particular, each flow will request a rate $r = \frac{s}{d}$, where s is the flow's size and d is the time until its deadline. Therefore, the result of D^3 depends highly on flow arrival order. Assuming flows arrive in the order $f_B \rightsquigarrow f_A \rightsquigarrow f_C$, the result of D^3 is shown in Figure 1d. Flow f_B will send with rate $\frac{2}{4} = 0.5$ and will finish right before its deadline. However, flow f_A, which arrives later than f_B, will fail to meet its deadline using the remaining bandwidth, as evident in Figure 1d. In fact, out of $3! = 6$ possible permutations of flow arrival order, D^3 will fail to satisfy some of the deadlines for 5 cases, the only exception being the order $f_A \rightsquigarrow f_B \rightsquigarrow f_C$, which is the order EDF chooses. Although D^3 also allows senders to terminate flows that fail to meet their deadlines (to save bandwidth), termination does not help in this scenario and is not presented in Figure 1d.



Flow ID	Size	Deadline
f_A	1	1
f_B	2	4
f_C	3	6

(a)

Figure 1: ***Motivating Example. (a) Three concurrent flows competing for a single bottleneck link; (b) Fair sharing; (c) SJF/EDF; (d) D^3 for flow arrival order $f_B \rightsquigarrow f_A \rightsquigarrow f_C$.***

2.2 Design Challenges

Although attractive performance gains are seen from the example, many design challenges remain to realize the expected benefits.

Decentralizing Scheduling Disciplines: Scheduling disciplines like EDF or SJF are centralized algorithms that require global knowledge of flow information, introducing a single point of failure and significant overhead for senders to interact with the centralized coordinator. For example, a centralized scheduler introduces considerable flow initialization overhead, while becoming a congestive hot-spot. This problem is especially severe in data center workloads where the majority of flows are mice. A scheduler maintaining only elephant flows like DevoFlow [8] seems unlikely to succeed in congestion control as deadline constraints are usually associated with mice. The need to address the above limitations leads to PDQ, a fully distributed solution where switches collaboratively control flow schedules.

Switching Between Flows Seamlessly: The example of §2.1 idealistically assumed we can start a new flow immediately after a previous one terminates, enabling all the transmission schedules (Figure 1b, 1c and 1d) to fully utilize the bottleneck bandwidth and thus complete flows as quickly as possible. However, achieving high utilization during flow switching in practice requires precise control of flow transmission time. One could simplify this problem by assuming synchronized time among both switches and senders, but that introduces additional cost and effort to coordinate clocks. PDQ addresses this problem by starting the next set of waiting flows slightly before the current sending flows finish.

Prioritizing Flows using FIFO Tail-drop Queues: One could implement priority queues in switches to approximate flow scheduling by enforcing packet priority. Ideally, this requires that each of the concurrent flows has a unique priority class. However, a data center switch can have several thousand active flows within a one second bin [6], while modern switches support only ∼10 priority classes [20]. Therefore, for today's data center switches, the number of priority classes per port is far below the requirements of such an approach, and it is unclear whether modifying switches to support a larger number of priority classes can be cost-effective. To solve this, PDQ explicitly controls the flow sending rate to regulate flow traffic and retain packets from low-priority flows at senders. With this flow pausing strategy, PDQ only requires per-link FIFO tail-drop queues at switches.

3. PROTOCOL

We first present an overview of our design. We then describe the design details implemented at the sender (§3.1), receiver (§3.2) and switches (§3.3). This section assumes each flow uses a single path. In §6, we will show how PDQ can be extended to support multipath forwarding.

Centralized Algorithm: To clarify our approach, we start by presenting it as an idealized *centralized* scheduler with complete visibility of the network, able to communicate with devices in the network with *zero delay*. To simplify exposition, the centralized scheduler assumes that flows have no deadlines, and our only goal is to optimize flow completion time. We will later relax these assumptions.

We define the *expected flow transmission time*, denoted by $\mathcal{T}_i$ for any flow i, to be the remaining flow size divided by its *maximal sending rate* $\mathcal{R}_i^{\max}$. The maximal sending rate $\mathcal{R}_i^{\max}$ is the minimum of the sender NIC rate, the switch link rates, and the rate that receiver can process and receive. Whenever network workload changes (a new flow arrives, or an existing flow terminates), the centralized scheduler recomputes the flow transmission schedule as follows:

1. B_e = available bandwidth of link e, initialized to e's line rate.
2. For each flow i, in increasing order of $\mathcal{T}_i$:
   (a) Let P_i be flow i's path.
   (b) Send flow i with rate $\mathcal{R}_i^{\text{sch}} = \min_{\forall e \in P_i}(\mathcal{R}_i^{\max}, B_e)$.
   (c) $B_e \leftarrow B_e - \mathcal{R}_i^{\text{sch}}$ for each $e \in P_i$.

Distributed Algorithm: We eliminate the unrealistic assumptions we made in the centralized algorithm to construct a fully distributed realization of our design. To distribute its operation, PDQ switches propagate flow information via explicit feedback in packet headers. PDQ senders maintain a set of flow-related variables such as flow sending rate and flow size and communicate the flow information to the intermediate switches via a scheduling header added to the transport layer of each data packet. When the feedback reaches the receiver, it is returned to the sender in an ACK packet. PDQ switches monitor the incoming traffic rate of each of their output queues and inform the sender to send data with a specific rate ($\mathcal{R}$>0) or to pause ($\mathcal{R}$=0) by annotating the scheduling header of data/ACK packets. We present the details of this distributed realization in the following sections.

3.1 PDQ Sender

Like many transport protocols, a PDQ sender sends a SYN packet for flow initialization and a TERM packet for flow termination, and resends a packet after a timeout. The sender maintains standard data structures for reliable transmission, including estimated round-trip time and states (e.g., timer) for in-flight packets. The PDQ sender maintains several state variables: its current sending rate ($\mathcal{R}_S$, initialized to zero), the ID of the switch (if any) who has paused the flow ($\mathcal{P}_S$, initialized to ø), flow deadline ($\mathcal{D}_S$, which is optional), the expected flow transmission time ($\mathcal{T}_S$, initialized to the flow size divided by sender NIC rate), the inter-probing time ($\mathcal{I}_S$, initialized to ø), and the measured RTT (RTT_S, estimated by an exponential decay).

The sender sends packets with rate $\mathcal{R}_S$. If the rate is zero, the sender sends a *probe* packet every $\mathcal{I}_S$ RTTs to get rate information from the switches. A probe packet is a packet with a scheduling header but no data content.

On packet departure, the sender attaches a scheduling header to the packet, containing fields set based on the values of each of the sender's state variables above. $\mathcal{R}_H$ is always set to the maximal sending rate $\mathcal{R}_S^{\max}$, while the remaining fields in the scheduling header are set to its current maintained variables. Note that the subscript H refers to a field in the scheduling $\underline{\text{h}}$eader; the subscript S refers to a variable maintained by the $\underline{\text{s}}$ender; the subscript i refers to a variable related to the ith flow in the switch's flow list.

Whenever an ACK packet arrives, the sender updates its flow sending rate based on the feedback: $\mathcal{T}_S$ is updated based on the remaining flow size, RTT_S is updated based on the packet arrival time, and the remaining variables are set to the fields in the scheduling header.

Early Termination: For deadline-constrained flows, when the incoming flow demand exceeds the network capacity, there might not exist a feasible schedule for all flows to meet their deadlines. In this case, it is desirable to discard a minimal number of flows while satisfying the deadline of the remaining flows. Unfortunately, minimizing the number of tardy flows in a dynamic setting is an $\mathcal{NP}$-complete problem.[2]

Therefore, we use a simple heuristic, called *Early Termination*, to terminate a flow when it cannot meet its deadline. Here, the sender sends a TERM packet whenever *any* of the following conditions happen:

1. Deadline is past (Time > $\mathcal{D}_S$).
2. The remaining flow transmission time is larger than the time to deadline (Time + $\mathcal{T}_S > \mathcal{D}_S$).
3. The flow is paused ($\mathcal{R}_S = 0$), and the time to deadline is smaller than an RTT (Time + $RTT_S > \mathcal{D}_S$).

3.2 PDQ Receiver

A PDQ receiver copies the scheduling header from each data packet to its corresponding ACK. Moreover, to avoid the sender overrunning the receiver's buffer, the PDQ receiver reduces $\mathcal{R}_H$ if it exceeds the maximal rate that receiver can process and receive.

3.3 PDQ Switch

The high-level objective of a PDQ switch is to let the most *critical* flow complete as soon as possible. To this end, switches share a common flow comparator, which decides

[2] Consider a subproblem where a set of concurrent flows that share a bottleneck link all have the same deadline. This subproblem of minimizing the number of tardy flows is exactly the $\mathcal{NP}$-complete subset sum problem [11].

flow criticality, to approximate a range of scheduling disciplines. In this study, we implement two disciplines, EDF and SJF, while we give higher priority to EDF. In particular, we say a flow is more critical than another one if it has smaller deadline (emulating EDF to minimize the number of deadline-missing flows). When there is a tie or flows have no deadline, we break it by giving priority to the flow with smaller expected transmission time (emulating SJF to minimize mean flow completion time). If a tie remains, we break it by flow ID. If desired, the operator could easily override the comparator to approximate other scheduling disciplines. For example, we also evaluate another scheduling discipline incorporating flow waiting time in §7.

The switch's purpose is to resolve flow contention: flows can preempt less critical flows to achieve the highest possible sending rate. To achieve this goal, the switches maintain state about flows on each link (§3.3.1) and exchange information by tagging the scheduling header. To compute the rate feedback ($\mathcal{R}_H$), the switch uses a flow controller (controlling which flows to send; §3.3.2) and a rate controller (computing the aggregate flow sending rate; §3.3.3).

3.3.1 Switch State

In order to resolve flow contention, the switch maintains state about flows on each link. Specifically, it remembers the most recent variables ($<\mathcal{R}_i$, $\mathcal{P}_i$, $\mathcal{D}_i$, $\mathcal{T}_i$, $RTT_i>$) obtained from observed packet headers for flow i, which it uses to decide at any moment the correct sending rate for the flows. However, we do not have to keep this state for *all* flows. Specifically, PDQ switches only store the most critical 2κ flows, where κ is the number of *sending* flows (i.e., flows with sending rate $\mathcal{R}_S > 0$). Since PDQ allocates as much link bandwidth as possible to the most critical flows until the link is fully utilized, the κ most critical flows fully utilize the link's bandwidth; we store state for 2κ flows in order to have sufficient information immediately available to unpause another flow if one of the sending flows completes. The remaining flows are not remembered by the switch, until they become sufficiently critical.

The amount of state maintained at the switch thus depends on how many flows are needed to fill up a link. In most practical cases, this value will be very small because (i) PDQ allows critical flows to send with their highest possible rates, and (ii) switch-to-switch links are typically only $1-10\times$ faster than server-to-switch links, e.g., current data center networks mostly use 1 Gbps server links and 10 Gbps switch links[3], and the next generation will likely be 10 Gbps server links and 40 or 100 Gbps switch links. However, if a flow's rate is limited to something less than its NIC rate (e.g., due to processing or disk bottlenecks), switches may need to store more flows.

Greenberg et al. [12] demonstrated that, under a production data center of a large scale cloud service, the number of concurrent flows going in and out of a machine is almost never more than 100. Under a pessimistic scenario where *every* server concurrently sends or receives 100 flows, we have an average of 12,000 active flows at each switch in a VL2 network (assuming flow-level equal-cost multi-path forwarding and 24 10-Gbps Ethernet ports for each switch, the same as done in [12]). Today's switches are typically equipped with $4-16$ MByte of shared high-speed memory[4], while storing all these flows requires 0.23 MByte, only 5.72% of a 4 MByte shared memory. Indeed, in our simulation using the trace from [6], the maximum memory consumption was merely 9.3 KByte.

Still, suppose our memory imposes a hard upper limit M on the number of flows the switch can store. PDQ, as described so far, will cause under-utilization when $\kappa > M$ and there are paused flows wanting to send. In this underutilized case, we run an RCP [10] rate controller—which does not require per-flow state—alongside PDQ. We inform RCP that its maximum link capacity is the amount of capacity not used by PDQ, and we use RCP only for the less critical flows (outside the M most critical) that are not paused by any other switches. RCP will let *all* these flows run simultaneously using the leftover bandwidth. Thus, even in this case of large κ (which we expect to be rare), the result is simply a partial shift away from optimizing completion time and towards traditional fair sharing.

3.3.2 The Flow Controller

The flow controller performs Algorithm 1 and 3 whenever it receives a data packet and an ACK packet, respectively. The flow controller's objective is to *accept* or *pause* the flow. A flow is accepted if *all* switches along the path accept it. However, a flow is paused if *any* switch pauses it. This difference leads to the need for different actions:

Pausing: If a switch decides to pause a flow, it simply updates the "pauseby" field in the header ($\mathcal{P}_H$) to its ID. This is used to inform other switches and the sender that the flow should be paused. Whenever a switch notices that a flow is paused by another switch, it removes the flow information from its state. This can help the switch to decide whether it wants to accept other flows.

Acceptance: To reach consensus across switches, flow acceptance takes two phases: (i) in the forward path (from source to destination), the switch computes the available bandwidth based on flow criticality (Algorithm 2) and updates the rate and pauseby fields in the scheduling header; (ii) in the reverse path, if a switch sees an empty pauseby field in the header, it updates the *global* decision of acceptance to its state ($\mathcal{P}_i$ and $\mathcal{R}_i$).

We now propose several optimizations to refine our design:

Early Start: Given a set of flows that are not paused by other switches, the switch accepts flows according to their criticality until the link bandwidth is fully utilized and the remaining flows are paused. Although this ensures that the more critical flows can preempt other flows to fully utilize the link bandwidth, this can lead to *low link utilization when switching between flows*. To understand why, consider two flows, A and B, competing for a link's bandwidth. Assume that flow A is more critical than flow B. Therefore, flow A is accepted to occupy the entire link's bandwidth, while flow B is paused and sends only probe packets, e.g., one per its RTT. By the time flow A sends its last packet (TERM), the sender of flow B does not know it should start sending data because of the feedback loop delay. In fact, it could take one to two RTTs before flow B can start sending data.

[3] For example, the NEC PF5240 switch supports 48×1 Gbps ports, along with 2×10 Gbps ports; Pronto 3290 switch provides 48×1 Gbps ports and 4×10 Gbps ports.

[4] For example, the "deep-buffered" switches like Cisco Catalyst 4500, 4700, 4900 and 4948 series have 16 MByte shared memory, while shallow-buffered switches like Broadcom Triumph and Scorpion have 4 MByte shared memory [3].

Although the RTT in data center networks is typically very small (e.g., ∼150 μs), the high-bandwidth short-flow nature makes this problem non-negligible. In the worst case where all the flows are short control messages (<10 KByte) that could finish in just one RTT, links could be idle more than half the time.

To solve this, we propose a simple concept, called *Early Start*, to provide seamless flow switching. The idea is to start the next set of flows slightly before the current sending flows finish. Given a set of flows that are not paused by other switches, a PDQ switch classifies a currently sending flow as nearly completed if the flow will finish sending in K RTTs (i.e., $\mathcal{T}_i < K \times RTT_i$), for some small constant K. We let the switch additionally accept as many nearly-completed flows as possible according to their criticality and subject to the resource constraint: aggregated flow transmission time (in terms of its estimated RTT) of the accepted nearly-completed flows ($\sum_i \mathcal{T}_i/RTT_i$) is no larger than K. The threshold K determines how early and how many flows will be considered as nearly-completed. Setting K to 0 will prevent concurrent flows completely, resulting in low link utilization. Setting K to a large number will result in congested links, increased queue sizes, and increased completion times of the most critical flows. Any value of K between 1 and 2 is reasonable, as the control loop delay is one RTT and the inter probing time is another RTT. In our current implementation we set $K = 2$ to maximize the link utilization, and we use the rate controller to drain the queue. Algorithm 2 describes this in pseudocode, and we will show that Early Start provides seamless flow switching (§5).

Dampening: When a more critical flow arrives at a switch, PDQ will pause the current flow and switch to the new flow. However, bursts of flows that arrive concurrently are common in data center networks, and can potentially cause frequent flow switching, resulting in temporary instability in the switch state.[5] To suppress this, we use dampening: after a switch has accepted a flow, it can only accept other paused flows after a given small period of time, as shown in Algorithm 1.

Suppressed Probing: One could let a paused sender send one probe per RTT. However, this can introduce significant bandwidth overhead because of the small RTTs in data center networks. For example, assume a 1-Gbps network where flows have an RTT of 150 μs. A paused flow that sends a 40-byte probe packet per RTT consumes $\frac{40 \text{ Byte}}{150 \ \mu\text{s}}/1$ Gbps $\approx$ 2.13% of the total bandwidth. The problem becomes more severe with larger numbers of concurrent flows.

To address this, we propose a simple concept, called *Suppressed Probing* to reduce the probing frequency. We make an observation that only the paused flows that are about to start have a need to send probes frequently. Therefore, it is desirable to control probing frequency based on the flow criticality and the network load. To control probing frequency, one would need to estimate *flow waiting time* (i.e., how long does it take until the flow can start sending). Although it is considered hard to predict future traffic workloads in data centers, switches can easily estimate a lower bound of the flow waiting time by checking their flow list. Assuming each flow requires at least X RTTs to finish, a PDQ switch estimates that a flow's waiting time is at least $X \times \max_{\forall \ell}\{\text{Index}(\ell)\}$ RTTs, where Index(ℓ) is the flow index in the list on link ℓ. The switch sets the inter-probing time field ($\mathcal{I}_H$) to $\max\{\mathcal{I}_H, X \times \text{Index}(\ell)\}$ in the scheduling header to control the sender probing rate ($\mathcal{I}_S$), as shown in Algorithm 3. The expected per-RTT probing overhead is significantly reduced from $O(n)$ (n flows, each of which sends one probe per RTT) to $\frac{1}{X}\sum_{k=1}^{n} 1/k = O(\log n)$. In our current implementation, we conservatively set X to 0.2 RTT$_{avg}$.

Algorithm 1: PDQ Receiving Data Packet

```
if $\mathcal{P}_H$ = other switch then
    Remove the flow from the list if it is in the list;
    return;
if the flow is not in the list then
    if the list is not full or the flow criticality is higher than the least critical flow in the list then
        Add the flow into the list with rate $\mathcal{R}_i = 0$.
        Remove the least critical flow from the list whenever the list has more than $\kappa$ flows.
    else
        Set $\mathcal{R}_H$ to RCP fair share rate;
        if $\mathcal{R}_H = 0$ then $\mathcal{P}_H$ = myID;
        return;
Let $i$ be the flow index in the list; Update the flow information: $<\mathcal{D}_i,\mathcal{T}_i,RTT_i> = <\mathcal{D}_H,\mathcal{T}_H,RTT_H>$;
if $W$=min($Availbw(i)$,$\mathcal{R}_H$)>0 then
    if the flow is not sending ($\mathcal{P}_i \neq \emptyset$), and the switch just accepted another non-sending flow then
        $\mathcal{P}_H$ = myID; $\mathcal{P}_i$ = myID; // Pause it
    else $\mathcal{P}_H = \emptyset$; $\mathcal{R}_H$=$W$; // Accept it
else $\mathcal{P}_H$ = myID; $\mathcal{P}_i$ = myID; // Pause it
```

Algorithm 2: Availbw(j)

```
$X$=0; $A$=0;
for ($i = 0$; $i < j$; $i = i + 1$) do
    if $\mathcal{T}_i/RTT_i < K$ and $X < K$ then
        $X = X + \mathcal{T}_i/RTT_i$;
    else
        $A = A + \mathcal{R}_i$;
    if $A \geq C$ then return 0;
return $C - A$;
```

Algorithm 3: PDQ Receiving ACK

```
if $\mathcal{P}_H$ = other switch then
    Remove the flow from the list if it is in the list;
if $\mathcal{P}_H \neq \emptyset$ then
    $\mathcal{R}_H = 0$; // Flow is paused
if the flow is in the list with index $i$ then
    $\mathcal{P}_i = \mathcal{P}_H$; $\mathcal{I}_H = \max\{\mathcal{I}_H, X \times i\}$; $\mathcal{R}_i = \mathcal{R}_H$;
```

3.3.3 *The Rate Controller*

The rate controller's objective is to control the aggregated flow sending rate of the flows accepted by the flow controller based on the queue size and the measured aggregate traffic. The rate adjustment serves the following purposes. First, whenever the queue builds up due to the use of Early Start, it helps drain the queue right after flow switching. Second, it helps tolerate the congestion caused by transient inconsistency. For example, if a packet carrying the pausing

[5] We later show that PDQ can quickly converge to the equilibrium state when the workload is stable (§4).

information gets lost, the corresponding sender that is supposed to stop will still be sending, and the rate controller can reduce the flow sending rate to react to the congestion. Finally, this allows PDQ to be friendly to other transport protocols in a multi-protocol network.

The rate controller maintains a single variable C to control the aggregated flow sending rate. This variable will be used to compute the sending rate field ($\mathcal{R}_H$) in the scheduling header, as shown in Algorithm 2.

The rate controller updates C every 2 RTTs because of the feedback-loop delay: we need about one RTT latency for the adjusted rate to take effect, and one additional RTT is used to measure the link congestion with that newly adjusted sending rate.

The rate controller updates C to $\max\{0, r_{\text{PDQ}} - q/(2 \times \text{RTT})\}$, where q is the instantaneous queue size and r_{PDQ} is the per-link aggregated rate for PDQ flows. If all traffic is transported using PDQ, one can configure the r_{PDQ} to be equal to the link rate. This allows PDQ flows to send with its highest possible rate. Otherwise, the network administrator can decide their priority by setting r_{PDQ} accordingly. For example, one could give preference to other protocols by periodically updating r_{PDQ} to the difference between the link rate and the measured aggregated traffic of the other protocols. Alternatively, one could set it based on the per-protocol traffic amount to achieve fairness across protocols.

4. FORMAL PROPERTIES

In this section, we present two formal properties of PDQ — deadlock-freedom and finite convergence time.

Assumptions: Without loss of generality, we assume there is no packet loss. Similarly, we assume flows will not be paused due to the use of flow dampening. Because PDQ flows periodically send probes, the properties we discuss in this section will hold with additional latency when the above assumptions are violated. For simplicity, we also assume the link rate C is equal to the maximal sending rate $\mathcal{R}_S^{\max}$ (i.e., $\mathcal{R}_S^{\text{sch}} = 0$ or C). Thus, each link accepts only one flow at a time.

Definitions: We say a flow is *competing* with another flow if and only if they share at least one common link. Moreover, we say a flow F_1 is a *precedential* flow of flow F_2 if and only if they are competing with each other and flow F_1 is more critical than flow F_2. We say a flow F is a *driver* if and only if (i) flow F is more critical than any other competing flow, or (ii) all the competing flows of flow F that are more critical than flow F are non-drivers.

Results (proof is in [14]): We verify that PDQ has no *deadlock*, which is a situation where two or more competing flows are paused and are each waiting for the other to finish (and therefore neither ever does). We further prove that PDQ will converge to the *equilibrium* in $P_{\max} + 1$ RTTs for stable workloads, where $P_{\max}$ is the maximal number of precedential flows of any flow. Given a collection of active flows, the equilibrium is defined as a state where the drivers are accepted while the remaining flows are paused.

5. PDQ PERFORMANCE

In this section, we evaluate PDQ's performance through comprehensive simulations. We first describe our evaluation setting (§5.1). Under a "query aggregation" scenario, PDQ achieves near-optimal performance and greatly outperforms D^3, RCP and TCP (§5.2). We then demonstrate that PDQ retains its performance gains under different workloads, including two realistic data center workloads from measurement studies (§5.3), followed by two scenarios to demonstrate that PDQ does not compromise on traditional congestion control performance metrics (§5.4). Moreover, PDQ retains its performance benefits on a variety of data center topologies (Fat-Tree, BCube and Jellyfish) and provides clear performance benefits at all scales that we evaluated (§5.5). Further, we show that PDQ is highly resilient to inaccurate flow information and packet loss (§5.6).

5.1 Evaluation Setting

Our evaluation considers two classes of flows:

Deadline-constrained Flows are time sensitive flows that have specific deadline requirements to meet. The flow size is drawn from the interval [2 KByte, 198 KByte] using a uniform distribution, as done in a prior study [20]. This represents query traffic (2 to 20 KByte in size) and delay sensitive short messages (>100 KByte) in data center networks [3]. The flow deadline is drawn from an exponential distribution with mean 20 ms, as suggested by [20]. However, some flows could have tiny deadlines that are unrealistic in real network applications. To address this, we impose a lower bound on deadlines, and we set it to 3 ms in our experiments. We use *Application Throughput*, the percentage of flows that meet their deadlines, as the performance metric of deadline-constrained flows.

Deadline-unconstrained Flows are flows that have no specific deadlines, but it is desirable that they finish early. For example, Dryad jobs that move file partitions across machines. Similarly, we assume the flow size is drawn uniformly from an interval with a mean of 100/1000 KByte. We use the average flow completion time as the performance metric.

We have developed our own event-driven packet-level simulator written in C++. The simulator models the following schemes:

PDQ: We consider different variants of PDQ. We use PDQ(Full) to refer to the complete version of PDQ, including Early Start (ES), Early Termination (ET) and Suppressed Probing (SP). Likewise, we refer to the partial version of PDQ which excludes the above three algorithms as PDQ(Basic). To better understand the performance contribution of each algorithm, we further extend PDQ(Basic) to PDQ(ES) and PDQ(ES+ET).

D^3: We implemented a complete version of D^3 [20], including the rate request processing procedure, the rate adaptation algorithm (with the suggested parameters $\alpha = 0.1$ and $\beta = 1$), and the quenching algorithm. In the original algorithm when the total demand exceeds the switch capacity, the fair share rate becomes negative. We found this can cause a flow to return the allocated bandwidth it already reserved, resulting in unnecessarily missed deadlines. Therefore, we add a constraint to enforce the fair share bandwidth fs to always be non-negative, which improves D^3's performance.

RCP: We implement RCP [10] and optimize it by counting the exact number of flows at switches. We found this improves the performance by converging to the fair share rate more quickly, significantly reducing the number of packet drops when encountering a sudden large influx of new flows [9].

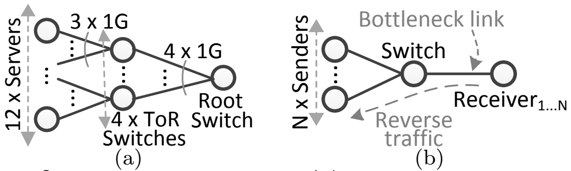


Figure 2: ***Example topologies: (a) a*** 17***-node single-rooted tree topology; (b) a single-bottleneck topology: sending servers associated with different flows are connected via a single switch to the same receiving server. Both topologies use*** 1 ***Gbps links, a switch buffer of*** 4 ***MByte, and FIFO taildrop queues. Per-hop transmission/propagation/processing delay is set to*** $11/0.1/25$ ***μs.***

This is exactly equivalent to D^3 when flows have no deadlines.

TCP: We implement TCP Reno and optimize it by setting a small $RTO_{\min}$ to alleviate the TCP Incast problem, as suggested by previous studies [3, 19].

Unless otherwise stated, we use single-rooted tree, a commonly used data center topology for evaluating transport protocols [3, 19, 20, 22]. In particular, our default topology is a two-level 12-server single-rooted tree topology with 1 Gbps link rate (Figure 2a), the same as used in D^3. We vary the traffic workload and topology in §5.3 and §5.5.

5.2 Query Aggregation

In this section, we consider a scenario called *query aggregation*: a number of senders initiate flows at the same time to the same receiver (the aggregator). This is a very common application scenario in data center networks and has been adopted by a number of previous works [22, 20, 3]. We evaluate the protocols in both the deadline-constrained case (§5.2.1) and the deadline-unconstrained case (§5.2.2).

5.2.1 Deadline-constrained Flows

Impact of Number of Flows: We start by varying the number of flows.[6] To understand bounds on performance, we also simulate an *optimal* solution, where an omniscient scheduler can control the transmission of any flow with no delay. It first sorts the flows by EDF, and then uses a dynamic programming algorithm to discard the minimum number of flows that cannot meet their deadlines (Algorithm 3.3.1 in [16]). We observe that PDQ has near-optimal application throughput across a wide range of loads (Figure 3a).

Figure 3a demonstrates that Early Start is very effective for short flows. By contrast, PDQ(Basic) has much lower application throughput, especially during heavy system load because of the long down time between flow switching. Early Termination further improves performance by discarding flows that cannot meet their deadline. Moreover, Figure 3a demonstrates that, as the number of concurrent flows increases, the application throughput of D^3, RCP and TCP decreases significantly.

Impact of Flow Size: We fix the number of concurrent flows at 3 and study the impact of increased flow size on the application throughput. Figure 3b shows that as the flow size increases, the performance of deadline-agnostic schemes (TCP and RCP) degrades considerably, while PDQ remains very close to optimal regardless of the flow size. However, Early Start and Early Termination provide fewer benefits in this scenario because of the small number of flows.

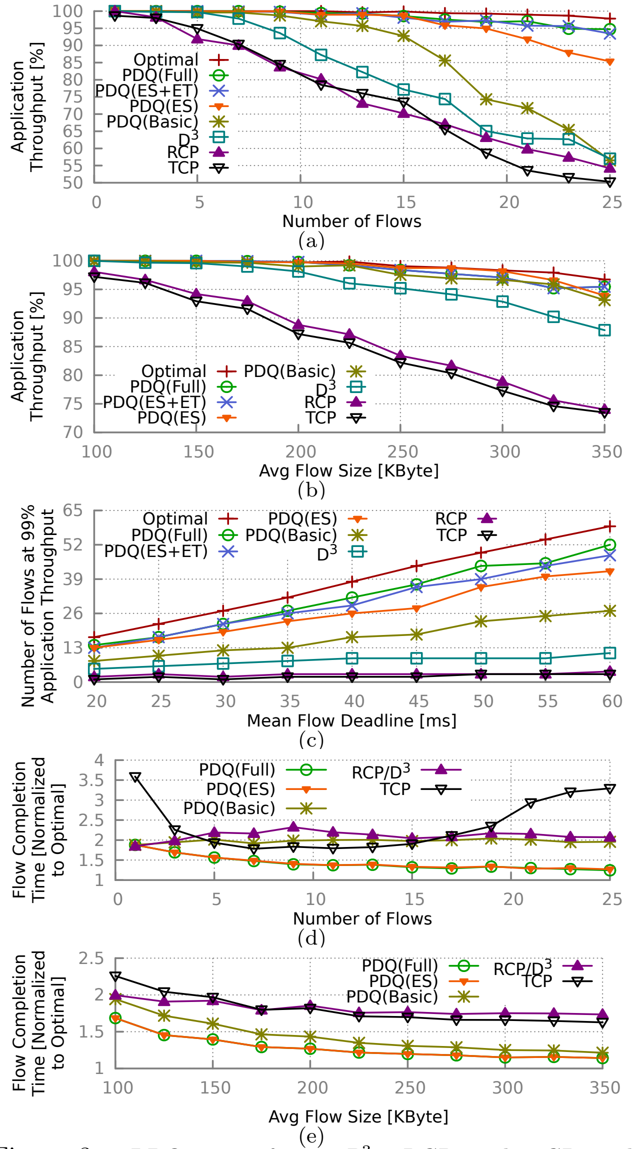


Figure 3: ***PDQ outperforms D***^{3}***, RCP and TCP and achieves near-optimal performance. Top three figures: deadline-constrained flows; bottom two figures: deadline-unconstrained flows.***

Impact of Flow Deadline: Data center operators are particularly interested in the operating regime where the network can satisfy almost every flow deadline. To this end, we attempt to find, using a binary search procedure, the maximal number of flows a protocol can support while ensuring 99% application throughput. We also vary the flow deadline, which is drawn from an exponential distribution, to observe the system performance with regard to different flow deadlines with mean between 20 ms to 60 ms. Figure 3c demonstrates that, compared with D^3, PDQ can support >3 times more concurrent flows at 99% application throughput, and this ratio becomes larger as the mean flow deadline increases. Moreover, Figure 3c shows that Suppressed Probing

[6] We randomly assign f flows to n senders while ensuring each sender has either $\lfloor f/n \rfloor$ or $\lceil f/n \rceil$ flows.

becomes more useful as the number of concurrent flows increases.

5.2.2 Deadline-unconstrained Flows

Impact of Flow Number: For deadline-unconstrained case, we first measure the impact of the number of flows on the average flow completion time. Overall, Figure 3d demonstrates that PDQ can effectively approximate the optimal flow completion time. The largest gap between PDQ and optimal happens when there exists only one flow and is due to flow initialization latency. RCP has a similar performance for the single-flow case. However, its flow completion time becomes relatively large as the number of flows increases. TCP displays a large flow completion time when the number of flows is small due to the inefficiency of slow start. When the number of concurrent flows is large, TCP also has an increased flow completion time due to the TCP incast problem [19].

Impact of Flow Size: We fix the number of flows at 3, and Figure 3e shows the flow completion time as the flow size increases. We demonstrate that PDQ can better approximate optimal flow completion time as flow size increases. The reason is intuitive: the adverse impact of PDQ inefficiency (e.g., flow initialization latency) on flow completion time becomes relatively insignificant as flow size increases.

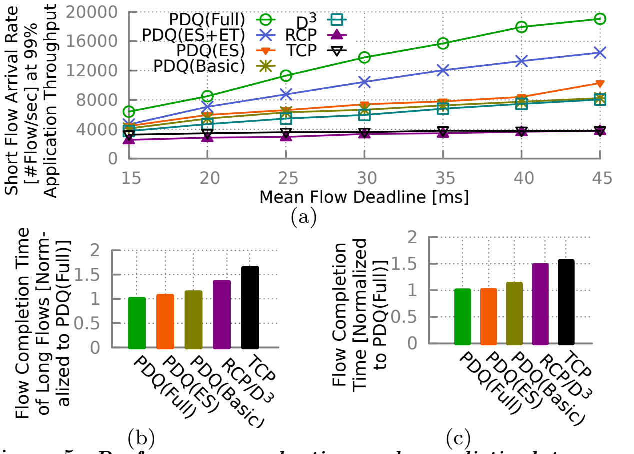
5.3 Impact of Traffic Workload

Impact of Sending Pattern: We study the impact of the following sending patterns: **(i)** *Aggregation*: multiple servers send to the same aggregator, as done in the prior experiment. **(ii)** *Stride*(i): a server with index x sends to the host with index $(x+i)$ *mod* N, where N is the total number of servers; **(iii)** *Staggered Prob*(p): a server sends to another server under the same top-of-rack switch with probability p, and to any other server with probability $1-p$; **(iv)** *Random Permutation*: a server sends to another randomly-selected server, with a constraint that each server receives traffic from exactly one server (i.e., 1-to-1 mapping).

Figure 4 shows that PDQ reaps its benefits across all the sending patterns under consideration. The worst pattern for PDQ is the Staggered Prob(0.7) due to the fact that the variance of the flow RTTs is considerably larger. In this sending pattern, the non-local flows that pass through the core network could have RTTs $3-5$ times larger than the local flow RTTs. Thus, the PDQ rate controller, whose update frequency is based on a measurement of *average* flow RTTs, could slightly overreact (or underreact) to flows with relatively large (or small) RTTs. However, even in such a case, PDQ still outperforms the other schemes considerably.

Impact of Traffic Type: We consider two workloads collected from real data centers. First, we use a workload with flow sizes following the distribution from a large-scale commercial data center measured by Greenberg et al. [12]. It represents a mixture of long and short flows: Most flows are small, and most of the delivered bits are contributed by long flows. In the experiment, we assume that the short flows (with a size of <40 KByte) are deadline-constrained. We conduct these experiments with random permutation traffic.

Figure 5a demonstrates that, under this particular workload, PDQ outperforms the other protocols by supporting a significantly higher flow arrival rate. We observed that, in this scenario, PDQ(Full) considerably outperforms PDQ(ES+ET). This suggests that Suppressed Probing plays an important role in minimizing the probing overhead especially when there exists a large collection of paused flows. Figure 5b shows that PDQ has lower flow completion time for long flows: a 26% reduction compared with RCP, and a 39% reduction compared with TCP.



(a)

(b)

(c)

Figure 5: ***Performance evaluation under realistic data center workloads, collected from* (a, b) *a production data center of a large commercial cloud service [12] and* (c) *a university data center located in Midwestern United States (EDU1 in [6]).***

We also evaluate performance using a workload collected from a university data center with 500 servers [6]. In particular, we first convert the packet trace, which lasts 10 minutes, to flow-level summaries using Bro [1], then we fed it to the simulator. Likewise, PDQ outperforms other schemes in this regime (Figure 5c).

5.4 Dynamics of PDQ

Next, we show PDQ's performance over time through two scenarios, each with varying traffic dynamics:

Scenario #1 (Convergence Dynamics): Figure 6 shows that PDQ provides seamless flow switching. We assume five flows that start at time 0. The flows have no deadlines, and each flow has a size of ~1 MByte. The flow size is perturbed slightly such that a flow with smaller index is more critical. Ideally, the five flows together take 40 ms to finish because each flow requires a raw processing time of $\frac{1 \text{ MByte}}{1 \text{ Gbps}} = 8$ ms. With seamless flow switching, PDQ completes at ~42 ms due to protocol overhead (~3% bandwidth loss due to TCP/IP header) and first-flow initialization time (two-RTT latency loss; one RTT latency for the sender to receive the SYN-ACK, and an additional RTT for the sender to receive the first DATA-ACK). We observe that PDQ can converge to equilibrium quickly at flow switching time, resulting in a near perfect (100%) bottleneck link utilization (Figure 6b). Although an alternative (naive) approach to achieve such high link utilization is to let every flow send with fastest rate, this causes the rapid growth of the queue and potentially leads to congestive packet loss. Unlike this approach, PDQ exhibits a very small queue size[7] and has no packet drops (Figure 6c).

[7] The non-integer values on the y axis comes from the small probing packets.

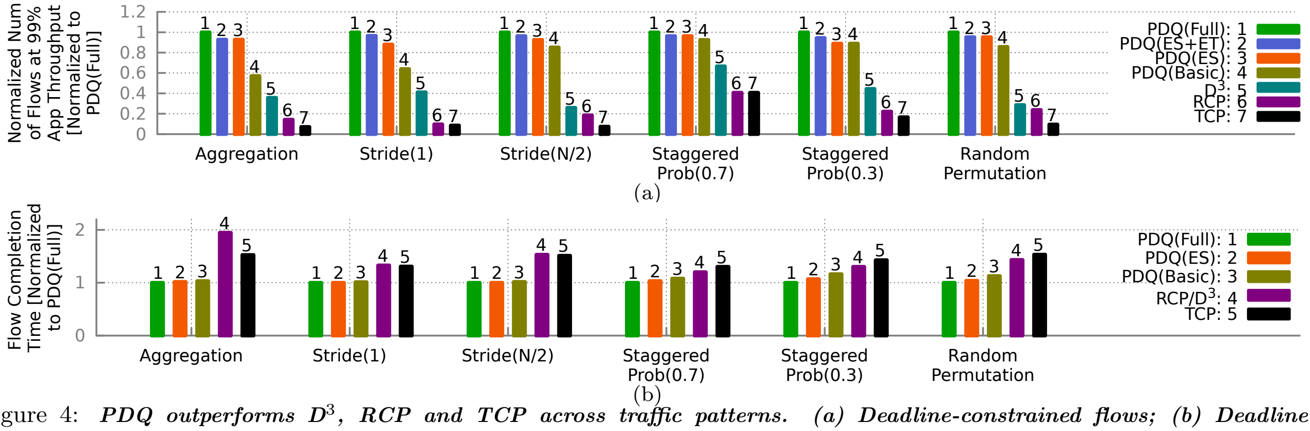


Figure 4: ***PDQ outperforms*** D^3***, RCP and TCP across traffic patterns. (a) Deadline-constrained flows; (b) Deadline-unconstrained flows.***

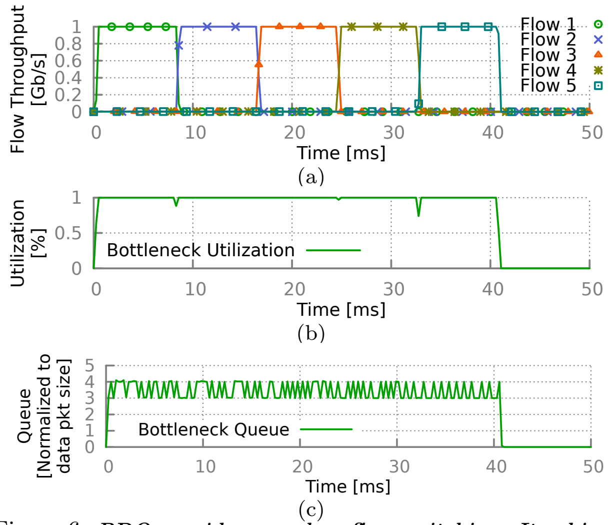


Figure 6: ***PDQ provides seamless flow switching. It achieves high link utilization at flow switching time, maintains small queue, and converges to the equilibrium quickly.***

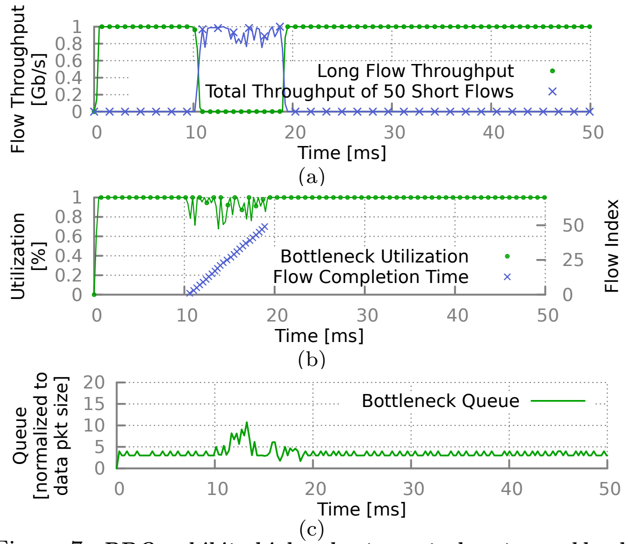


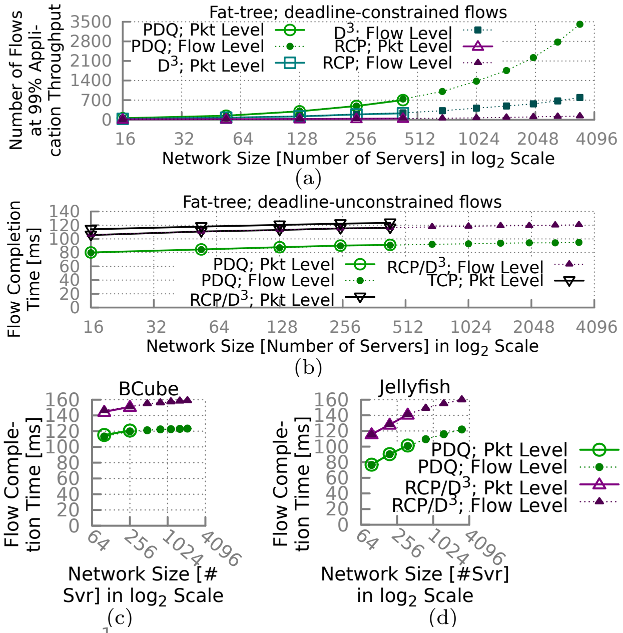
Figure 7: ***PDQ exhibits high robustness to bursty workload. We use a workload of*** 50 ***concurrent short flows all start at time*** 1 ***ms, and preempting a long-lived flow.***

Scenario #2 (Robustness to Bursty Traffic): Figure 7 shows that PDQ provides high robustness to bursty workloads. We assume a long-lived flow that starts at time 0, and 50 short flows that all start at 10 ms. The short flow sizes are set to 20 KByte with small random perturbation. Figure 7a shows that PDQ adapts quickly to sudden bursts of flow arrivals. Because the required delivery time of each short flow is very small ($\frac{20\text{ KByte}}{1\text{ Gbps}} \approx 153\ \mu s$), the system never reaches stable state during the preemption period (between 10 and 19 ms). Figure 7b shows PDQ adapts quickly to the burst of flows while maintaining high utilization: the average link utilization during the preemption period is 91.7%. Figure 7c suggests that PDQ does not compromise the queue length by having only 5 to 10 packets in the queue, which is about an order of magnitude smaller than what today's data center switches can store. By contrast, XCP in a similar environment results in a queue of ~60 packets (Figure 11(b) in [15]).

5.5 Impact of Network Scale

Today's data centers have many thousands of servers, and it remains unclear whether PDQ will retain its successes at large scales. Unfortunately, our packet-level simulator, which is optimized for high processing speeds, does not scale to large-scale data center topology within reasonable processing time. To study these protocols at large scales, we construct a *flow-level simulator* for PDQ, D^3 and RCP. In particular, we use an iterative approach to find the equilibrium flow sending rates with a time scale of 1 ms. The flow-level simulator also considers protocol inefficiencies like flow initialization time and packet header overhead. Although the flow-level simulator does not deal with packet-level dynamics such as timeouts or packet loss, Figure 8 shows that, by comparing with the results from packet-level simulation, the flow-level simulation does not compromise the accuracy significantly.

We evaluate three scalable data center topologies: (1) Fat-tree [2], a structured 2-stage Clos network; (2) BCube [13], a server-centric modular network; (3) Jellyfish [18], an unstructured high-bandwidth network using random regular graphs. Figure 8 demonstrates that PDQ scales well to large scale, regardless of the topologies we tested. Figure 8e shows that about 40% of flow completion times under PDQ are reduced by at least 50% compared to RCP. Only $5-15\%$ of



(e)

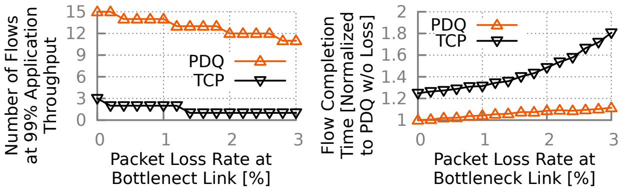
Figure 8: ***PDQ performs well across a variety of data center topologies. (a,b) Fat-tree; (c) BCube with dual-port servers; (d) Jellyfish with*** 24***-port switches, using a*** 2:1 ***ratio of network port count to server port count. (e) For network flows, the ratio of the flow completion time under PDQ to the flow completion time under RCP (flow-level simulation; # servers is*** ∼128***). All experiments are carried out using random permutation traffic; top figure: deadline-constrained flows; bottom four figures: deadline-unconstrained flows with*** 10 ***sending flows per server.***

the flows have a larger completion time, and no more than 0.9% of the flows have 2× completion time.

5.6 PDQ Resilience

Resilience to Packet Loss: Next, to evaluate PDQ's performance in the presence of packet loss, we randomly drop packets at the bottleneck link, in both the forward (data) and reverse (acknowledgment) direction. Figure 9 demonstrates that PDQ is even more resilient than TCP to packet loss. When packet loss happens, the PDQ rate controller detects anomalous high/low link load quickly and compensates for it with explicit rate control. Thus, packet loss does not significantly affect its performance. For a heavily lossy channel where the packet loss rate is 3% in both directions (i.e., a round-trip packet loss rate of $1-(1-0.03)^2 \approx 5.9\%$), as shown in Figure 9b, the flow completion time of PDQ has increased by 11.4%, while that of TCP has significantly increased by 44.7%.

Resilience to Inaccurate Flow Information: For many data center applications (e.g., web search, key-value stores, data processing), previous studies have shown that flow size can be precisely known at flow initiation time.[8] Even for applications without such knowledge, PDQ is resilient to inaccurate flow size information. To demonstrate this, we consider the following two flow-size-unaware schemes. *Random*: the sender randomly chooses a flow criticality at flow initialization time and uses it consistently. *Flow Size Estimation*: the sender estimates the flow size based on the amount of data sent already, and a flow is more critical than another one if it has smaller estimated size. To avoid excessive switching among flows, the sender does not change the flow criticality for every packet it sends. Instead, the sender updates the flow criticality only for every 50 KByte it sends. Figure 10 demonstrates two important results: (i) PDQ does require a reasonable estimate of flow size as random criticality can lead to large mean flow completion time in heavy-tailed flow size distribution. (ii) With a simple estimation scheme, PDQ still compares favorably against RCP in both uniform and heavy-tailed flow size distributions.



(a) (b)

Figure 9: ***PDQ is resilient to packet loss in both forward and reverse directions: (a) deadline-constrained and (b) deadline-unconstrained cases. Query aggregation workload.***

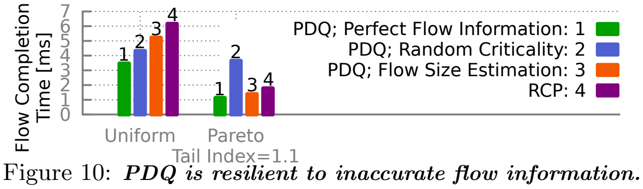


Figure 10: ***PDQ is resilient to inaccurate flow information. For PDQ without flow size information, the flow criticality is updated for every*** 50 ***KByte it sends. Query aggregation workload,*** 10 ***deadline-unconstrained flows with a mean size of*** 100 ***KByte. Flow-level simulation.***

6. MULTIPATH PDQ

Several recent works [17, 21] show the benefits of multipath TCP, ranging from improved reliability to higher network utilization. Motivated by this work, we propose Multipath PDQ (M-PDQ), which enables a single flow to be striped across multiple network paths.

When a flow arrives, the M-PDQ sender splits the flow into multiple subflows, and sends a SYN packet for each subflow. To minimize the flow completion time, the M-PDQ sender periodically shifts the load from the paused subflows to the sending one with the minimal remaining load. To support M-PDQ, the switch uses flow-level Equal-Cost Multi-Path (ECMP) to assign subflows to paths. The PDQ switch requires no additional modification except ECMP. The M-PDQ receiver maintains a single shared buffer for a multipath flow to resequence out-of-order packet arrivals, as done in Multipath TCP [21].

We illustrate the performance gains of M-PDQ using BCube [13], a data center topology that allows M-PDQ to exploit the path diversity between hosts. We implement BCube address-based routing to derive multiple parallel paths. Using ran-

[8] See the discussion in §2.1 of [20].

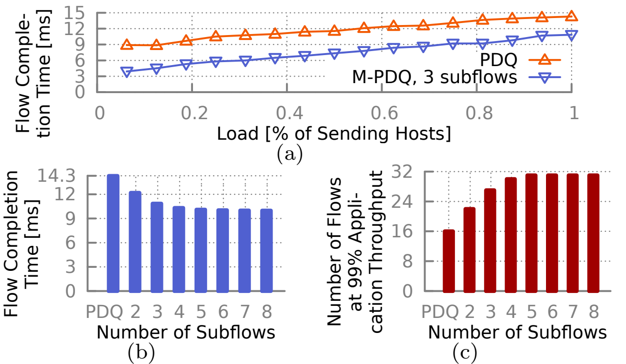


Figure 11: ***Multipath PDQ achieves better performance. BCube(2,3) with random permutation traffic. (a, b) deadline-unconstrained, (c) deadline-constrained flows.***

dom permutation traffic, Figure 11a demonstrates the impact of the system load on flow completion time of M-PDQ. Here, we split a flow into 3 M-PDQ subflows. Under light loads, M-PDQ can reduce flow completion time by a factor of two. This happens because M-PDQ exploits more links that are underutilized or idle than single-path PDQ. As load increases, these advantages are reduced, since even single-path PDQ can saturate the bandwidth of nearly all links. However, as shown in Figure 11a, M-PDQ still retains its benefits because M-PDQ allows a critical flow to have higher sending rate by utilizing multiple parallel paths. Finally, we fix the workload at 100% to stress the network (Figures 11b and 11c). We observe that M-PDQ needs about 4 subflows to reach 97% of its full potential. By allowing servers to use all four interfaces (whereas single-path PDQ can use only one), M-PDQ provides a significant performance improvement.

7. DISCUSSION

Fairness. One could argue the performance gains of PDQ over other protocols stem from the fact that PDQ unfairly penalizes less critical flows. Perhaps counter-intuitively, the performance gain of SJF over fair sharing does not usually come at the expense of long jobs. An analysis [4] shows that at least 99% of jobs have a smaller completion time under SJF than under fair sharing, and this percentage increases further when the traffic load is less than half.[9] Our results further demonstrate that, even in complex data center networks with thousands of concurrent flows and multiple bottlenecks, $85-95\%$ of PDQ's flows have a smaller completion time than RCP, and the worst PDQ flow suffers an inflation factor of only 2.57 as compared with RCP (Figure 8e). Moreover, unfairness might not be a primary concern in data center networks where the network is owned by a single entity that has full control of flow criticality. However, if desired, the operator can easily override the flow comparator to achieve a wide range of goals, including fairness. For example, to prevent starvation, the operator could gradually increase the criticality of a flow based on its waiting time. Using a fat-tree topology with 256 servers, Figure 12 demonstrates that this "flow aging" scheme is effective, reducing the worst flow completion time by ~48%, while the mean flow completion time increases only 1.7%.

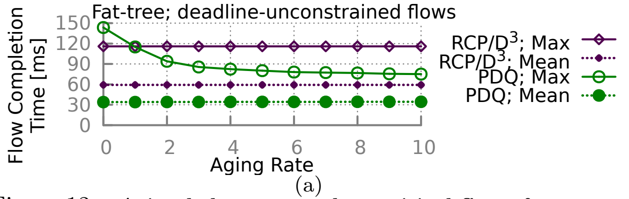


Figure 12: ***Aging helps prevent less critical flows from starvation and shortens their completion time. The PDQ sender increases flow criticality by reducing $\mathcal{T}_H$ by a factor of $2^{\alpha t}$, where α is a parameter that controls the aging rate, and t is the flow waiting time (in terms of 100 ms). Flow-level simulation; 128-server fat-tree topology; random permutation traffic.***

When flow completion time is not the priority. Flow completion time is not the best metric for some protocols. For example, real-time audio and video may require the ability to *stream*, or provide a number of flows with a fixed fraction of network capacity. For these applications, protocols designed for streaming transport may be a better fit. One can configure the rate controller (§3.3.3) to slice the network into PDQ-traffic and non-PDQ-traffic, and use some other transport protocol for non-PDQ-traffic. In addition, there are also applications where the receiver may not be able to process incoming data at the full line rate. In such cases, sending any rate faster than what receiver can process does not offer substantial benefits. Assuming the receiver buffers are reasonably small, PDQ will back off and allocate remaining bandwidth to another flow.

Does preemption in PDQ require rewriting applications? A preempted flow is paused (briefly), not terminated. From the application's perspective, it is equivalent to TCP being slow momentarily; the transport connection stays open. Applications do not need to be rewritten since preemption is hidden in the transport layer.

Incentive to game the system. Users are rational and may have an incentive to improve the completion time of their own flows by splitting each flow into small flows. While a similar issue happens to D^3, TCP and RCP[10], users in PDQ may have an even greater incentive, since PDQ does preemption. It seems plausible to penalize users for having a large number of short flows by reducing their flows' criticality. Developing a specific scheme remains as future work.

Deployment. On end hosts, one can implement PDQ by inserting a shim layer between the IP and the transport layers. In particular, the sender maintains a set of PDQ variables, intercepts all calls between IP and transport layer, attaches and strips off the PDQ scheduling header[11], and passes the packet segment to IP/transport layer accordingly. Additionally, the shim layer could provide an API that al-

[9] Assuming a M/G/1 queueing model with heavy-tailed flow distributions; see [4].

[10] In TCP/RCP, users may achieve higher aggregated throughput by splitting a flow into smaller flows; in D^3, users may request a higher rate than the flow actually needs.

[11] The 16-byte scheduling header consists of 4 fields, each occupying 4 bytes: $\mathcal{R}_H$, $\mathcal{P}_H$, $\mathcal{D}_H$, and $\mathcal{T}_H$. The PDQ receiver adds $\mathcal{I}_S$ and RTT_S to the header by reusing the fields used by $\mathcal{D}_H$ and $\mathcal{T}_H$. This is feasible because $\mathcal{D}_H$ and $\mathcal{T}_H$ are used only in the forward path, while $\mathcal{I}_S$ and RTT_S are used only in the reverse path. Any reasonable hashing that maps switch ID to 4-byte $\mathcal{P}_H$ should provide negligible collision probability.

lows applications to specify the deadline and flow size, or it could avoid the API by estimating flow sizes (§5.6). The PDQ sender can easily override TCP's congestion window size to control the flow sending rate. We note that PDQ requires only a few more variables per flow on end hosts. On switches, similar to previous proposals such as D^3, a vendor can implement PDQ by making modifications to the switch's hardware and software. Per-packet operations like modifying header fields are already implemented on most vendors' hardware (e.g., ASICs), which can be directly used by our design. The more complex operations like computing the aggregated flow rate and sorting/updating the flow list can be implemented in software. We note that PDQ's per-packet running time is $O(\kappa)$ for the top κ flows and $O(1)$ for the rest of the flows, where κ is a small number of flows with the highest criticality and can be bounded as in §3.3.1. The majority of the sending flows' scheduling headers would remain unmodified[12] by switches.

8. RELATED WORK

D^3: While D^3 [20] is a deadline-aware protocol that also employs explicit rate control like PDQ, it neither resequences flow transmission order nor preempts flows, resulting in a substantially different flow schedule which serves flows according to the order of their arrival. Unfortunately, this allows flows with large deadlines to hog the bottleneck bandwidth, blocking short flows that arrived later.

Fair Sharing: TCP, RCP [10] and DCTCP [3] all emulate fair sharing, which leads to suboptimal flow completion time.

TCP/RCP with Priority Queueing: One could use priority queuing at switches and assigning different priority levels to flows based on their deadlines. Previous studies [20] showed that, using two-level priorities, TCP/RCP with priority queueing suffers from losses and falls behind D^3, and increasing the priority classes to four does not significantly improve performance. This is because flows can have very different deadlines and require a large number of priority classes, while switches nowadays provide only a small number of classes, mostly no more than ten.

ATM: One could use ATM to achieve QoS priority control. However, ATM's CLP classifies traffic into only two priority levels, while PDQ gives each flow a unique priority. Moreover, ATM is unable to preempt flows (i.e., new flows cannot affect existing ones).

DeTail: In a recent (Oct 2011) technical report, Zats et al. propose DeTail [23], an in-network multipath-aware congestion management mechanism that reduces the flow completion time "tail" in datacenter networks. However, it targets neither mean flow completion time nor the number of deadline-missing flows. Unlike DeTail which removes the tail, PDQ can save ∼30% flow completion time on average (compared with TCP and RCP), reducing the completion time of almost every flow (e.g., $85\% - 95\%$ of the flows, Figure 8e). We have not attempted a direct comparison due to the very different focus and the recency of this work.

9. CONCLUSION

We proposed PDQ, a flow scheduling protocol designed to complete flows quickly and meet flow deadlines. PDQ provides a distributed algorithm to approximate a range of scheduling disciplines based on relative priority of flows, minimizing mean flow completion time and the number of deadline-missing flows. We perform extensive packet-level and flow-level simulation of PDQ and several related works, leveraging real datacenter workloads and a variety of traffic patterns, network topologies, and network sizes. We find that PDQ provides significant advantages over existing schemes. In particular, PDQ can reduce by ∼30% the average flow completion time as compared with TCP, RCP and D^3; and can support 3× as many concurrent senders as D^3 while meeting flow deadlines. We also design a multipath variant of PDQ by splitting a single flow into multiple subflows, and demonstrate that M-PDQ achieves further performance and reliability gains under a variety of settings.

[12] Until, of course, the flow is preempted or terminated.

10. REFERENCES

[1] Bro network security monitor. http://www.bro-ids.org.

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

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

[4] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *SIGMETRICS*, 2001.

[5] N. Bansal and M. Harchol-Balter. End-to-end statistical delay service under GPS and EDF scheduling: A comparison study. In *INFOCOM*, 2001.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[7] J. Brutlag. Speed matters for Google web search, 2009.

[8] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[9] N. Dukkipati, Y. Ganjali, and R. Zhang-Shen. Typical versus worst case design in networking. In *HotNets*, 2005.

[10] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev.*, 2006.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[12] 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 *SIGCOMM*, 2009.

[13] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[14] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. Technical report. http://arxiv.org/abs/1206.2057, 2012.

[15] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, 2002.

[16] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer, 2nd edition, 2002.

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

[18] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[19] 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 *SIGCOMM*, 2009.

[20] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[21] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, 2011.

[22] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *CoNEXT*, 2010.

[23] D. Zats, T. Das, and R. H. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. Technical report, Oct 2011.