Queues don’t matter when you can JUMP them

Matthew P. Grosvenor  Malte Schwarzkopf  Ionel Gog
Andrew W. Moore  Robert N. M. Watson  Steven Hand  Jon Crowcroft
Datacenter Networks
Datacenter Networks

- Commodity hardware
Datacenter Networks

- Commodity hardware
- Static network topology
Context

Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain

© Google google.com/datacenters
Context

Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation
Datacenter Networks
- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation
- Statistically Multiplexed
Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation
- Statistically Multiplexed
Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation
- Statistically Multiplexed

NO GUARANTEES
Illustrative experiment

- 12 node 10G test cluster
- 8 nodes Hadoop MR
- 2 nodes PTPd
- 2 nodes memcached
permanent queues build up in the switch. At this point, the impact of retransmissions degrades throughput by 30% (vs 20% for the UDP and TCP implementations). By sending fixed-size bursts followed by a 25 ms pause, we observe a 7 µs (1 in 1000) increase in the request latency. To demonstrate the usefulness of the UDP and TCP implementations, we built a simple distributed two-phase commit (2PC) application, with explicit acknowledgements and retransmissions. The application communicates over TCP or UDP, offering reliable delivery. The coordinator can send its messages by UDP broadcast when another system requests it. For Hadoop, the metric of interest is the job run-time. In Hadoop to memcached interference mitigated by Ethernet Flow Control (ECN) and Data Center TCP, the coordinator can offer constant two-phase commit in RAMCloud. The three applications are: an ideal case (25 µs), a contended case (75 µs), and one for memcached (260 µs). For memcached, the metric is the request latency. Figure 5 shows interference ratios greater than 12:1, with Ethernet Flow Control providing better interference control than other schemes. Hosts and switches issue special congestion notifications. In Figure 6, we show the request rate for one coordinator and seven servers as a function of network interference. We exhibit better throughput even at high levels of network interference. Congestion Notification (ECN) and Data Center TCP offer reliable delivery. The PTP sync offset is close to zero, indicating good latency performance.
The application communicates over TCP or over UDP, depending on whether the request is to a coordinator or not. Hadoop’s performance is not noticeably affected by interference when it is running alone on an idle network. For memcached, the metric of interest is the request latency. Figure 5 shows the request rate for one coordinator and seven servers as a function of network interference. All cases are normalized to the ideal case, which offers reliable delivery and low latency. Since interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference. All cases are normalized to the ideal case, which offers reliable delivery and low latency.

PTPd and memcached in isolation offer reliable delivery and low latency. PTPd achieves 6.5 Mbps of PTP sync offset: close to zero = good, while memcached achieves 6.2 Mbps of memcached latency: lower = good. Furthermore, memcached latency degrades as its messages “jump the queue”. At high interference ratios (e.g., 30% throughput drop), memcached’s unique advantage over TCP is its capability to detect congestion and slow down requests, whereas TCP continues to send packets at a rate that may exceed the switch buffer capacity. Ethernet Flow Control achieves 6.5 Mbps of memcached latency: lower = good, with a maximum throughput of 150 Mbps. However, PTPd’s reliable delivery and low latency make it a more reliable choice for applications requiring high availability and low latency. DCTCP achieves 6.5 Mbps of memcached latency: lower = good. Several readily deployable congestion control schemes are compared with PTPd and memcached in isolation, with a focus on interference control. We discuss each result in turn. Hadoop’s performance is not noticeably affected by interference when it is running alone on an idle network. PTP sync offset: close to zero = good.
10,000 requests/sec observed on an idle network. By another that sends fixed-size bursts followed by a 25ms phase atomic-commit (2PC) application.

§

Figure 5: The application communicates over TCP or over UDP. PTPd and memcached in isolation.

6.5\( \times 10^3 \) packets/s reached the switch, which is a 200% increase from the baseline (5Gb/s) traffic. This optimization yields a 6.5\( \times 10^3 \) improvement over both TCP and UDP. This is due to the switch's implementation of Ethernet Flow Control (ECN) and Data Center TCP (DCTCP).

The root mean square (RMS) of each application-specific interference ratios (top), with the in-\( \times \)side interference control than other schemes. In the baseline case, with no interfering traffic from Hadoop, the interference ratio is below 1.0, as we will further show in §4.6.

Since §6.2, we show the request rate for one coordinator. This coordinator offers reliable delivery, the coordinator can send its messages by UDP broadcast when another application generates a constant 10Gb/s of UDP traffic and with explicit acknowledgements and retransmissions.

Distributed Atomic Commit

Several readily deployable congestion control schemes exhibit better performance, e.g., like the one shown in Figure 6.

We discuss each result in turn.

In the baseline case, each application running alone on an idle network. The request latency for PTPd is the time synchronization offset and for memcached it is the request latency. Figure 6 shows the waiting time, for PTPd it is the time synchronization offset and for memcached it is the request latency. All cases are normalized to the ideal case, which has each application running alone on an idle network.

We repeat the multi-application experiment within two network epochs. This is not the same as the root mean square (RMS) of each application-specific interference ratios (top), which measures prediction accuracy. The waiting time is a statistical measure of the difference between the actual and predicted waiting time. RMS is a statistical measure of the difference between the actual and predicted waiting time. RMS is a statistical measure of the difference between the actual and predicted waiting time. RMS is a statistical measure of the difference between the actual and predicted waiting time.
What’s the problem?
Network Interference

Switch

- Hadoop
- memcached
- PTPd
Network Interference

Queuing caused by Hadoop

- Hadoop
- memcached
- PTPd

Switch
Network Interference

Queuing caused by Hadoop

Hadoop
memcached
PTPd

Delaying traffic from PTPd and memcached
Network Interference:

Congestion from one application causes queuing that delays traffic from another application.*

*possibly related
Solving network interference?

Borrow some old ideas

Packet by Packet Generalised Processor Sharing (PGPS)

(Weighted) Fair Queuing (WFQ)

Differentiated Service Classes (diff-serv)

Parekh-Gallager Theorem
Solving network interference?

Borrow some old ideas

Packet by Packet Generalised Processor Sharing (PGPS)

(Weighted) Fair Queuing (WFQ)

Differentiated Service Classes (diff-serv)

Parekh-Gallager Theorem

Apply in a new context: Datacenters
Opportunities & Constraints

Datacenter Opportunities
Datacenter Opportunities

- Static network
- Single admin domain
- Cooperation
Opportunities & Constraints

Datacenter Opportunities
- Static network
- Single admin domain
- Cooperation

Deployability Constraints
Opportunities & Constraints

Datacenter Opportunities
- Static network
- Single admin domain
- Cooperation

Deployability Constraints
- Unmodified applications
- Unmodified kernel code
- Commodity hardware
Understanding delays

Delay type I - Queuing Delay ($D_q$)
Understanding delays

Delay type II - **Servicing Delay ($D_s$)**

Switch
Understanding delays

Delay type II - **Servicing Delay (D_s)**

Switch

1

2

3

4

D_s
Understanding delays

Delay type II - Servicing Delay ($D_s$)

Switch
Understanding delays

Delay type II - **Servicing Delay ($D_s$)**

Switch

- Input 1
- Input 2
- Input 3
- Input 4

$D_s$
Understanding delays

Delay type II - Servicing Delay ($D_s$)

Switch
Understanding delays

Switch

D_q

D_s

4 3 2 1
Understanding delays

Servicing delay causes queuing delay

\[ D_q \]

\[ D_s \]

Switch
Eliminating Queuing Delay
Eliminating Queuing Delay
If we can find a bound for servicing delay, we can rate-limit hosts so that they never experience queuing delay.
Assume sending hosts $n = 4$
Assume sending hosts $n = 4$

1

2

3

4

Switch

Assume edge speed $R = 10\, \text{Gb/s}$
Assume sending hosts $n = 4$

Assume packet size $P = 1500\text{B}$

Assume edge speed $R = 10\text{Gb/s}$
Calculating Service Delay

Assume edge speed \( R = 10 \text{ Gb/s} \)

Assume packet size \( P = 1500 \text{ B} \)

Assume sending hosts \( n = 4 \)

Delay per packet
\[ = \frac{P}{R} \]
\[ = \frac{1500 \text{ B}}{10 \text{ Gb/s}} \]
\[ = 1.5 \mu s \]
Calculating Service Delay

Assume sending hosts $n = 4$

- Delay per packet = $P/R = 1500B / 10Gb/s = 1.5 \mu s$

Total delay = $n \times$ per packet
- = $4 \times 1.5 \mu s$
- = $6 \mu s$

Assume edge speed $R = 10Gb/s$

Switch

Assume packet size $P = 1500B$
Calculating Servicing Delay

Servicing delay \( = n \times \frac{P}{R} \)
Calculating Servicing Delay

servicing delay* = n \times \frac{P}{R}

Where

n - number of hosts
P - bytes sent
R - edge speed

*Assuming a fair scheduler
Calculating Servicing Delay

network**
servicing delay* = n × \frac{P}{R}

Where

n - number of hosts
P - bytes sent
R - edge speed

*Assuming a fair scheduler
**Apply hose constraint model
Key Idea

Rate-Limiting

1. Network is idle
2. Hosts send $\leq P$ bytes
3. Wait $(n \times P / R)$ secs
4. Goto 1

Network Epoch
Eliminating Synchronization
Eliminating Synchronization

Epoch

Epoch

4 packets
Eliminating Synchronization

Epoch

Epoch

4 packets
Eliminating Synchronization

≈ 8 packets per epoch
network epoch = $2n \times \frac{P}{R}$

Where

- n - number of hosts
- P - bytes sent
- R - edge speed
- $2$ - mesochronous compensation
The dark side of network epoch

throughput = \frac{R}{2n}

Where

n is the number of hosts
R is the edge speed
The dark side of network epoch

\[ \text{throughput} = \frac{10 \text{Gb/s}}{2 \times 1000} = 5 \text{Mb/s} \]

Where

- \( n = 1000 \) hosts
- \( R = 10 \text{ Gb/s} \)
The dark side of network epoch

throughput\(^*\) = \frac{10\text{Gb/s}}{2 \times 1000} = 5\text{Mb/s}

Where

\(n = 1000\) hosts

\(R = 10\ \text{Gb/s}\)

\(^*\)at guaranteed latency!
solution: assume there is no problem?
Changing the assumptions

Pessimistic assumption of 4:1

1
2
3
4
What if we assume 2:1?
What if we assume 2:1? Hosts can send 2x the rate!
Changing the assumptions

What if we assume 1:1?
What if we assume 1:1? **Hosts can send 4x the rate!**
Changing the assumptions

What if assumption is wrong?

1

2

3

4
Changing the assumptions

What if assumption is wrong? Queuing will happen!
Which assumption?

4:1
Which assumption?

Rate limit

low throughput
Which assumption?

Rate limit

Latency Distribution

Guaranteed latency
Which assumption?
Which assumption?

Rate limit

line rate throughput
Which assumption?

Rate limit

Latency Distribution

Line rate throughput

1:1

No latency guarantee
Which assumption?
Which assumption?

4:1

2:1

1:1
QJump with priorities

- Low latency
- High priority
- Low rate-limit

Diagram: Flow of QJump with priorities.
QJump with priorities

Medium Latency
Low latency

Medium priority
Medium rate-limit
QJump with priorities

- High Throughput
- Medium Latency
- Low latency

Low priority
No rate-limit
Queue Jumping!

- High Throughput
- Medium Latency
- Low latency
QJump with priorities

Queues don’t matter when you can Jump them!
Prioritization

Use **hardware priorities** to run different **QJump** levels together, but **isolated*** from each other.

* from layers below
### Implementation

```c
long epoch_cycles = to_cycles(network_epoch);
long timeout = start_time;
long bucket[NUM_QJUMP_LEVELS];

int qJumpRateLimiter(struct sk_buff* buffer) {
    long cycles_now = asm("rdtsc"); /* read cycle ctr */
    int level = buffer->priority;
    if (cycles_now > timeout) { /* new token alloc? */
        timeout += epoch_cycles;
        bucket[level] = tokens[level];
    }
    if (buffer->len > bucket[level]) {
        return DROP; /* tokens for epoch exhausted */
    }
    bucket[level] -= buffer->len;
    sendToHWQueue(buffer, level);
    return SENT;
}
```

In practice, packets are rarely dropped because if the driver (not, the packet is dropped of bytes to itself (lines 8–10). It then checks to see if new epoch has begun. If so, it issues a fresh allocation in the application or use our application utility to assign its critical path. For granularity timing, but uses only one instruction on the altime counter (line 1). We then synthesize a clock from the CPU revert the network epoch value from seconds into cycles quantify time in cycles. This requires us to initially configuration. To keep the rate-limiter efficient, all operations VLAN support to send layer 2 priority-tagged packets. Inserted and removed at runtime, making them flexible and control (TC) subsystem to rate-limit packets. TC mod- in software. A result, per host are sufficient when using IEEE 802.1Q priority- for each flow, each host only needs one coarse-grained that use rate-limiters. Instead of requiring a rate-limiter deployed as an addition to the kernel network egress path deployed as a component in the hypervisor and levels. In a multi-tenant environment, the rate-limiter is mission control to the network, and an application utility to configure unmodified applications to use the network epoch (Q). We call the assignment of a UMP has two components: a rate-limiter to provide ad- level is a function of the sum of the 0.5% overhead. When a new packet arrives at the rate-limiter, it is clas- implementation shows our a custom rate-limiter implemen-
Implementation

Linux TC

```c
long bucket[NUM_QJUMP.Levels];

int qJumpRateLimiter(struct sk_buff* buffer) {
    long cycles_now = asm("rdtsc"); /* read cycle ctr */
    int level = buffer->priority;
    if (cycles_now > timeout) { /* new token alloc? */
        timeout += epoch_cycles;
        bucket[level] = tokens[level];
    }
    if (buffer->len > bucket[level]) {
        return DROP; /* tokens for epoch exhausted */
    }
    bucket[level] -= buffer->len;
    sendToHWQueue(buffer, level);
    return SENT;
}
```

[Image 15x701 to 62x756]
Linux TC

```c
long bucket[NUM_QJUMP_LEVELS];

~36 cycles / packet

int level = buffer->priority;
if (cycles_now > timeout) {
    timeout += epoch_cycles;
    bucket[level] = tokens[level];
}
if (buffer->len > bucket[level]) {
    return DROP; /* tokens for epoch exhausted */
}
bucket[level] -= buffer->len;
sendToHWQueue(buffer, level);
return SENT;
```
Linux TC

```c
long bucket[NUM_QJUMP_LEVELS];
```

~36 cycles / packet

```c
int level = buffer->priority;
if (cycles_now > timeout) { /* new token alloc? */
    bucket[level] -= buffer->len;
    sendToHWQueue(buffer, level);
    return SENT;
}
if (buffer->len > bucket[level]) {
    return DROP; /* tokens for epoch exhausted */
}
```

Smart Buffer Sizing

Implementation
Implementation

Linux TC

```c
long bucket[NUM_QJUMP_LEVELS];
```

~36 cycles / packet

```c
int level = buffer->priority;
if (cycles_now > timeout) { /* new token alloc? */
    }
if (buffer->len > bucket[level]) {
    bucket[level] -= buffer->len;
    sendToHWQueue(buffer, level);
    return SENT;
}
```

Smart Buffer Sizing

Unmodified Applications

```
```

Implementation

Linux TC

```c
long bucket[NUM_QJUMP_LEVELS];
```

~36 cycles / packet

```c
int level = buffer->priority;
if (cycles_now > timeout) { /* new token alloc? */
    }
if (buffer->len > bucket[level]) {
    bucket[level] -= buffer->len;
    sendToHWQueue(buffer, level);
    return SENT;
}
```

Smart Buffer Sizing

Unmodified Applications

```
```
Implementation

Linux TC

```c
3 long bucket[NUM_QJUMP_LEVELS];
```

~36 cycles / packet

```c
7 int level = buffer->priority;
8 if (cycles_now > timeout) { /* new token alloc? */
```

Smart Buffer Sizing

```c
11 }
12 if (buffer->len > bucket[level]) {
```

Unmodified Applications

```c
15 bucket[level] -= buffer->len;
```

802.1 Q

```c
18 }
```
How well does it work?

![Graph showing time vs. time since start with Hadoop enabled]
How well does it work?

Figure 5: Comparison of performance with and without QJump for Hadoop. The graph shows the time taken for requests to complete, with and without QJump.

Figure 6: Comparison of performance with and without QJump for memcached. The graph shows the time taken for requests to complete, with and without QJump.

**Time [µs]**

- **+ Hadoop**
- **+ Hadoop, with QJump**

**Time since start [sec]**

Since interference is transient in these experiments, we measure the root mean square (RMS) of each application-specific metric. The RMS makes it possible to compare different applications, despite their different rates.

Outperforms Alternatives

One of the best features of QJump is its ability to improve the performance of a wide range of applications. We test QJump with a variety of applications, including Hadoop, memcached, and even PTPd and memcached in isolation.

The results show that QJump improves the performance of all these applications. For Hadoop, we see a significant improvement in the time taken for requests to complete. We also see a decrease in the time taken for memcached requests to complete.

We also test QJump with PTPd and memcached in isolation. The results show that QJump improves the performance of both applications.

Furthermore, QJump achieves better performance than other schemes. For example, the UDP-over-TCP implementation does not offer constant two-phase commit. This is not the same as the root mean square error (RMSE), which measures prediction accuracy.

Finally, we test QJump with six cases: an ideal case, a contended case, and one for each of the four schemes used to mitigate network interference. The results show that QJump achieves better performance than other schemes.

Several readily deployable congestion control schemes exist, including Ethernet Flow Control (802.1x), Explicit Congestion Notification (ECN), and DCTCP. We repeat the multi-application experiment for two of these schemes, including Ethernet Flow Control (802.1x) and DCTCP.
How does it compare?

Figure 7: Normalized RMS application metric

- Hadoop runtime
- PTPd sync. offset
- memcached req. latency

good = close to 1
How does it compare?

![Graph showing normalized RMS application metric for Hadoop runtime, PTPd sync. offset, and memcached req. latency. The graph compares ideal and contended cases.](image)
How does it compare?

![Graph showing normalized RMS app. metric for Hadoop runtime, PTPd sync. offset, and memcached req. latency.]

- Hadoop runtime: 318
- PTPd sync. offset: 12614
- memcached req. latency

**Topologies:**
- Ideal
- Contended
- Eth. Flow Cnt.

**Notes:**
- The graph indicates a good performance for the Contended and Eth. Flow Cnt. topologies.
- Further details and analysis are provided in the text, including the effectiveness of WRED and the impact of ECN in resolving network interference.
How does it compare?

Figure 7: Normalized RMS app. metric for different cases.

- **Ideal** runtime
- **Contended** runtime
- **Eth. Flow Cnt.** runtime
- **ECN (WRED)** runtime

### Performance Comparison

- **Hadoop runtime**
- **PTPd sync. offset**
- **memcached req. latency**

**Key Observations**

- **Good** performance in terms of normalized RMS metric.
- Comparative analysis shows variations in performance across different conditions.

### Figures

- **Figure 8**: Additional graphs available for both workloads.

### References

- [http://www.cl.cam.ac.uk/netos/camsas/qjump](http://www.cl.cam.ac.uk/netos/camsas/qjump)
- [ns2](http://www.cl.cam.ac.uk/netos/camsas/qjump) simulation provided.
How does it compare?

![Diagram showing performance comparison between Hadoop, PTPd, and memcached.](Diagram)

- **Hadoop**: Blue bars for runtime performance, showing good results.
- **PTPd sync. offset**: Orange bars indicating offset performance, with values around 318 and 12614.
- **memcached**: Red bars for required latency, showing performance.

**Legend**:
- **Ideal** performance
- **Contended** performance
- **Eth. Flow Cnt.** congestion
- **ECN (WRED)** congestion avoidance
- **DCTCP** congestion

**Key Points**:
- Hadoop's performance remains unlimited positive influence on memcached, but increases based on distribution of flow sizes in specific configurations.
- ECN very effectively resolves interference experienced by PTPd and memcached.
- DCTCP uses the rate at which ECN markings are received to build an estimate of network congestion.

**Figure 7**: Normalized RMS app. metric

**Figure 8**: Comparison of different marking thresholds pairs, ranging between [5, 10] and lower.

**Further Reading**:
- The pFabric architecture has been shown to schedule flows close to optimally, requiring modifications both at the switches and work layer mechanism in which switches indicate queueing to end hosts by marking TCP packets.
- Our Arista 7050 switch implements ECN with Weighted Random Early Detection (WRED).
- The effectiveness of WRED depends on an administrator correctly configuring upper and lower ECN marking thresholds pairs, ranging between [5, 10] and [2560, 5120] (upper, lower), in packets.

**Additional Resources**:
- Early Congestion Notification (ECN)
- Datacenter TCP (DCTCP)

**Notes**:
- Despite its simplicity, the pFabric architecture has been shown to schedule flows close to optimally, achieving average and 99th percentile flow completion times (FCTs). Although it works best on short flows, it works well for both workloads.
- We configured DCTCP with the recommended marking thresholds of [65, 65].
- We also run the same workloads derived from different loads, average FCTs for small flows (0kB, 100kB], and the average FCTs for large flows (10MB, 100MB] and the average FCTs for small flows (10MB, 100MB].

**Equation**:
- For both workloads, we normalize flows to their average FCTs and data mining.

**Further Exploration**:
- An extended set of graphs for both workloads is available at [http://www.cl.cam.ac.uk/netos/camsas/qjump](http://www.cl.cam.ac.uk/netos/camsas/qjump).

**Conclusion**:
- The graph shows that Hadoop's performance remains close to or better than pFabric's, especially on the web search workload.
How does it compare?

Normalized RMS app. metric

Ideal
Contended
Eth. Flow Cnt.
ECN (WRED)

Hadoop runtime
PTPd sync. offset
memcached req. latency

good

DCTCP*

*currently requires kernel patch
How does it compare?

Figure 7: Normalized RMS app. metric

- Ideal
- Contended
- Eth. Flow Cnt.
- ECN (WRED)
- DCTCP*
- QJUMP

**Hadoop runtime**
- Good

**PTPd sync. offset**
- 318
- 12614

**memcached req. latency**
- Currently requires kernel patch
Impact of ECN and DCTCP

- ECN (WRED)
- Early Detection (WRED)
- Early Congestion Notification (ECN)

Hadoop's bulk data transfers are affected by network congestion. DCTCP reduces the variance in PTPd synchronization by applying its pause frames and rate control mechanisms. We configured DCTCP with recommended settings for better performance.

*Currently requires kernel patch*
QJump applies datacenter simplifications to QoS rate calculations.
QJump applies datacenter simplifications to QoS rate calculations.

It provides service levels ranging from **guaranteed latency** through to line-rate throughput.
Conclusions

QJump applies datacenter opportunities to simplify QoS rate calculations.

It provides service levels ranging from guaranteed latency through to line-rate throughput.

It can be deployed using **without** modifications to applications, kernel code or hardware.
Queues don’t matter when you can JUMP them!

Matthew P. Grosvenor Malte Schwarzkopf Ionel Gog Robert N. M. Watson
Andrew W. Moore Steven Hand† Jon Crowcroft
University of Cambridge Computer Laboratory
† now at Google, Inc.

Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion from throughput-intensive applications causes queuing that delays traffic from latency-sensitive applications. To mitigate network interference, QJUMP applies Internet QoS-inspired techniques to datacenter applications. Each application is assigned to a latency sensitivity level (or class). Packets from higher levels are rate-limited in the end host, but once allowed into the network can “jump-the-queue” over packets from lower levels. In settings with known node counts and link speeds, QJUMP can support service levels ranging from strictly bounded latency (but with low rate) through to line-rate throughput (but with high latency variance).

We have implemented QJUMP as a Linux Traffic Control module. We show that QJUMP achieves bounded latency and reduces in-network interference by up to 300×, outperforming Ethernet Flow Control (802.3x), ECN (WRED) and DCTCP. We also show that QJUMP improves average flow completion times, performing close to or better than DCTCP and pFabric.

1 Introduction

Many datacenter applications are sensitive to tail latencies. Even if as few as one machine in 10,000 is a straggler, up to 18% of requests can experience high latency [13]. This has a tangible impact on user engagement and thus potential revenue [8, 9].

One source of latency tails is network interference from throughput-intensive applications that cause queuing that extends memcached request latency tails by 85 times the interference-free maximum (§2).

If memcached packets can somehow be prioritized to “jump-the-queue” over Hadoop’s packets, memcached will no longer experience latency tails due to Hadoop. Of course, multiple instances of memcached may still interfere with each other, causing long queues or incast collapse [10]. If each memcached instance can be appropriately rate-limited at the origin, this too can be mitigated.

These observations are not new: QoS technologies like DiffServ [7] demonstrated that coarse-grained classification and rate-limiting can be used to control network latencies. Such schemes struggled for widespread deployment, and hence provided limited benefit [12]. However, unlike the Internet, datacenters have well-known network structures (i.e. host counts and link rates), and the bulk of the network is under the control of a single authority. In this environment, we can enforce system-wide policies, and calculate specific rate-limits which take into account worst-case behavior, ultimately allowing us to provide a guaranteed bound on network latency.

QJUMP implements these concepts in a minimal rate-limiting Linux kernel module and application utility. QJUMP has four key features: It:

1. resolves network interference for latency-sensitive applications without sacrificing utilization for throughput-intensive applications;
2. offers bounded latency to applications requiring low-rate, latency-sensitive messaging (e.g. timing, consensus and network control systems);
3. is simple and immediately deployable, requiring no changes to hardware or application code; and
4. performs close to or better than connecting sys-
Want to know more?

<table>
<thead>
<tr>
<th>Setup</th>
<th>50th%</th>
<th>99th%</th>
</tr>
</thead>
<tbody>
<tr>
<td>one host, idle network</td>
<td>85</td>
<td>126μs</td>
</tr>
<tr>
<td>two hosts, shared switch</td>
<td>110</td>
<td>130μs</td>
</tr>
<tr>
<td>shared source host, shared egress port</td>
<td>228</td>
<td>268μs</td>
</tr>
<tr>
<td>shared dest. host, shared ingress port</td>
<td>125</td>
<td>278μs</td>
</tr>
<tr>
<td>shared host, shared ingress and egress</td>
<td>221</td>
<td>229μs</td>
</tr>
<tr>
<td><strong>two hosts, shared switch queue</strong></td>
<td><strong>1,920</strong></td>
<td><strong>2,100μs</strong></td>
</tr>
</tbody>
</table>

---

Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. To mitigate network interference, QJUMP applies Internet QoS-inspired techniques to datacenter applications. Each application is assigned to a latency sensitivity level (or class). Packets from higher levels are rate-limited in the end host, but once allowed into the network can “jump-the-queue” over packets from lower levels. In settings with known node counts and link speeds, QJUMP can support service levels ranging from strictly bounded latency (but with low rate) through to line-rate throughput (but with high latency variance).

We have implemented QJUMP as a Linux Traffic Control module. We show that QJUMP achieves bounded latency and reduces in-network interference by up to 300×, outperforming Ethernet Flow Control (802.3x), ECN (WRED) and DCTCP. We also show that QJUMP improves average flow completion times, performing close to or better than DCTCP and pFabric.

1 Introduction

Many datacenter applications are sensitive to tail latencies. Even if as few as one machine in 10,000 is a straggler, up to 18% of requests can experience high latency [13]. This has a tangible impact on user engagement and thus potential revenue [8, 9].

One source of latency tails is network interference from throughput-intensive applications.
Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion pacings, essiations, esifications, ong level ts, and nder can s. In set- ing QJUMP bounded through- lic Con- bounded y up to 302.3x), QJUMP forming a high lage-er, policies account provide QJUMP limiting QJUMP. 1. res ap the 2. off lo co 3. is no 4. me

QJUMP fixes Naiad barrier synchronization from 824µs in the shared case to 476µs, a nearly 2-

case do not completely agree due of randomness in the load generated. In this experiment, memcached is deployed, we assign each running alone on an

to the highest

Data Center Interference

Resolves Network Interference

Ping

Multi-application Environment

Hadoop.

Performance

We also show that in a realistic multi-application setting, performance is rate-limited. We measure in-network latency for the baseline distribution, despite sharing the network with baseline objects and Hadoop for batch data analysis. Since resolv-

effects are close to the ideal. The median latency improves

due of randomness in the load generated. We avoid sharing hosts between applications in these ex-

Application Setting

RPC vs. Bulk Transfer

To provide an additional view, we show a timeline of average request latency: CDF over 10k samples.

Figure 3:

0.0 0.2 0.4 0.6 0.8 1.0

Latency in µs

0 500 1000 1500 2000

(0) alone

(2) + Hadoop

(3) + Had. w/ QJ

0 500 1000 1500 2000

Latency in µs

0 1 2 3 4 5

Latency in µs

300 600 900 1200

Latency in µs

(0) alone

(2) + iperf

(3) + iperf w/ QJ

(0) alone

(2) + Hadoop

(3) + Had. w/ QJ

Setup | 50th% | 99th% |
--- | --- | --- |
one host, idle network | 85 | 126µs |
two hosts, shared switch | 110 | 130µs |
shared source host, shared egress port | 228 | 268µs |
shared dest. host, shared ingress port | 125 | 278µs |
shared host, shared ingress and egress | 221 | 229µs |
two hosts, shared switch queue | 1,920 | 2,100µs |
Want to know more?

<table>
<thead>
<tr>
<th>Setup</th>
<th>50th%</th>
<th>99th%</th>
</tr>
</thead>
<tbody>
<tr>
<td>one host, idle network</td>
<td>85</td>
<td>126µs</td>
</tr>
<tr>
<td>two hosts, shared switch</td>
<td>110</td>
<td>130µs</td>
</tr>
<tr>
<td>shared source host, shared egress port</td>
<td>228</td>
<td>268µs</td>
</tr>
<tr>
<td>shared dest. host, shared ingress port</td>
<td>125</td>
<td>278µs</td>
</tr>
<tr>
<td>shared host, shared ingress a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image1.png)

![Diagram](image2.png)

![Diagram](image3.png)
A

- average
- latency bound
- 99th %ile
- 100th %ile

B

End-to-end latency [μs]

C

Rate limit [Gb/s]

Throughput

Latency [μs]

Latency bound validation topology: 10 hyperservers

Latency bound validation: 60 host fan-in of 1,000

Remote Procedure Calls

Resolves Network Interference

Note the change in x-axis scale at 0.2

ECN

 Limiting is applied in reaction to ECN-marked packets.

TCP

+ Hadoop

+ Hadoop w/ QJ

+ iperf

+ iperf w/ QJ

measuring in-network latency for experiments with dag capture card and two optical taps on either side of the switch by over 300

Datacenters

Shared, multi-application environments. We consider a datacenter setup with three different applications:

1. The small difference: CDF of 9 million samples.

2. The small difference: CDF of 9 million samples.

3. The small difference: CDF of 9 million samples.

4. The small difference: CDF of 9 million samples.

5. The small difference: CDF of 9 million samples.

6. The small difference: CDF of 9 million samples.

7. The small difference: CDF of 9 million samples.

8. The small difference: CDF of 9 million samples.

9. The small difference: CDF of 9 million samples.

10. The small difference: CDF of 9 million samples.

11. The small difference: CDF of 9 million samples.
switch processing delay, represented as experiments with Figure 3: (a) CDF of packet latency across a switch.

Hadoop. We show this by repeating the memcached experiment (2–4µs) arises due to a small on-chip FIFO through the switch by over 300 of the latency-bandwidth spectrum.

Application-level latency experiments: §2.2

Remote Procedure Calls

network interference: CDF of 9 million samples.

while (RPCs) and bulk data transfers represent extreme ends.

Our experiments in §6.3 shows the latency distribution of flows' rates via special “pause” and “unpause” messages. Our 2PC system detects component failure and outperforms other readily available systems.

memcached is rate-limited: (region 0.0–1.0).

In real-world datacenter networks is an active problem. In this work, we avoid sharing hosts between applications in these experiments and share only the network infrastructure. We execute these experiments on the topology shown in Figure 5.

When their queues are nearly full, alerting senders to slow network interference experienced by Naiad [21]. This is not the same as the root mean square error (2PC) application.

In Figure 6: (c) Throughput [req/s] and 6.4. While Fastpass eliminates in-network queueing, requests are handled by hosts and switches issue special messages (DeTail) or computed from the remaining bandwidth on a first-come-first-serve basis. DeTail also addresses load imbalance caused by poor flow alignment.

Our experiments in §6.4 show the latency distribution of systems, including those we already compared against. We consider a Multi-application Environment for application-level latency.

We modify transport protocols. Hosts and switches issue special messages (DeTail) or compute bandwidth allocation from the remaining on-host interference is outside the scope of our work, but we function on commodity hardware, unmodified transport protocols.

Network congestion in datacenter networks is an active problem. In this work, a shared network, and a shared network with Hadoop. We show this by repeating the memcached experiment (2–4µs) arises due to a small on-chip FIFO through the switch by over 300 of the latency-bandwidth spectrum.

JQ fixes Naiad barrier synchronization at the highest rate limit.
Want to know more?

**Figure 3:**
- Top-left: UMP experienced by memcached sharing a network with Hadoop to Data-mining workload applications now co-exist without interference (Figure 22). Figure 2.2 shows memcached's request throughput and latency as limitations, with a scale-up emulation using a 60-host virtualized topology. We now show that our model and the synchronization offsets for memcached it is the request latency. Figure 6:

**Table 1:**
- Title: Web-search workload. Normalized FCT [log₁₀] (a) (0, 100kB): average.
- Title: Data-mining workload. Normalized FCT [log₁₀] (d) (0, 100kB): average.

**Figure 6:**
- Title: Web-search workload. Normalized FCT [log₁₀] (a) (0, 100kB): average.
- Title: Data-mining workload. Normalized FCT [log₁₀] (d) (0, 100kB): average.

**Figure 9:**
- Title: Burst size / switch buffer size [log₂]
- Title: Latency in µs
- Title: End-to-end latency [µs]
Just one more thing…

Figure 1: Motivating experiments: Hadoop traffic interferes with (a) PTPd, (b) memcached and (c) Naiad traffic.

<table>
<thead>
<tr>
<th>Setup</th>
<th>50th%</th>
<th>99th%</th>
</tr>
</thead>
<tbody>
<tr>
<td>one host, idle network</td>
<td>85</td>
<td>126μs</td>
</tr>
<tr>
<td>two hosts, shared switch</td>
<td>110</td>
<td>130μs</td>
</tr>
<tr>
<td>shared source host, shared egress port</td>
<td>228</td>
<td>268μs</td>
</tr>
<tr>
<td>shared dest. host, shared ingress port</td>
<td>125</td>
<td>278μs</td>
</tr>
<tr>
<td>shared host, shared ingress and egress</td>
<td>221</td>
<td>229μs</td>
</tr>
<tr>
<td>two hosts, shared switch queue</td>
<td>1,920</td>
<td>2,100μs</td>
</tr>
</tbody>
</table>

Table 1: Median and 99th percentile latencies observed as ping and iperf share various parts of the network.

2 Motivation

We begin by showing that shared switch queues are the primary source of network interference. We then quantify the extent to which network interference impacts application-observable metrics of performance.

2.1 Where does the latency come from?

Network interference may come from many places as the

1. Clock Synchronization

Precise clock synchronization is important to distributed systems such as Google’s Spanner [11]. PTPd offers microsecond-granularity time synchronization from a time server to machines on a local network. In Figure 1a, we show a timeline of PTPd synchronizing a host clock on both an idle network and when sharing the network with Hadoop. In the shared case, Hadoop’s shuffle phases causes queuing, which delays PTPd’s synchronization packets. This causes PTPd to temporarily fall 200–500μs out of synchronization, 50× worse than on an idle network.

2. Key-value Stores

Memcached is a popular in-memory key-value store used by Facebook and others to store small objects for quick retrieval [25]. We benchmark memcached using the memcacheLoad generator and measure the request latency. Figure 1b shows the distribution of request latencies on an idle network and a
Figure 1a (page 2) is used as a motivational experiment to show that Hadoop MapReduce is capable of interfering with the behaviour of precision time protocol. This figure is repeated in Figure 5 (page 8) in a slightly different form, combined with results from memcached combined. In this case, the figure shows that QJump is capable of resolving interference in PTPd as well as memcached.

Figure 1a
NSDI 2015 - Queues don't matter when you can Jump them!

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 1a</td>
<td>PTPd synchronization offset with and without sharing the network with Hadoop Map-Reduce</td>
</tr>
<tr>
<td>Fig. 1b</td>
<td>Memcached request latencies with and without sharing the network with Hadoop Map-Reduce</td>
</tr>
<tr>
<td>Fig. 1c</td>
<td>Naiad barrier synchronization latencies with and without sharing the network with Hadoop Map-Reduce</td>
</tr>
<tr>
<td>Tbl. 1</td>
<td>Latencies observed as ping and iperf share various parts of the network</td>
</tr>
<tr>
<td>Fig. 3a</td>
<td>Ping packet latency across a switch with and without QJump enabled</td>
</tr>
<tr>
<td>Fig. 3b</td>
<td>QJump reducing memcached request latency in the presence of Hadoop Map-Reduce traffic</td>
</tr>
<tr>
<td>Fig. 3c</td>
<td>QJump fixes Naiad barrier synchronization latency in the presence of Hadoop Map-Reduce traffic</td>
</tr>
<tr>
<td>Fig. 5</td>
<td>PTPd, memcached and Hadoop sharing a cluster, with and without QJump enabled</td>
</tr>
<tr>
<td>Fig. 6</td>
<td>QJump offers constant two phase commit throughput even at high levels of network interference</td>
</tr>
<tr>
<td>Fig. 7</td>
<td>QJump comes closest to ideal performance when compared with Ethernet Flow Control, ECN and DCTCP</td>
</tr>
<tr>
<td>Fig. 9</td>
<td>Normalized flow completion times in a 144-host simulation. QJump outperforms stand-alone TCP, DCTCP and pFabric for small flows</td>
</tr>
<tr>
<td>Fig. 10</td>
<td>Memcached throughput and latency as a function of the QJump rate limits</td>
</tr>
<tr>
<td>Fig. 11</td>
<td>Latency bound validation of QJump with 60 host generating full rate, fan in traffic</td>
</tr>
</tbody>
</table>
Just one more thing...

Guaranteed latency in datacenter networks

QJump offers a range of network service levels, from guaranteed latency for low-rate, latency-sensitive network coordination services to line-rate throughput...
QJump applies datacenter opportunities to simplify QoS rate calculations.

It provides levels of service from guaranteed latency through to line-rate throughput.

It can be deployed using without modifications to applications, kernel code or hardware.

All source data, patches and source code at [http://camsas.org/qjump](http://camsas.org/qjump)
Backup Slides
What is it good for?

Throughput [req/s] vs. Burst size / switch buffer size [log₂]

- **Broadcast UDP + QJump**
- **UDP + retries**
- **TCP**

For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes.

- **Broadcast UDP + QJump**
- **UDP + retries**
- **TCP**

Outperforms Alternatives

We discuss each result in turn. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes.

- **Broadcast UDP + QJump**
- **UDP + retries**
- **TCP**

Outperforms Alternatives

We discuss each result in turn. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes.

- **Broadcast UDP + QJump**
- **UDP + retries**
- **TCP**

Outperforms Alternatives

We discuss each result in turn. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes.

- **Broadcast UDP + QJump**
- **UDP + retries**
- **TCP**

Outperforms Alternatives

We discuss each result in turn. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes. For Hadoop, the metric of interest is the job run-time. Throughput even at high levels of network interference. Interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference mitigated by interference control than other schemes.
Accuracy of Switch Model

Number of hosts sending 0 1 2 3 4 5 6 7 8 9 10

Latency [μs]

modelled worst case
Accuracy of Switch Model

Latency [μs]

modelled worst case

Number of hosts sending
Figure 10: memcached throughput (top) and latency (bottom, log_{10}) as a function of the QJUMP rate limit.

- Region A: Throughput factor $f < 1.0$
  - $99^{th}$ percentile latency below the latency bound
  - All requests make it within the latency bound

- Region B: $1.0 < f < 2.7$
  - $99^{th}$ percentile latency above the latency bound
  - Some packet queueing, but all requests make it within the latency bound

- Region C: $f > 2.7$
  - Permanent queueing occurs
  - Latency spirals upwards until the latency bound is not met

**Related Work**

Network congestion in datacenter networks is an active research area. Table 2 compares the properties of recent systems, including those we already compared against.

- **Fastpass** [29]: employs a global arbiter that times the admission of packets into the network and routes them. While Fastpass eliminates in-network queueing, requests for allocation must queue at the centralized arbiter.
- **EyeQ** [22]: primarily aims for bandwidth partitioning, although it also reduces latency tails. It, however, requires a full-bisection bandwidth network and a kernel patch in addition to a TC module.
- **Deadline Aware TCP (D^2TCP)** [33]: extends DCTCP's window adjustment algorithm with the notion of flow deadlines, scheduling flows with earlier deadlines first. Like DCTCP, D^2TCP requires switches supporting ECN; it also requires inter-switch coordination, kernel and application modifications.
- **HULL** combines DCTCP's congestion avoidance applied on network links' utilization (rather than queue length) with a special packet-pacing NIC [2]. Its rate-limiting is applied in reaction to ECN-marked packets.
- **D^3** [35]: allocates bandwidth on a first-come-first-serve basis. It requires special switch and NIC hardware and modifies transport protocols.
- **PDQ** uses Earliest Deadline First (EDF) scheduling to prioritize straggler flows, but requires coordination across switches and application changes.
- **DeTail** [37] and pFabric [3]: pre-emptively schedule flows using packet forwarding priorities in switches. DeTail also addresses load imbalance caused by poor flow hashing. Flow priorities are explicitly specified by modified applications (DeTail) or computed from the remaining flow duration (pFabric). However, both systems require special switch hardware: pFabric uses very short queues and 64-bit priority tags, and DeTail coordinates flows' rates via special "pause" and "unpause" messages.
- **SILO** [21]: employs a similar reasoning to QJUMP to estimate expected queue lengths. It places VMs according to traffic descriptions to limit queueing and paces hosts using "null" packets.
- **TDMA Ethernet** [34]: trades bandwidth for reduced queueing by time diving network access, but requires in-

---

Figure 11: Latency bound validation: 60 host fan-in of $f_7$ and $f_0$ traffic; 100 million samples per data point. Data are sent to a single destination. Figure 11 shows the latency distribution of coordination packets as a function of the throughput factor at the highest QJUMP level, $f_7$. If the $f_7$ is set to less than 1.0 (region A), the latency bound is met (as we would expect). In region B, where $f_7$ is between 1.0 and 2.7, transient queueing affects some packets—as evident from the $100^{th}$ percentile outliers—but all requests make it within the latency bound. Beyond $f_7 = 2.7$ (region C), permanent queueing occurs.

This experiment offers two further insights about QJUMP's rate-limiting: (i) at throughput factors near 1.0, the latency bound is usually still met, and (ii) via rate-limiting, QJUMP prevents latency-sensitive applications from interfering with their own traffic.
ECN WRED Config.

Normalized RMS latency

- memcached
- PTPd
- Hadoop

ECN minimum marking threshold [segments]
<table>
<thead>
<tr>
<th>Setup</th>
<th>50\textsuperscript{th}%</th>
<th>99\textsuperscript{th}%</th>
</tr>
</thead>
<tbody>
<tr>
<td>one host, idle network</td>
<td>85</td>
<td>126μs</td>
</tr>
<tr>
<td>two hosts, shared switch</td>
<td>110</td>
<td>130μs</td>
</tr>
<tr>
<td>shared source host, shared egress port</td>
<td>228</td>
<td>268μs</td>
</tr>
<tr>
<td>shared dest. host, shared ingress port</td>
<td>125</td>
<td>278μs</td>
</tr>
<tr>
<td>shared host, shared ingress and egress</td>
<td>221</td>
<td>229μs</td>
</tr>
<tr>
<td>two hosts, shared switch queue</td>
<td>1,920</td>
<td>2,100μs</td>
</tr>
</tbody>
</table>
application setting, we avoid sharing hosts between applications in these experiments and share only the network infrastructure. We consider a multi-application environment, a range of applications with different latency and bandwidth requirements, sharing the same infrastructure.

Multi-application Environment

Remote Procedure Calls

Figure 2. Barrier synchronization offsets for Naiad. On an idle network, 90% of synchronizations take no more than 4µs for each of our switches. This verifies that queueing latency at switches which the switch must process packets in-order. The average synchronization offset is 34µs, and the maximum is 60µs.

Naiad Barrier Synchronization

Figure 3a shows the latency distribution of 9 million samples. As in Figure 2, we reduce its packets' latency at the network interface. (c) shows the timeline of average request latency: (top) alone, (middle) + iperf, (bottom) + iperf w/ QJ enabled. With QJ enabled, the request latency decreases from 824µs in the shared case to 476µs, a nearly 2× improvement.

Figure 3b shows a timeline of average request latency: (top) alone, (middle) + iperf, (bottom) + iperf w/ QJ enabled, the request latency decreases from 824µs in the shared case to 476µs, a nearly 2× improvement.

We execute these experiments on the topology shown in Figure 3. The small difference in latency between idle switch latency (1.6µs) and latency due to a small on-chip FIFO throughput is due to a small on-chip FIFO throughput. The latency due to a small on-chip FIFO throughput is due to the switch processing delay, represented as \( s \) in Equation 2.

\[ \text{Latency} = t + s \]

\( t \) is the latency within the switch, \( s \) is the switch processing delay, represented as \( s \) in Equation 2. The latency due to a small on-chip FIFO throughput is due to the switch processing delay, represented as \( s \) in Equation 2.

Figure 4 shows the distribution (CDF) of memcached request latencies when running on an idle network network, 90% of synchronizations take no more than 4µs for each of our switches. In this experiment, memcached is configured at an intermediate level, rate-limited, and has a latency of 4µs for each of our switches.

In this experiment, memcached is configured at an intermediate level, rate-limited, and has a latency of 4µs for each of our switches.

Figure 5 resolves network interference. (b) mitigates the latency tails from Figure 5a, we reduce its packets' latency at the network interface. (c) shows the two latency phases) emerge. With QJ enabled, the request latency decreases from 824µs in the shared case to 476µs, a nearly 2× improvement.
How well does it work?

memcached key-value store vs Hadoop

![Graph showing latency distribution for memcached and Hadoop](image)
Naiad data processing framework vs Hadoop

How well does it work?

The diagram compares the latency performance of Naiad data processing framework against Hadoop. It shows the cumulative distribution function (CDF) of request latencies across different configurations:

- Blue line: Naiad alone
- Red line: Naiad + Hadoop
- Green dotted line: Naiad + Hadoop with QJ

The x-axis represents latency in microseconds (µs), while the y-axis represents the probability of events happening within a certain latency range. The graph illustrates that adding Hadoop or QJ to Naiad significantly improves latency performance compared to running Naiad alone.
How well does it work?

![Graph](image)

- **idle network**: memcached avg. latency, PTPd offset
- **+ Hadoop**: memcached avg. latency, PTPd offset
- **+ Hadoop, with QJump**: memcached avg. latency, PTPd offset

<table>
<thead>
<tr>
<th>Time [µs]</th>
<th>1200</th>
<th>800</th>
<th>400</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>200</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>250</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>300</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>350</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**Time since start [sec]**

Distributed Atomic Commit

- Bounded latency level, we built a simple distributed two-phase atomic-commit (2PC) application.
- This optimization yields a significant improvement over both TCP and UDP.
- Since interference is transient in these experiments, we showed the request rate for one coordinator and seven servers as a function of network interference.
- All cases are normalized to the ideal case, which offers reliable delivery, the coordinator can send its messages by UDP broadcast when it generates a constant 10Gb/s of UDP traffic and offers constant two-phase commit with explicit acknowledgements and retransmissions.
- The application communicates over TCP or over UDP, depending on the availability of acknowledgements.
- We discuss each result in turn.

- Hadoop's performance is not noticeably affected by interference,
- Interference control than other schemes.
- Server interference ratios (RMSE), which measures prediction accuracy.

**Figure 5**: Time [µs]

- Fig. 5 (bottom) shows the request rate for one coordinator and seven servers as a function of network interference.
- Fig. 5 (middle) shows the request rate for one coordinator and seven servers as a function of network interference.
- Fig. 5 (top) shows the request rate for one coordinator and seven servers as a function of network interference.

**Figure 6**: Time since start [sec]

- Shows interference ratios (RMSE), which measures prediction accuracy.
- Server interference mitigation by QJump.
- The network interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference.
- The request rate for one coordinator and seven servers as a function of network interference.
- Shows the request rate for one coordinator and seven servers as a function of network interference.
- Shows the request rate for one coordinator and seven servers as a function of network interference.

**Conclusion**: The network interference is transient in these experiments, we show the request rate for one coordinator and seven servers as a function of network interference.
- The request rate for one coordinator and seven servers as a function of network interference.
- Shows the request rate for one coordinator and seven servers as a function of network interference.
- Shows the request rate for one coordinator and seven servers as a function of network interference.
Flow Completion Times

Web-search workload

Normalized FCT $[\log_{10}]$

Load

(a) (0, 100kB]: average.

(b) (0, 100kB]: 99th percentile.

(c) (10MB, $\infty$): average.

Data-mining workload

Normalized FCT $[\log_{10}]$

Load

(d) (0, 100kB]: average.

(e) (0, 100kB]: 99th percentile.

(f) (10MB, $\infty$): average.
How to calculate $f$

![Graph](image)

- Latency bound validation topology: 10 hypervisors (HV) and 60 guests (G1..60) and 120 apps.
- SILO advocates for allocation of bandwidth on a first-come-first-serve basis. It requires special switch and NIC hardware and employs a similar reasoning to SILO.
- EyeQ primarily aims for bandwidth partitioning, trades bandwidth for reduced latency; 100 million samples per data point.
- Rate limit ($f$)
- Max. latency
- 99%ile latency