# Preemptive Distributed Quick Flow Scheduling

By Matt Hoffman

Paper: Finishing Flows Quickly with Preemptive Scheduling by Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey

# Quick review:

In data centers, **buffering delay** can make up the vast majority of **latency**.

With no buffering, a typical RTT is **150 microseconds**.

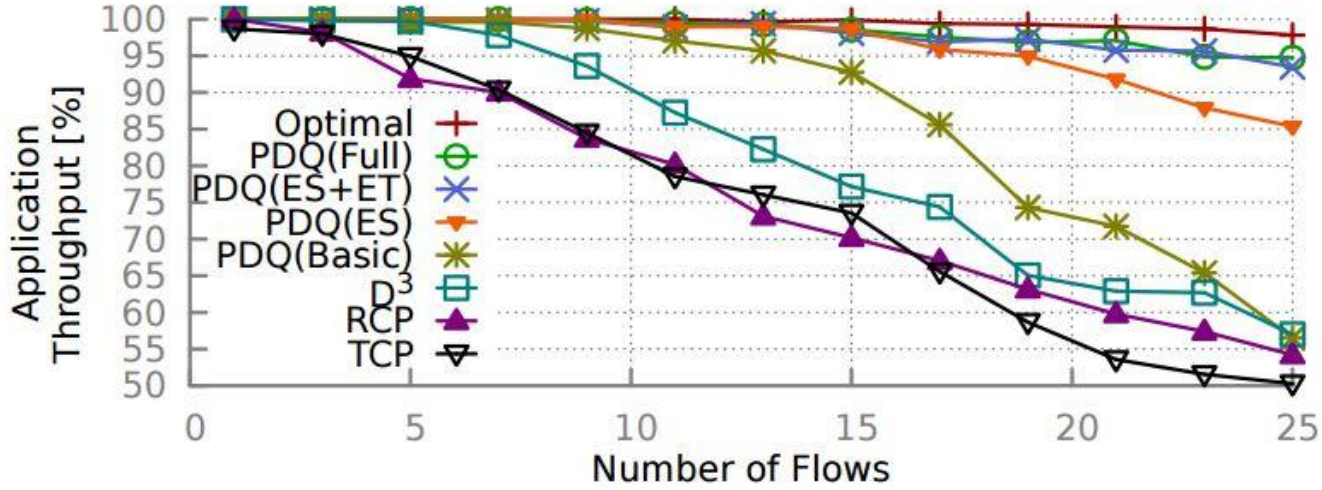
Two primary types of flows inside a database:

- Large background flows to synchronize databases
- Short flows with tight deadlines, such as divide-and-aggregate traffic

# The Problem:

TCP tends to fill available buffer space, which can cause greatly increased latency within a database.

A typical short flow in a database has a stringent **deadline**, and delays can cause user experiences to be greatly reduced (ex: WolframAlpha results).



# The Goal:

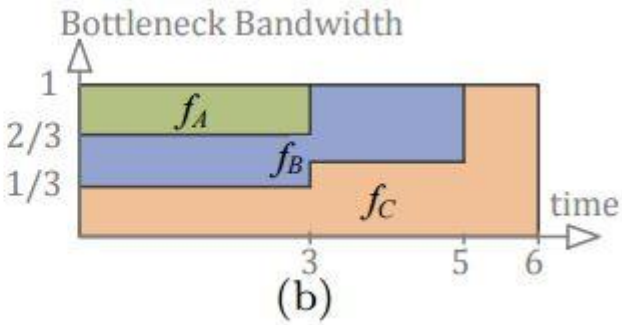
While **DCTCP** helps to solve the problem of unnecessary buffering delays, it does not attempt to minimize **completion times** or meet **flow deadlines**.

**Fair sharing** in a database is far from optimal.

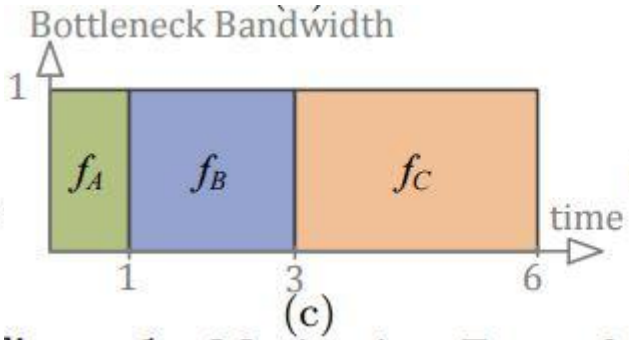
Instead, we would like to use **Earliest Deadline First (EDF)** to maximize the number of met deadlines, and **Shortest Job First (SJF)** to minimize flow completion time.

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

(a)




Fair sharing




EDF/SJF

# Implementing EDF and SJF

A full implementation of EDF and SJF assumes a **centralized scheduler** which knows the global state of the system.  This is not feasible in a large data center and would impede our goal of low latency.

Instead, we want scheduling to be done in a **distributed** way.

Note: We care about deadlines and jobs on a per flow basis, not per packet.

# Introducing PDQ

**Preemptive Distributed Quick (PDQ)** Flow Scheduling is designed to **approximate** EDF and SJF using only existing **First-In-First-Out (FIFO)** tail-drop queues.

Some properties of PDQ:

- Requires switches to **keep state** about each flow.
- Can operate as a shim layer between the IP and transport layers.
- Provides benefits for both deadline constrained traffic and regular traffic.

# How does PDQ work?

**Centralized example**

First, flows are prioritized by deadline (EDF). In the event of ties, job size is then used (SJF).

Whenever the network workload changes, we iterate through each flow in order of priority and allocate as much bandwidth as needed/available to that flow.

Flows with lower priority may end up with zero bandwidth, which means they are **paused** until enough resources are available.

# How does PDQ work?

## Centralized example

1. Initialize $B_e$ = Maximum Available Bandwidth of each link *e*
2. For each flow *i*, in increasing order of priority
    a. Let $P_i$ represent flow *i*'s path
    b. Let $R_i$ represent the maximum sending rate of flow *i*
    c. Let $A_i$ = min($R_i$, $B_e$ for each link *e* in Path $P_i$), which represents the bandwidth allocated to flow *i*
    d. For each link *e* in Path $P_i$, update the available bandwidth $B_e = B_e - A_i$

# How does PDQ work?

**In a distributed system**

We use **explicit feedback** in packet headers to propagate flow information. This feedback is returned to the sender in an ACK packet.

PDQ switches inform the sender to send data with a specific **rate** ($R > 0$) or to **pause** ($R = 0$) by annotating the header of the data/ACK packets.

Essentially: Instead of the switches enforcing bandwidth usage of each flow, the sender regulates its own bandwidth based on switch feedback.

# How does PDQ work?

**Pausing**

When a switch pauses a flow, it adds its ID to the packet header.  Other switches are able to delete information about that flow.

While paused, the sender sends a **probe packet** periodically to get updated rate information.  The sender then updates its sending rate based on feedback.

Since the Round Trip Time (RTT) in a database is extremely small, these probe packets are typically sent once every several RTTs.

If a 40 byte probe were sent every 150 microseconds (typical database RTT), that represents about **2.13 Mbs** of bandwidth (The paper incorrectly says 21.3 Mbps).

# How does PDQ work?

**Early Termination**

For deadline constrained flows, we want to be able to terminate flows whose demands exceed network capacity.

Minimizing missed deadlines in a dynamic setting is **NP-complete**, so the following criteria are used instead:

1. Deadline has already passed.
2. Remaining flow Transmission Time exceeds the time until the deadline.
3. The flow is paused and the time to deadline is less than one Round Trip Time

If any of these criteria are met, the flow should be terminated.  The **sender** is in change of determining when to terminate a flow.

# Other issues to address

**Switches maintaining state about each flow**

A switch needs to remember the most recent scheduling variables for each flow. These include: **Rate**, **Deadline**, **Transmission Time**, and **Round Trip Time**.

Each switch only needs to maintain the **2k** most critical flows, where **k** is the number of sending flows.

In practice, we have a worst case of **100** active flows at per server, translating to 12000 flows at each switch in a VL2 network.  Storing all these flows requires only **0.23 MB** of storage.

# Other issues to address

**Early Start**

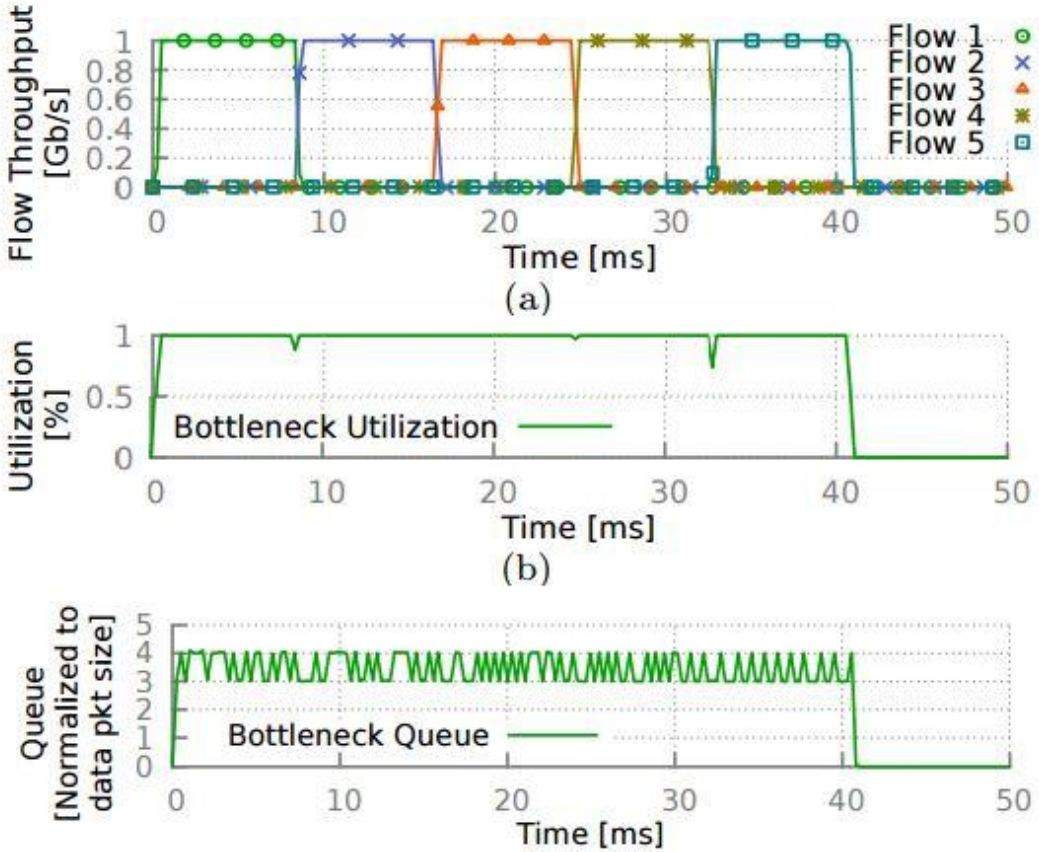
Since probes are not sent every RTT, it can take **several RTTs** before a paused flow is able to start sending data once bandwidth is available.  For many short flows, this can result in significant **lost bandwidth**.

Solution: Start the next set of flows before the current sending flows finish.  A PDQ switch can use the estimated Transmission Time and RTT of a flow to determine if it is almost complete.

# Other issues to address

**Early Start Performance**

PDQ is able to provide seamless flow switching, maintaining high throughput with a small queue size, and converges quickly.



# Other issues to address

**Dampening**

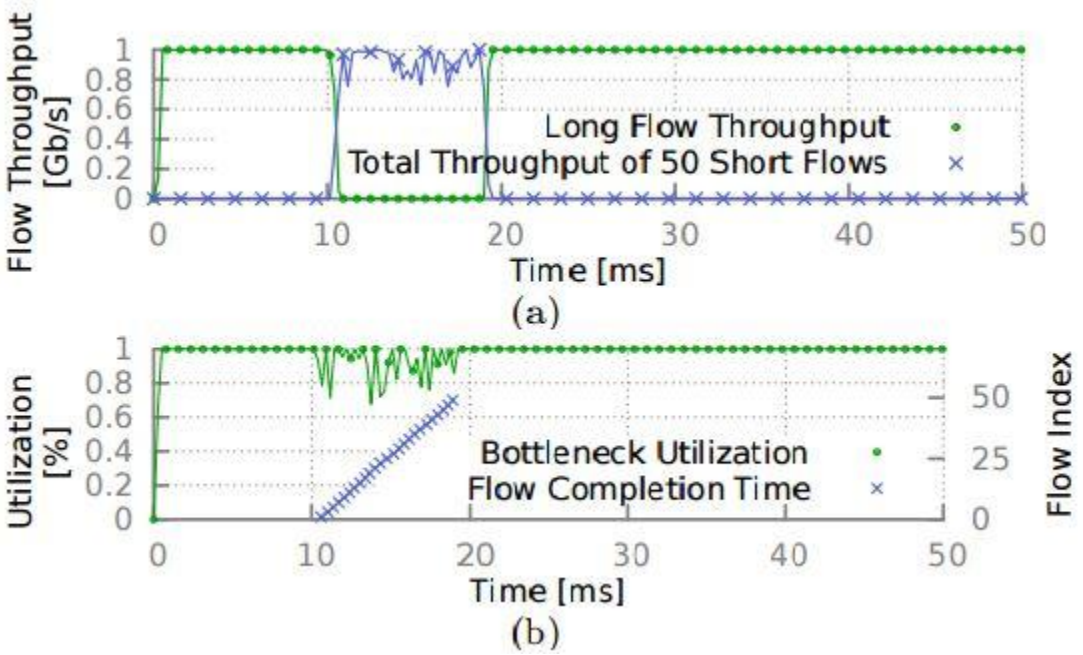
When a flow that is currently sending gets **preempted** by a more critical flow, PDQ will pause the current flow.  If bursts of flows arrive at the same time, this can cause temporary **instability**.

Solution: After a switch has accepted a flow, it can only accept other paused flows after a given small period of time.

# Other issues to address

**Dampening Performance**

PDQ is robust against a burst workload. These graphs show 50 short flows starting at the same time, preempting a large flow.



# PDQ Performance

The following metrics are taken in a two-level 12-server single-rooted tree with 1 Gbps link rates.

TCP comparisons are using TCP Reno with a small $RTO_{min}$ to avoid Incast.

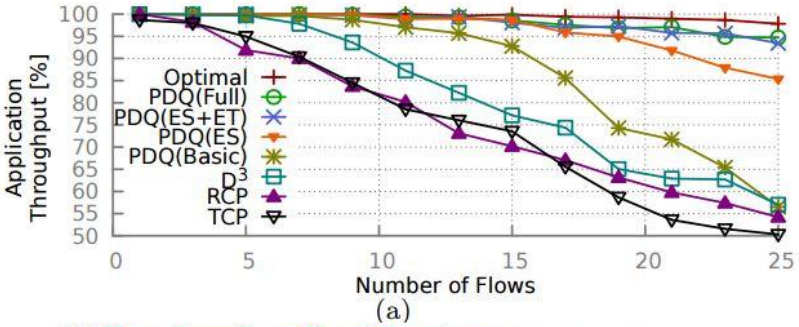
Application throughput is defined as the percent of query-aggregate flows that **met their deadlines**.

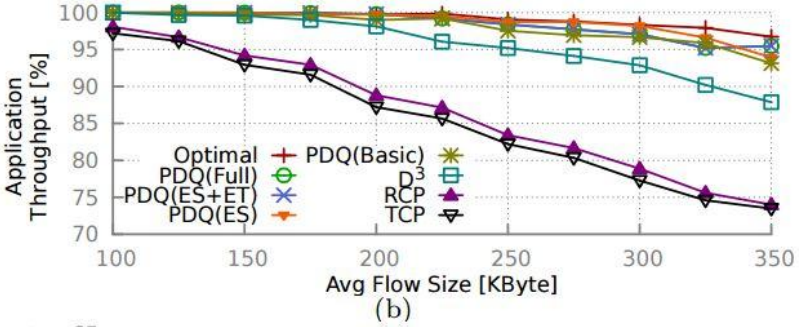
Flow deadlines are randomly distributed between 20 ms and 60 ms

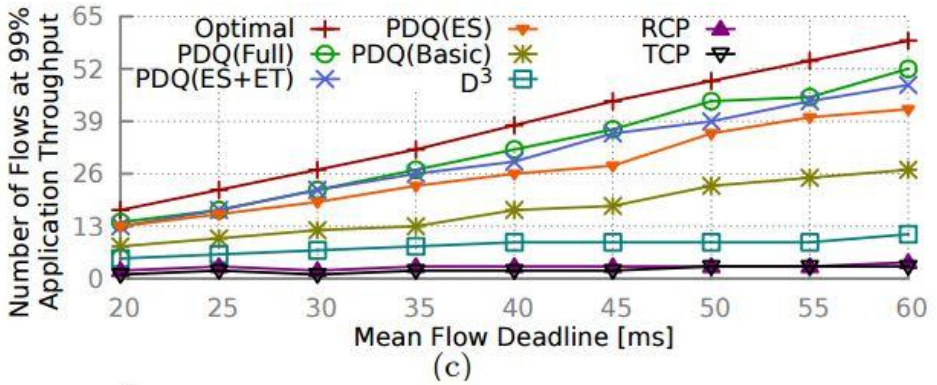
PDQ (full) refers to Early Start, Early Termination, and Probe Suppression (not sending probes every RTT)

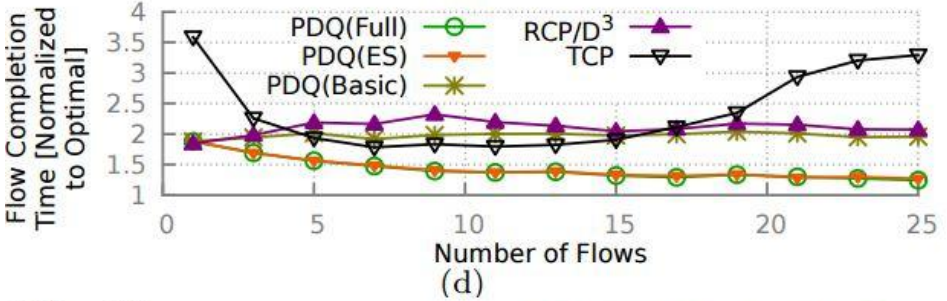
# PDQ Performance

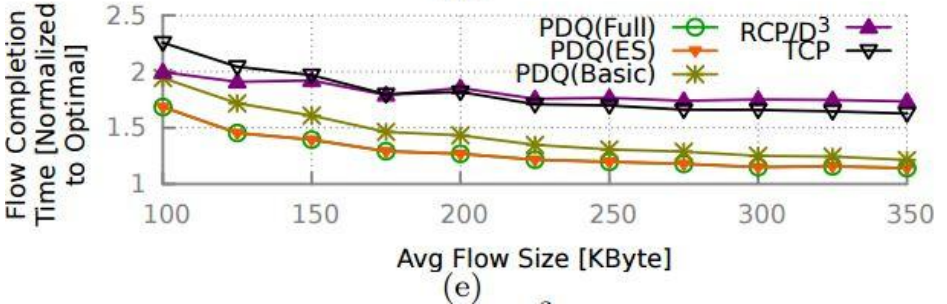
PDQ consistently outperforms TCP, and other transport protocols



# PDQ Performance

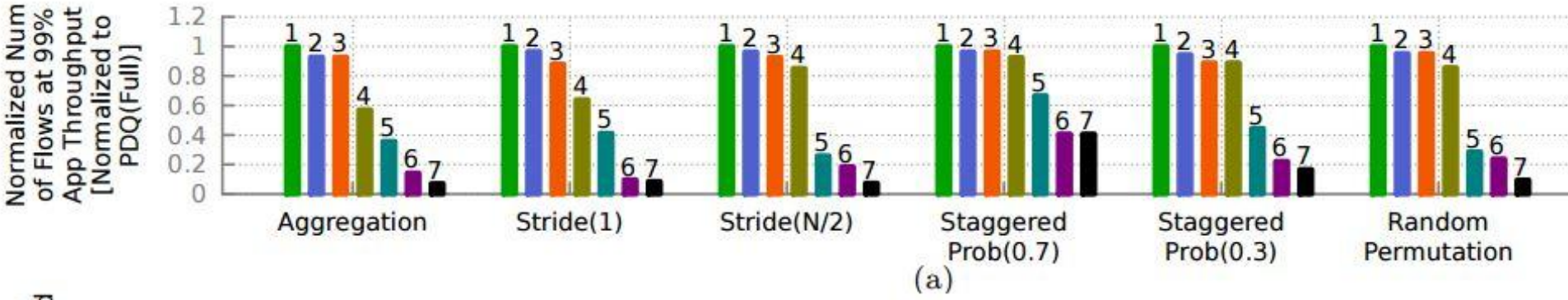
## Some more graphs

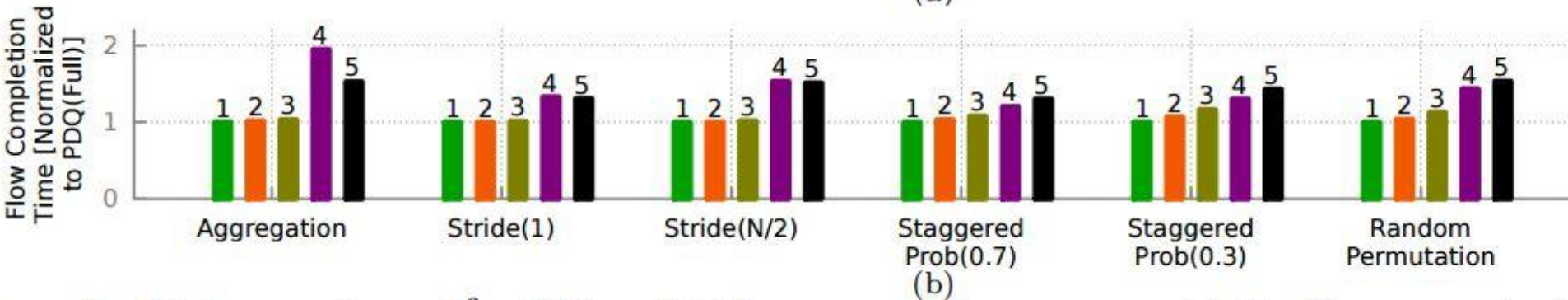
PDQ(Full): 1
PDQ(ES+ET): 2
PDQ(ES): 3
PDQ(Basic): 4
D3: 5
RCP: 6
TCP: 7

Top row

PDQ(Full): 1
PDQ(ES): 2
PDQ(Basic): 3
RCP/D3: 4
TCP: 5

Bottom row



# Discussion

1. PDQ gains performance at the cost of fairness.  Is fairness something that we should care more about?  Is PDQ more vulnerable to users “gaming” the system than TCP?

2. Not all traffic is represented by short flows with deadlines and large background flows.  Do other forms of traffic suffer under PDQ?