

Fastpass: A Centralized “Zero-Queue” Datacenter Network

Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, Hans Fugal
M.I.T. Computer Science & Artificial Intelligence Lab Facebook

<http://fastpass.mit.edu/>

ABSTRACT

An ideal datacenter network should provide several properties, including low median and tail latency, high utilization (throughput), fair allocation of network resources between users or applications, deadline-aware scheduling, and congestion (loss) avoidance. Current datacenter networks inherit the principles that went into the design of the Internet, where packet transmission and path selection decisions are distributed among the endpoints and routers. Instead, we propose that each sender should delegate control—to a centralized arbiter—of when each packet should be transmitted and what path it should follow.

This paper describes Fastpass, a datacenter network architecture built using this principle. Fastpass incorporates two fast algorithms: the first determines the time at which each packet should be transmitted, while the second determines the path to use for that packet. In addition, Fastpass uses an efficient protocol between the endpoints and the arbiter and an arbiter replication strategy for fault-tolerant failover. We deployed and evaluated Fastpass in a portion of Facebook’s datacenter network. Our results show that Fastpass achieves high throughput comparable to current networks at a $240\times$ reduction in queue lengths (4.35 Mbytes reducing to 18 Kbytes), achieves much fairer and consistent flow throughputs than the baseline TCP ($5200\times$ reduction in the standard deviation of per-flow throughput with five concurrent connections), scalability from 1 to 8 cores in the arbiter implementation with the ability to schedule 2.21 Terabits/s of traffic in software on eight cores, and a $2.5\times$ reduction in the number of TCP retransmissions in a latency-sensitive service at Facebook.

1. INTRODUCTION

Is it possible to design a network in which: (1) packets experience *no queueing delays* in the network, (2) the network achieves *high utilization*, and (3) the network is able to support *multiple resource allocation objectives* between flows, applications, or users?

Such a network would be useful in many contexts, but especially in datacenters where queueing dominates end-to-end latencies, link rates are at technology’s bleeding edge, and system operators have to contend with multiple users and a rich mix of workloads. Meeting complex service-level objectives and application-specific goals would be much easier in a network that delivered these three ideals.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM’14, August 17–22, 2014, Chicago, IL, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2836-4/14/08...\$15.00.

<http://dx.doi.org/10.1145/2619239.2626309>.

Current network architectures distribute packet transmission decisions among the endpoints (“congestion control”) and path selection among the network’s switches (“routing”). The result is strong fault-tolerance and scalability, but at the cost of a loss of control over packet delays and paths taken. Achieving high throughput requires the network to accommodate bursts of packets, which entails the use of queues to absorb these bursts, leading to delays that rise and fall. Mean delays may be low when the load is modest, but tail (e.g., 99th percentile) delays are rarely low.

Instead, we advocate what may seem like a rather extreme approach to exercise (very) tight control over when endpoints can send packets and what paths packets take. We propose that *each* packet’s timing be controlled by a logically centralized *arbiter*, which also determines the packet’s path (Fig. 1). If this idea works, then flow rates can match available network capacity over the time-scale of individual packet transmission times, unlike over multiple round-trip times (RTTs) with distributed congestion control. Not only will persistent congestion be eliminated, but packet latencies will not rise and fall, queues will never vary in size, tail latencies will remain small, and packets will never be dropped due to buffer overflow.

This paper describes the design, implementation, and evaluation of *Fastpass*, a system that shows how centralized arbitration of the network’s usage allows endpoints to burst at wire-speed while eliminating congestion at switches. This approach also provides *latency isolation*: interactive, time-critical flows don’t have to suffer queueing delays caused by bulk flows in other parts of the fabric. The idea we pursue is analogous to a hypothetical road traffic control system in which a central entity tells every vehicle when to depart and which path to take. Then, instead of waiting in traffic, cars can zoom by all the way to their destinations.

Fastpass includes three key components:

1. A fast and scalable **timeslot allocation algorithm** at the arbiter to determine when each endpoint’s packets should be sent (§3). This algorithm uses a fast maximal matching to achieve objectives such as max-min fairness or to approximate minimizing flow completion times.
2. A fast and scalable **path assignment algorithm** at the arbiter to assign a path to each packet (§4). This algorithm uses a fast edge-coloring algorithm over a bipartite graph induced by switches in the network, with two switches connected by an edge if they have a packet to be sent between them in a timeslot.
3. A **replication strategy for the central arbiter** to handle network and arbiter failures, as well as a reliable control protocol between endpoints and the arbiter (§5).

We have implemented Fastpass in the Linux kernel using high-precision timers (hrtimers) to time transmitted packets; we achieve sub-microsecond network-wide time synchronization using the IEEE1588 Precision Time Protocol (PTP).

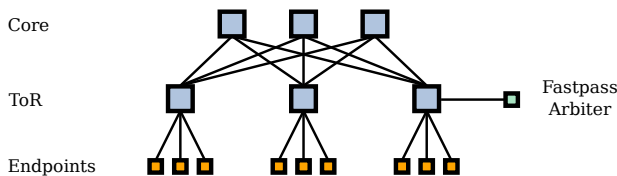


Figure 1: Fastpass arbiter in a two-tier network topology.

We conducted several experiments with Fastpass running in a portion of Facebook’s datacenter network. Our main findings are:

1. High throughput with nearly-zero queues: On a multi-machine bulk transfer workload, Fastpass achieved throughput only 1.6% lower than baseline TCP, while **reducing the switch queue size from a median of 4.35 Mbytes to under 18 Kbytes**. The resulting RTT reduced from 3.56 ms to 230 μ s.
2. Consistent (fair) throughput allocation and fast convergence: In a controlled experiment with multiple concurrent flows starting and ending at different times, Fastpass **reduced the standard deviation of per-flow throughput by a factor over 5200 \times** compared to the baseline with five concurrent TCP connections.
3. Scalability: Our implementation of the arbiter shows nearly linear scaling of the allocation algorithm from one to eight cores, with the **8-core allocator handling 2.21 Terabits/s**. The arbiter responds to requests within tens of microseconds even at high load.
4. Fine-grained timing: The implementation is able to synchronize time accurately to within a few hundred nanoseconds across a multi-hop network, sufficient for our purposes because a single 1500-byte MTU-sized packet at 10 Gbits/s has a transmission time of 1230 nanoseconds.
5. Reduced retransmissions: On a real-world latency-sensitive service located on the response path for user requests, **Fastpass reduced the occurrence of TCP retransmissions by 2.5 \times** , from between 4 and 5 per second to between 1 and 2 per second.

These experimental results indicate that Fastpass is viable, providing a solution to several specific problems observed in datacenter networks. First, reducing the tail of the packet delay distribution, which is important because many datacenter applications launch hundreds or even thousands of request-response interactions to fulfill a single application transaction. Because the longest interaction can be a major part of the transaction’s total response time, reducing the 99.9th or 99.99th percentile of latency experienced by packets would reduce application latency.

Second, avoiding false congestion: packets may get queued behind other packets headed for a bottleneck link, delaying non-congested traffic. Fastpass does not incur this delay penalty.

Third, eliminating incast, which occurs when concurrent requests to many servers triggers concurrent responses. With small router queues, response packets are lost, triggering delay-inducing retransmissions [26], whereas large queues cause delays to other bulk traffic. Current solutions are approximations of full control, based on estimates and assumptions about request RTTs, and solve the problem only partially [28, 37].

And last but not least, better sharing for heterogeneous workloads with different performance objectives. Some applications care about low latency, some want high bulk throughput, and some want to minimize job completion time. Supporting these different objectives within the same network infrastructure is challenging using distributed congestion control, even with router support. By contrast,

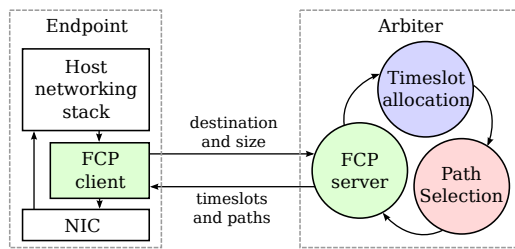


Figure 2: Structure of the arbiter, showing the timeslot allocator, path selector, and the client-arbiter communication.

a central arbiter can compute the timeslots and paths in the network to jointly achieve these different goals.

2. FASTPASS ARCHITECTURE

In Fastpass, a logically centralized arbiter controls all network transfers (Fig. 2). When an application calls `send()` or `sendto()` on a socket, the operating system sends this *demand* in a request message to the Fastpass arbiter, specifying the destination and the number of bytes. The arbiter processes each request, performing two functions:

1. **Timeslot allocation:** Assign the requester a set of *timeslots* in which to transmit this data. The granularity of a timeslot is the time taken to transmit a single MTU-sized packet over the fastest link connecting an endpoint to the network. The arbiter keeps track of the source-destination pairs assigned each timeslot (§3).
2. **Path selection.** The arbiter also chooses a path through the network for each packet and communicates this information to the requesting source (§4).

Because the arbiter knows about all current and scheduled transfers, it can choose timeslots and paths that yield the “*zero-queue*” property: the arbiter arranges for each packet to arrive at a switch on the path just as the next link to the destination becomes available.

The arbiter must achieve high throughput and low latency for both these functions; a single arbiter must be able to allocate traffic for a network with thousands of endpoints within a few timeslots.

Endpoints communicate with the arbiter using the Fastpass Control Protocol (FCP) (§5.3). FCP is a reliable protocol that conveys the demands of a sending endpoint to the arbiter and the allocated timeslot and paths back to the sender. FCP must balance conflicting requirements: it must consume only a small fraction of network bandwidth, achieve low latency, and handle packet drops and arbiter failure without interrupting endpoint communication. FCP provides reliability using timeouts and ACKs of aggregate demands and allocations. Endpoints aggregate allocation demands over a few microseconds into each request packet sent to the arbiter. This aggregation reduces the overhead of requests, and limits queuing at the arbiter.

Fastpass can recover from faults with little disruption to the network (§5). Because switch buffer occupancy is small, packet loss is rare and can be used as an indication of component failure. Endpoints report packet losses to the arbiter, which uses these reports to isolate faulty links or switches and compute fault-free paths. The arbiter itself maintains only soft state, so that a secondary arbiter can take over within a few milliseconds if the primary arbiter fails.

To achieve the ideal of zero queuing, Fastpass requires precise timing across the endpoints and switches in the network (§6.3). When endpoint transmissions occur outside their allocated timeslots, packets from multiple allocated timeslots might arrive at a switch at the same time, resulting in queueing. Switch queues allow the network to tolerate timing inaccuracies: worst-case queueing is no

larger than the largest discrepancy between clocks, up to 1.5 Kbytes for every 1.2 μ s of clock divergence at 10 Gbits/s.

Fastpass requires no switch modifications, nor the use of any advanced switch features. Endpoints require some hardware support in NICs that is currently available in commodity hardware (§6.3), with protocol support in the operating system. Arbiters can be ordinary server-class machines, but to handle very large clusters, a number of high-speed ports would be required.

2.1 Latency experienced by packets in Fastpass

In an ideal version of Fastpass, endpoints receive allocations as soon as they request them: the latency of communication with the arbiter and the time to compute timeslots and paths would be zero. In this ideal case, the end-to-end latency of a packet transmission would be the time until the allocated timeslot plus the time needed for the packet to traverse the path to the receiver with empty queues at all egress ports.

In moderately-loaded to heavily-loaded networks, ideal allocations will typically be several timeslots in the future. As long as the Fastpass arbiter returns results in less than these several timeslots, Fastpass would achieve the ideal minimum packet latency in practice too.

In lightly-loaded networks, Fastpass trades off a slight degradation in the mean packet latency (due to communication with the arbiter) for a significant reduction in the tail packet latency.

2.2 Deploying Fastpass

Fastpass is deployable incrementally in a datacenter network. Communication to endpoints outside the Fastpass boundary (e.g., to hosts in a non-Fastpass subnet or on the Internet) uses Fastpass to reach the boundary, and is then carried by the external network. Incoming traffic either passes through gateways or travels in a lower priority class. Gateways receive packets from outside the boundary, and use Fastpass to send them within the boundary. Alternatively, incoming packets may use a lower-priority class to avoid inflating network queues for Fastpass traffic.

This paper focuses on deployments where a single arbiter is responsible for all traffic within the deployment boundary. We discuss larger deployments in §8.1.

3. TIMESLOT ALLOCATION

The goal of the arbiter’s timeslot allocation algorithm is to choose a *matching* of endpoints in each timeslot, i.e., a set of sender-receiver endpoint pairs that can communicate in a timeslot. For a simpler exposition, we assume here that all endpoint links run at the same rate. The demand for any given link in the network can exceed its capacity; the arbiter selects sender-receiver pairs and assigns a path to packets (described in §4) to ensure that traffic issued in a given timeslot will not exceed any link’s bandwidth.

Networks are often organized into tiers, with each tier providing network transport to components below it: top-of-rack switches connect servers in a rack, aggregation switches connect racks into clusters, core routers connect clusters. Fastpass requires that tiers be *rearrangeably non blocking* (RNB) [14], networks where any traffic that satisfies the input and output bandwidth constraints of the network can be routed such that no queuing occurs.

The RNB property allows the arbiter to perform timeslot allocation separately from path selection: as long as the allocated matching satisfies the bandwidth constraints in and out of each tier, path selection is guaranteed to successfully assign paths on the physical topology. Consequently, each tier can be abstracted as a single

switch for the purpose of timeslot allocation.¹ The result is a tree topology on which it is easy for timeslot allocation to check bandwidth constraints, even when the physical network is oversubscribed and has many paths between endpoints. Non-oversubscribed (full-bisection bandwidth) topologies [4, 18, 27, 38] can be abstracted further: we can view the entire network as a single switch.

Because the arbiter has knowledge of all endpoint demands, it can allocate traffic according to global policies that would be harder to enforce in a distributed setting. For instance, the arbiter can allocate timeslots to achieve max-min fairness, to minimize flow completion time, or to limit the aggregate throughput of certain classes of traffic. When conditions change, the network does not need to converge to a good allocation – the arbiter can change the allocation from one timeslot to the next. As a result, the policy (e.g., fairness) can be achieved even over short time scales.

How fast must a viable allocation algorithm be? At first glance, endpoint link speeds determine the allowed allocator runtime, since the arbiter’s processing rate must match endpoint link speed. This is about one timeslot per microsecond for 10 Gbits/s links with 1500-byte timeslots. However, parallelism can enable longer runtimes: if the allocation of multiple timeslots can be run in parallel, allocation of each timeslot can take longer while still maintaining the arbiter’s processing rate.

A long runtime (compared to the minimum RTT between the endpoints) is acceptable with some workloads, but not others. On heavily-loaded endpoints, the time until the first available timeslot can be tens to hundreds of microseconds, so traffic will observe the ideal end-to-end latency (§2.1), even if allocation takes many microseconds. On the other hand, traffic on lightly-loaded networks doesn’t enjoy this masking of allocation latency; the algorithm must finish promptly if a small end-to-end latency increase is desired.

Complete knowledge of all network demands thus becomes a double-edged sword; in order to meet these latency and throughput requirements, the timeslot allocator requires very fast algorithms.

Finding an allocation with the largest possible number of pairs (a *maximum matching*) is expensive; switch algorithms (e.g., [34, 9, 24]) generally use heuristics to find good, but not maximum, matchings. Fastpass uses a similar approach: as the arbiter processes demands, it greedily allocates a source-destination pair if allocating the pair does not violate bandwidth constraints.² When the arbiter finishes processing all demands, it has a *maximal matching*, a matching in which none of the unallocated demands can be allocated while maintaining the bandwidth constraints.

3.1 A pipelined allocator

The allocator takes a list of all network demands (how many timeslots are waiting to be sent between each pair of endpoints), and computes the allocated matching and the remaining demands after the allocation. Figure 3 shows how Fastpass allocators are arranged into a pipeline: the input to the allocator processing timeslot t is the remaining demand after allocating timeslot $t - 1$.

The arbiter implements different allocation policies by changing the order in which demands are processed. For max-min fairness, the arbiter orders demands by the last timeslot that was allocated to the source-destination pair, “least recently allocated first”; for minimizing flow completion time (min-FCT), the arbiter tracks the

¹A switch where port capacity reflects the aggregate bandwidth in and out of the tier to that component.

²In a non-oversubscribed network, the arbiter checks that neither the source nor the destination have already been allocated to a different pair. Oversubscribed topologies require the arbiter to additionally check bandwidth constraints in and out of each network tier.

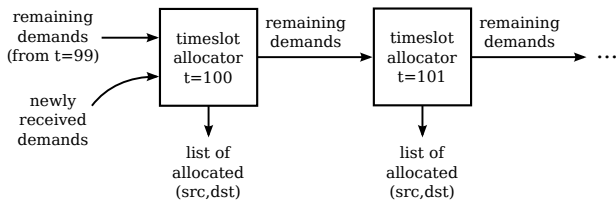


Figure 3: Pipelined timeslot allocation. The allocator for timeslot t processes the demands not fully satisfied by the allocator for $t - 1$

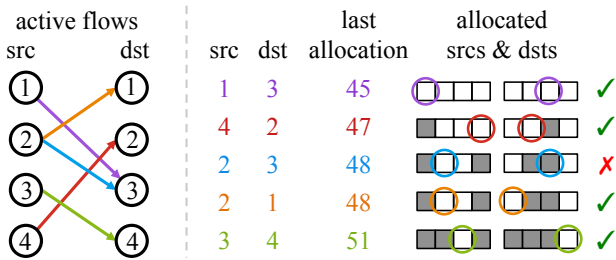


Figure 4: Timeslot allocation for the max-min fairness allocation objective.

number of pending bytes (measured in MTU-sized chunks) and performs "fewest remaining MTUs first".

Figure 4 demonstrates the allocation of one timeslot in a simple network with four endpoints. The allocator orders the demands by the last timeslot allocated to each pair, and processes them in that order. On the right is the state used to track bandwidth constraints: one bit for each source and for each destination. The first two demands can be allocated because both the source and destination are available, but the third demand cannot be allocated because destination 3 has already been allocated. The remaining two demands can be allocated, yielding a maximal matching.

Each allocator in the pipeline receives a stream of demands. Ideally, an allocator could process each demand as soon as it is produced by the previous allocator. If demands can appear out of the desired order, however, the allocator must reorder them first. In a worst-case scenario, the last demand from the previous allocator should be processed first. The allocator would have to wait until the previous allocator produced all demands in the stream before it could start processing, eliminating the concurrency gained by pipelining.

Fortunately, with both max-min fairness and min-FCT (and other such objectives), demands can be kept in roughly the correct order with only limited reordering. For example, in max-min fairness, the allocator of timeslot t only changes the last allocated timeslot of a source-destination pair if that pair is allocated, and will only change it to t . Therefore, an allocator for timeslot $t + 1$ can process all demands with last allocated timeslot strictly less than t immediately. Only demands with last allocated timeslot equal to t need to be kept until all demands are received.

To reduce the overhead of processing demands, the allocator allocates a batch of 8 timeslots in one shot using a fast data structure, the *bitmap table*. This table maintains a bitmap for each sender-receiver pair in the network, with one bit per timeslot. A "1" in the bitmap signifies that the pair is not scheduled to communicate in that timeslot, while a "0" indicates otherwise. To find the first available timeslot for a given packet, the allocator computes the bitwise AND of the source and destination bitmaps, and then uses the "find first set" operation (the `bsf` instruction on x86). Modern processors perform this operation fast [2]. Pairs that have been allocated and have remaining demand are kept, and the arbiter will attempt to

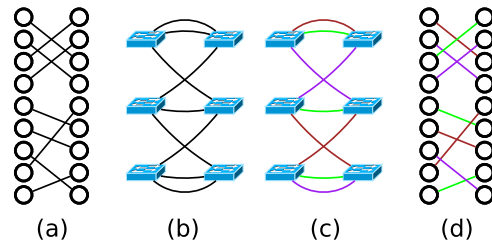


Figure 5: Path selection. (a) input matching (b) ToR graph (c) edge-colored ToR graph (d) edge-colored matching.

allocate more batch timeslots to them; pairs that cannot be allocated in this batch are sent to the next allocator in the pipeline.

3.2 Theoretical guarantees

We prove in the appendix that in those topologies where timeslot allocation and path selection can be separated, the average latency with Fastpass is no worse than $2 \times$ the average latency of an optimal scheduler with half as much network capacity on its worst-case workload. The result upper-bounds the latency cost of using Fastpass over any other solution. The bound, however, is not tight: Fastpass achieves low latency even at much higher network loads (§7).

4. PATH SELECTION

Path selection assigns packets that have been allocated timeslots to paths through the network that avoid queueing. Common datacenter topologies (e.g., multi-rooted trees) include redundant paths between endpoints. If the timeslot allocator admits traffic that utilizes the full network bandwidth, and more packets attempt to traverse a link than that link can sustain, queueing will result. To utilize the full network bandwidth without queueing, path selection must balance packet load across all available links.

Existing approaches for load balancing all have significant disadvantages. Equal-cost multi-path (ECMP, RFC 2992) routing can result in multiple flows hashing onto the same path, causing significant skew over short time scales. Hedera [5] is able to re-route "elephant" flows for better load balance, but focuses only on such flows; the load generated by small flows at finer scales is left unbalanced, and that could be substantial in practice.

The goal of path selection is to assign packets to paths such that no link is assigned multiple packets in a single timeslot; this property guarantees that there will be no queueing within the network. Timeslot allocation guarantees that this property holds for the links directly connected to each endpoint; path selection must provide this guarantee for the remainder of the network. In a network with two tiers, ToR and core, with each ToR connected directly to a subset of core switches (Fig. 1) each path between two ToRs can be uniquely specified by a core switch. Thus path selection entails assigning a core switch to each packet such that no two packets (all of which are to be sent in the same timeslot) with the same source ToR or destination ToR are assigned the same core switch.

Edge-coloring. This assignment can be performed with *graph edge-coloring* [21]. The edge-coloring algorithm takes as input a bipartite graph and assigns a color to each edge such that no two edges of the same color are incident on the same vertex. We model the network as a bipartite graph where the vertices are ToR switches, edges are the allocated packets, and colors represent core switches/paths. The edge-coloring of this graph provides an assignment of packets to paths such that no link is assigned multiple packets.

Figure 5 shows an example. The matching of packets to be transmitted in a given timeslot (a) is transformed into a bipartite multi-

graph of ToRs (b), where the source and destination ToRs of every packet are connected. Edge-coloring colors each edge ensuring that no two edges of the same color are incident on the same ToR (c). The assignment guarantees that at most one packet occupies the ingress, and one occupies the egress, of each port (d).

Edge-coloring requires uniform link capacities; in networks with heterogeneous link capacities, we can construct a virtual network with homogeneous link capacities on which to assign paths. Here, we replace each physical switch with high-capacity links with multiple switches with low capacity links that connect to the same components as the physical switch (e.g., one switch with 40 Gbits/s links would be replaced by four switches with 10 Gbits/s links). All packets assigned a path through the duplicate switches in the virtual topology would be routed through the single high-capacity switch on the physical topology.

Edge-coloring also generalizes to oversubscribed networks and networks with multiple tiers. Only traffic that passes through a higher network tier is edge-colored (e.g., in a two-tier network, only inter-rack traffic requires path selection). For a three-tier datacenter with ToR, Agg, and Core switches (and higher-tier ones), paths can be assigned hierarchically: the edge-coloring of the ToR graph assigns Agg switches to packets, then an edge-coloring of the Agg graph chooses Core switches [21, §IV].

Fast edge-coloring. A network with n racks and d nodes per rack can be edge-colored in $O(nd \log d)$ time [12, 23]. Fast edge-coloring algorithms invariably use a simple and powerful building block, the *Euler-split*. An Euler-split partitions the edges of a regular graph where each node has the same degree, $2d$, into two regular graphs of degree d . The algorithm is simple: (1) find an Eulerian cycle (a cycle that starts and ends at the same node, and contains every edge exactly once, though nodes may repeat) of the original graph, (2) assign alternate edges of the cycle to the two new graphs, (3) repeat.

An Euler-split divides the edges into two groups that can be colored separately. $d - 1$ Euler-splits can edge-color a graph with power-of-two degree d by partitioning it into d perfect matchings, which can each be assigned a different color. Graphs with non-power-of-two degree can be edge-colored using a similar method that incorporates one search for a perfect matching, and has only slightly worse asymptotic complexity [23].

The Fastpass path selection implementation maintains the bipartite graph in a size-optimized bitmap-based data structure that can fit entirely in a 32 KB L1 cache for graphs with up to 6,000 nodes. This data structure makes the graph walks needed for Euler-split fast, and yields sufficiently low latencies for clusters with a few thousand nodes (§7.3).

5. HANDLING FAULTS

Three classes of faults can render a Fastpass network ineffective: failures of in-network components (nodes, switches and links), failures of the Fastpass arbiter, and packet losses on the communication path between endpoints and the arbiter.

5.1 Arbiter failures

Fastpass runs multiple arbiters, one primary and a few secondaries. The arbiters all subscribe to a pre-determined IP multicast destination address to which all requests from endpoints are sent (responses are unicast to the endpoint). All the arbiters receive all requests (modulo packet loss), but only the designated primary responds to all requests; the secondaries drop the requests and generate no allocations.

The secondaries detect the failure of the primary as follows. The primary sends out tiny watchdog packets on the multicast group every T_w microseconds. If a secondary receives no watchdog packets during an interval of time T_d , that secondary assumes that the primary

has failed, and starts responding to client requests. If there is more than one secondary, a pre-defined rank order determines the order in which secondaries attempt to take over as the primary. Our current implementation uses only one secondary.

In practice, arbiters can be aggressive about detecting and reacting to failure, allowing recovery within 1–2 ms. An implementation can use $T_w = 100$ microseconds and $T_d = 1$ millisecond to achieve fast failover, consuming under 10 Mbits/s.

Upon an arbiter failover, the new arbiter must obtain an updated snapshot of all endpoint demands, so it can start allocating timeslots. The Fastpass Control Protocol (described below) assumes only soft-state at the arbiter, allowing endpoints to re-synchronize with the new arbiter in 1–2 round-trip times.

The transition period between old and new arbiters needs careful handling to prevent persistent queues from building up. For example, if the old arbiter tells A to send to C at some timeslot, and the new arbiter tells B to send to C at the same time, a queue will form at C 's ingress until encountering a free timeslot.

Conservative timing can ensure that by the time an arbiter failure is detected and the secondary takes over, no allocations made by the failed arbiter remain in the network. In our implementation, the arbiter allocates each timeslot $65 \mu\text{s}$ before the timeslot's start time to leave enough time for notifications to reach the endpoints; this number is much smaller than the 1 millisecond before the secondary takes over.

As a result, the old and new arbiters do not have to share information to make the failover possible, simplifying the implementation: the new arbiter just resynchronizes with endpoints and starts allocating.

5.2 Network failures

Arbiters need to know the network topology under their control, so packets can successfully traverse the network in their allotted timeslots. If links or switches become unavailable undetected, the arbiter would continue to allocate traffic through them, causing packet loss.

Because Fastpass maintains low queue occupancies, it can use packet drops to detect network faults. Sporadic packet losses due to bit flips in the physical layer still occur, but persistent, correlated packet loss is almost surely due to component failure. The path each packet traverses is known to the endpoints, which helps localize the fault.

A network service performs fault isolation, separate from the arbiter. Endpoints report packet drops to this fault isolation service, which correlates reports from multiple endpoints to identify the malfunctioning component. The faulty component is then reported to the arbiter, which avoids scheduling traffic through the component until the fault is cleared. Since packet errors can be detected quickly, failure detection and mitigation times can be made extremely short, on the order of milliseconds.

5.3 Fastpass Control Protocol (FCP)

Communication between endpoints and the arbiter is not scheduled and can experience packet loss. FCP must protect against such loss. Otherwise, if an endpoint request or the arbiter's response is dropped, a corresponding timeslot would never be allocated, and some packets would remain stuck in the endpoint's queue.

TCP-style cumulative ACKs and retransmissions are not ideal for FCP. At the time of retransmission, the old packet is out of date: for a request, the queue in the endpoint might be fuller than it was; for an allocation, an allocated timeslot might have already passed.

FCP provides reliability by communicating aggregate counts; to inform the arbiter of timeslots that need scheduling, the endpoint sends the sum total of timeslots it has requested so far for that

destination. The arbiter keeps a record of each endpoints’ latest demands; the difference from the kept record and the new aggregate demands specifies the amount of new timeslots to be allocated.

The counts are idempotent: receiving the same count multiple times does not cause timeslots to be allocated multiple times. Idempotency permits aggressive timeouts, leading to low allocation latency even in the face of occasional packet loss. Endpoints detect the loss of allocated timeslots using aggregate counts sent by the arbiter, triggering a request for more timeslots.

Handling arbiter failure. When a secondary arbiter replaces a failed primary, its aggregate counts are out of sync with the endpoints. The mismatch is detected using a random nonce established when the arbiter and endpoint synchronize. The nonce is incorporated into the packet checksum, so when an arbiter is out of sync with an endpoint, it observes a burst of failed checksums from an endpoint that triggers a re-synchronization with that endpoint.

Upon re-synchronization, the arbiter resets its aggregate counts to zeros, and the endpoint recomputes fresh aggregate counts based on its current queues. The process takes one round-trip time: as soon as the endpoint processes the RESET packet from the arbiter, it can successfully issue new requests to the arbiter.

6. IMPLEMENTATION

We implemented the arbiter using Intel DPDK [1], a framework that allows direct access to NIC queues from user space. On the endpoints, we implemented a Linux kernel module that queues packets while requests are sent to the arbiter. Our source code is at <http://fastpass.mit.edu/>

6.1 Client

FCP is implemented as a Linux transport protocol (over IP). A Fastpass *qdisc* (queueing discipline) queues outgoing packets before sending them to the NIC driver, and uses an FCP socket to send demands to and receive allocations from the arbiter.

The Fastpass *qdisc* intercepts each packet just before it is passed to the NIC, and extracts the destination address from its packet header.³ It does not process transport protocol headers (e.g., TCP or UDP).

The Fastpass arbiter schedules network resources, obviating the need for TCP congestion control. TCP’s congestion window (*cwnd*) could needlessly limit flow throughput, but packet loss is rare, so *cwnd* keeps increasing until it no longer limits throughput. At this point, TCP congestion control is effectively turned off.

The current implementation does not modify TCP. The evaluated system maintains long-lived flows, so flows are not *cwnd*-limited, and we preferred not to deal with the complexities of TCP. Nevertheless, modifying TCP’s congestion control would be worthwhile for improving short-lived TCP flow performance.

The client is not limited to sending exactly one packet per timeslot. Instead, it greedily sends packets while their total transmission time is less than the timeslot length. This aggregation reduces the amount of potentially wasted network bandwidth caused by many small packets destined to the same endpoint.

6.2 Multicore Arbiter

The arbiter is made up of three types of cores: *comm-cores* communicate with endpoints, *alloc-cores* perform timeslot allocation, and *pathsel-cores* assign paths.

The number of cores of each type can be increased to handle large workloads: each *comm-core* handles a subset of endpoints, so endpoints can be divided among more cores when protocol handling becomes a bottleneck; *alloc-cores* work concurrently using pipeline

³On a switched network, MAC addresses could be used. However, in the presence of routing, IP addresses are required.

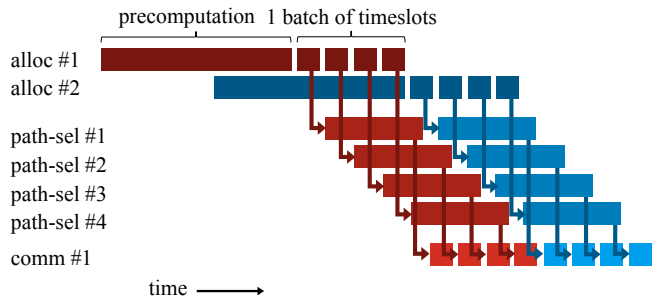


Figure 6: Multicore allocation: (1) allocation cores assign packets to timeslots, (2) path selection cores assign paths, and (3) communication cores send allocations to endpoints.

parallelism (§3); and *pathsel-cores* are “embarrassingly parallel”, since path assignments for different timeslots are independent.

Fig. 6 shows communication between arbiter cores. *Comm-cores* receive endpoint demands and pass them to *alloc-cores* (not shown). Once a timeslot is completely allocated, it is promptly passed to a *pathsel-core*. The assigned paths are handed to *comm-cores*, which notify each endpoint of its allocations.

Performance measurements of each core type are presented in §7.

6.3 Timing

To keep queue occupancy low, end-node transmissions should occur at the times prescribed by the arbiter. Otherwise, packets from multiple endpoints might arrive at a switch’s egress port together, resulting in queueing.

The amount of queueing caused by time-jitter is determined by the discrepancy in clocks. For example, if all clocks are either accurate or one timeslot fast, at most two packets will arrive at any egress: one from an accurate node, the other from a fast node.

Clock synchronization. The deployment synchronizes end-node time using the IEEE1588 Precision Time Protocol (PTP), which achieves sub-microsecond clock synchronization by carefully mitigating causes of time-synchronization jitter. PTP-capable NICs timestamp synchronization packets in hardware [29], thus avoiding jitter due to operating system scheduling. NICs with PTP support are widely available; the experiment used Intel 82599EB NICs.

Variable queueing delays inside the network also cause synchronization inaccuracies, and PTP-supported switches report their queueing delays so endpoints can compensate. However, Fastpass keeps queue-length variability low, enabling high quality time synchronization without PTP switch support.

Client timing. The client uses Linux high-resolution timers (*hrtimers*), previously demonstrated to achieve microsecond-scale precision when shaping flow throughput [33].

The client uses locks to synchronize access to the *qdisc* queues and to allocation state. Because waiting for these locks when transmitting packets causes variable delays, we allow the *qdisc* to send packets after their scheduled times, up to a configurable threshold. Endpoints re-request overly late allocations from the arbiter.

7. EVALUATION

The goal of Fastpass is to simultaneously eliminate in-network queueing, achieve high throughput, and support various inter-flow or inter-application resource allocation objectives in a real-world data-center network. In this section, we evaluate how well Fastpass meets these goals, compared to a baseline datacenter network running TCP.

Summary of Results

§7.1	(A) Under a bulk transfer workload involving multiple machines, Fastpass reduces median switch queue length to 18 KB from 4351 KB, with a 1.6% throughput penalty. (B) Interactivity: under the same workload, Fastpass’s median ping time is 0.23 ms vs. the baseline’s 3.56 ms, 15.5× lower with Fastpass.
§7.2	(C) Fairness: Fastpass reduces standard deviations of per-sender throughput over 1 s intervals by over 5200× for 5 connections.
§7.3	(D) Each comm-core supports 130 Gbits/s of network traffic with 1 μs of NIC queueing. (E) Arbiter traffic imposes a 0.3% throughput overhead. (F) 8 alloc-cores support 2.2 Terabits/s of network traffic. (G) 10 pathsel-cores support >5 Terabits/s of network traffic.
§7.4	(H) In a real-world latency-sensitive service, Fastpass reduces TCP retransmissions by 2.5×.

Experimental setup. We conducted experiments on a single rack of 32 servers, with each server connected to a top-of-rack (ToR) switch with a main 10 Gbits/s Ethernet (GbE) network interface card (NIC). Servers also have a 1 Gbit/s NIC meant for out-of-band communication. The 10 GbE top-of-rack switch has four 10 GbE uplinks to the four cluster switches [16]. Each server has 2 Intel CPUs with 8 cores each (16 hyper-threads per CPU, for a total of 32 hyper-threads) and 148 GB RAM. One server is set aside for running the arbiter. We turn off TCP segmentation offload (TSO) to achieve better control over the NIC send queues.

7.1 Throughput, queueing, and latency

Experiment A: throughput and queueing. Our first experiment compares the throughput and switch queue occupancy of Fastpass to the baseline network. Four rack servers run `iperf` to generate traffic (20 TCP flows per sender) to a single receiver. The experiment lasts 20 minutes and is run twice—once with the baseline and once with Fastpass.

Fastpass achieves throughput close to the baseline’s: 9.43 Gbits/s in the baseline run versus 9.28 Gbits/s with Fastpass. At the same time, Fastpass reduces the median switch queue occupancy from 4.35 Megabytes in the baseline to just 18 kilobytes with Fastpass, a reduction of a factor of 242× (Fig. 7).

It isn’t just the median but also the tails of the queue-occupancy distribution that are a lot lower, as shown here:

	Median	90 th %ile	99 th	99.9 th
Baseline (Kbytes)	4351	5097	5224	5239
Fastpass (Kbytes)	18	36	53	305

Most of the 1.6% difference in throughput can be attributed to Fastpass reserving about 1% of the achieved throughput for FCP traffic. The remainder is due to the client re-requesting timeslots when packet transmissions were delayed more than the allowed threshold (§6.3).

Switch queues are mostly full in the baseline because TCP continues to increase the sending rate until a packet is dropped (usually due to a full queue). In contrast, Fastpass’s timeslot allocation keeps queues relatively empty: the 99.9th percentile occupancy was 305 KB over the entire experiment. Although the design intends queues to be strictly 0, the implementation does not yet achieve it because of jitter in endpoint transmission times. We believe that queueing

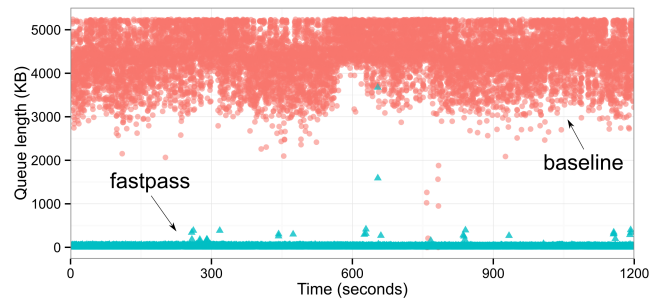


Figure 7: Switch queue lengths sampled at 100ms intervals on the top-of-rack switch. The diagram shows measurements from two different 20 minute experiments: baseline (red) and Fastpass (blue). Baseline TCP tends to fill switch queues, whereas Fastpass keeps queue occupancy low.

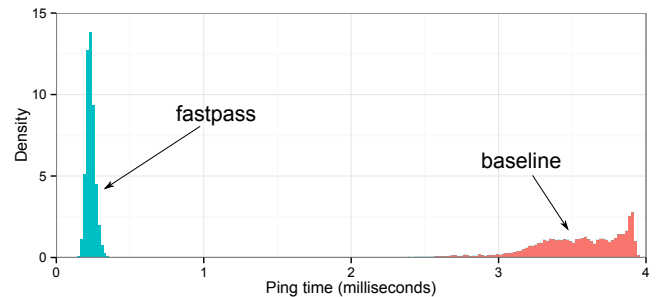


Figure 8: Histogram of ping RTTs with background load using Fastpass (blue) and baseline (red). Fastpass’s RTT is 15.5× smaller, even with the added overhead of contacting the arbiter.

can be further reduced towards the zero ideal by using finer-grained locking in the kernel module.

Experiment B: latency. Next, we measure the round-trip latency of interactive requests under high load. This experiment uses the same setup as Experiment A, augmented with a fifth machine that sends a small request to the receiving server every 10 ms, using ping.

Fastpass reduces the end-to-end round-trip time (RTT) for interactive traffic when the network is heavily loaded by a factor of 15.5×, from a median of 3.56 ms to 230 μs (Fig. 8). The tail of the distribution observes significant reductions as well:

	Median	90 th %ile	99 th	99.9 th
Baseline (ms)	3.56	3.89	3.92	3.95
Fastpass (ms)	0.23	0.27	0.32	0.38

Note that with Fastpass ping packets are scheduled in both directions by the arbiter, but even with the added round-trips to the arbiter, end-to-end latency is substantially lower because queues are much smaller. In addition, Fastpass achieves low latency for interactive traffic without requiring the traffic to be designated explicitly as “interactive” or “bulk,” and without using any mechanisms for traffic isolation in the switches: it results from the fairness properties of the timeslot allocator.

7.2 Fairness and convergence

Experiment C: fairness and convergence. Here we examine how fairly Fastpass allocates throughput to multiple senders, and how quickly it converges to a fair allocation when a sender arrives or departs. Five rack servers each send a bulk TCP flow to a sixth receiving server. The experiment begins with one bulk flow; every

30 seconds, a new bulk flow arrives until all five are active for 30 seconds, and then one flow terminates every 30 seconds. The entire experiment therefore lasts 270 seconds.

We calculate each connection’s throughput over 1-second intervals. The resulting time series for the baseline TCP and for Fastpass are shown in Figure 9.

The baseline TCPs exhibit much larger variability than Fastpass. For instance, when the second connection starts, its throughput is about 20-25% higher than the first connection throughout the 30-second interval; similarly, when there are two senders remaining between time 210 to 240 seconds, the throughputs “cross over” and are almost never equal. With more connections, the variation in throughput for TCP is more pronounced than with Fastpass.

To quantify this observation, we calculate the standard deviation of the per-connection throughputs achieved in each 1-second interval, in Mbits/s, when there are 3, 4, and 5 concurrent connections each for the baseline TCP and Fastpass. We then compute the median over all standard deviations for a given number of connections (a median over 60 values when there are 3 or 4 connections and over 30 values when there are 5 connections). The results are:

#connections	Baseline	Fastpass	Improvement
3	543.86	15.89	34.2×
4	628.49	0.146	4304.7×
5	459.75	0.087	5284.5×

These results show that Fastpass exhibits significantly lower variability across the board: its standard deviation of throughput is over 5200 **times** lower than the baseline when there are five concurrent connections.

Fastpass’s pipelined timeslot allocation algorithm prioritizes flows based on their last transmission time, so when a new flow starts, it is immediately allocated a timeslot. From that point on, all flows contending for the bottleneck will be allocated timeslots in sequence, yielding immediate convergence and perfect fairness over intervals as small as 1 MTU per flow (for 5 flows on 10 Gbits/s links, this yields fairness at the granularity of 6 μ s).

The benchmark shows low total throughput for one and two senders because of packet processing overheads, which are usually reduced by TSO. (In contrast, Experiments A and B use many more connections, so they achieve high utilization). Fastpass senders also require additional processing in the Fastpass qdisc, which is limited to using one core; NIC support (§8.3) or a multicore implementation will alleviate this bottleneck.

7.3 Arbiter performance

Experiment D: request queueing. To estimate how long requests queue at the arbiter before they are processed, we measure the NIC polling rate of the comm-core under increasing amounts of network traffic. Every 10 seconds, a rack server starts a TCP transfer to an unloaded server.

As the load increases, the arbiter spends more time processing requests, the NIC polling rate decreases (Fig. 10), and requests are delayed in the arbiter’s NIC queues. A deployment can control this queueing delay by limiting the amount of traffic each comm-core handles: 130 Gbits/s for 1 μ s queueing, 170 Gbits/s for 10 μ s, etc.

Experiment E: communication overhead. To determine the network capacity requirements at the arbiter, we measure the total amount of control traffic the arbiter receives and transmits in experiment D. The network overhead of communication with the arbiter is 1-to-500 for request traffic and 1-to-300 for allocations for the tested workload (Fig. 11): to schedule as much as 150 Gbits/s, the comm core receives less than 0.3 Gbits/s of requests and sends out 0.5 Gbits/s of allocations. When the NIC polling rate decreases

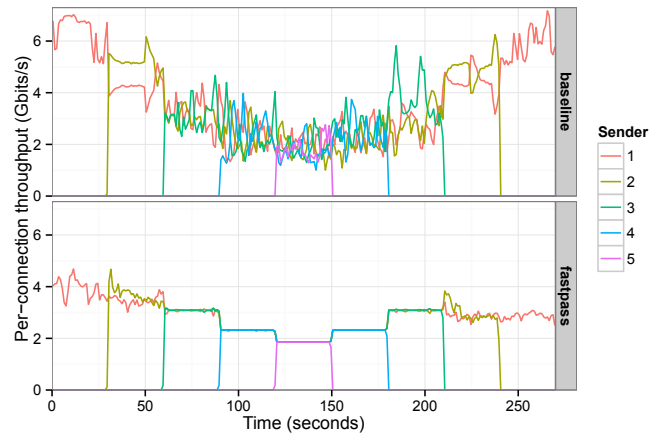


Figure 9: Each connection’s throughput, with a varying number of senders. Even with 1s averaging intervals, baseline TCP flows achieve widely varying rates. In contrast, for Fastpass (bottom), with 3, 4, or 5 connections, the throughput curves are on top of one another. The Fastpass max-min fair timeslot allocator maintains fairness at fine granularity. The lower one- and two-sender Fastpass throughput is due to Fastpass qdisc overheads (§7.2).

sufficiently, incoming request packets start getting dropped (seen around 160 Gbits/s). The arbiter is still able to allocate all traffic because FCP retransmissions summarize aggregate demands; hence, not every demand packet needs to be received.

Experiment F: timeslot allocation cores. To determine the number of arbiter cores necessary for timeslot allocation, we measure the throughput of the max-min fair timeslot allocation implementation. Requests are generated by a synthetic stress-test-core, rather than received from a comm-core. The workload has Poisson arrivals, senders and receivers chosen uniformly at random from 256 nodes, and requests are for 10 MTUs. We vary the mean inter-arrival time to produce different network loads.

	2 cores	4 cores	6 cores	8 cores
Throughput (Gbits/s)	825.6	1545.1	1966.4	2211.8

Eight alloc-cores support over 2.21 Terabits/s of network traffic, or 221 endpoints transmitting at a full 10 Gbits/s. This corresponds to over 86% network utilization.

Experiment G: path selection cores. To determine the number of arbiter cores needed for path selection, we measure the average processing time per timeslot as load increases. We use the synthetic workload described above with exponentially distributed request sizes (with a mean of 10 MTUs). The chosen topology has 32 machines per rack and four paths between racks, with no over-subscription (motivated in part by the “4-post” cluster topology [16]). Note that non-over-subscribed topologies could be considered worst-case topologies for path selection: over-subscription reduces the amount of traffic leaving each rack and thus simplifies path-selection.

Fig. 12 shows that the processing time increases with network utilization until many of the nodes reach full degree (32 in the tested topology), at which point the cost of pre-processing⁴ the graph decreases, and path selection runs slightly faster.

Because path-selection can be parallelized by having a different core select paths for each timeslot, these measurements indicate how many pathsel-cores are required for different topologies. For exam-

⁴Path selection adds dummy edges to the graph until all nodes have the same degree (i.e., number of packets).

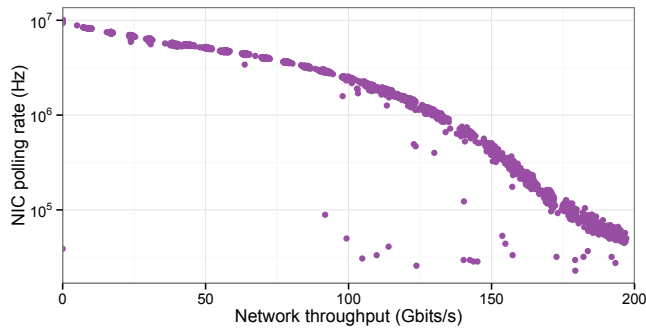


Figure 10: As more requests are handled, the NIC polling rate decreases. The resulting queuing delay can be bounded by distributing request-handling across multiple comm-cores.

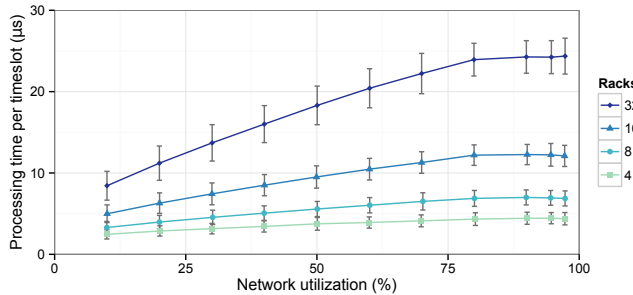


Figure 12: Path selection routes traffic from 16 racks of 32 endpoints in $<12\mu\text{s}$. Consequently, 10 pathsel-cores would output a routing every $<1.2\mu\text{s}$, fast enough to support 10 Gbits/s endpoint links. Error bars show one standard deviation above and below the mean.

ple, path selection of 16 racks can be done in under 12 microseconds; hence, for 1.2 microsecond timeslots, 10 pathsel-cores suffice.

7.4 Production experiments at Facebook

Workload. We deployed Fastpass on a latency-sensitive service that is in the response path for user web requests. This service has a partition-aggregate workload similar to common search workloads [10]. Each server runs both an aggregator and a leaf; when an aggregator receives a query from a web server, it requests relevant data from several leaves, processes them, and returns a reduced result. Each rack of 30–34 aggregator-leaf servers works independently.

To maintain acceptable latency as traffic load changes during the day, the service adjusts the number of internal queries generated by each request; in aggregate, a rack handles between 500K and 200M internal queries per second. When load is low, the aggregator considers more items and produces higher quality results. In times of heavy load, when the 99th percentile latency increases, the number of items considered per web request is reduced aggressively.

Cluster traffic is bursty, but most of the time utilizes a fraction of network capacity (Fig. 13). We measure throughput over 100 μs time intervals on one production server. 25% of these intervals have no ingress traffic, 25% have no egress traffic, and only 10% of these intervals have aggregate traffic exceeding 2 Gbits/s.

The production workload changes gently over time scales of tens of minutes, enabling comparative testing when schemes are applied in sequence. The live experiment ran for 135 minutes: the rack started in baseline mode, switched to Fastpass at 30 minutes, and back to baseline at 110 minutes.

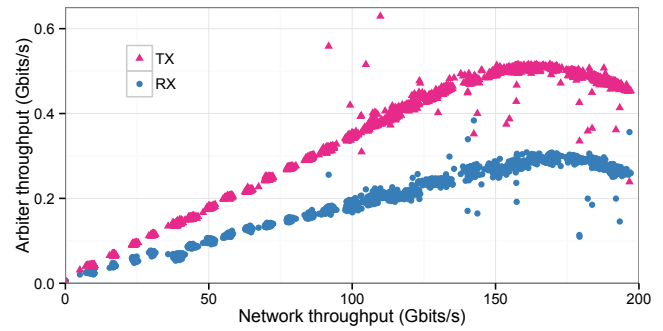


Figure 11: The arbiter requires 0.5 Gbits/s TX and 0.3 Gbits/s RX bandwidth to schedule 150 Gbits/s: around 0.3% of network traffic.

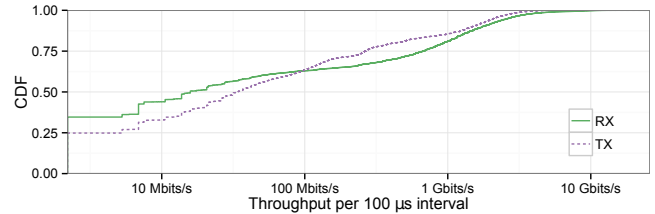


Figure 13: Distribution of the sending and receiving rates of one production server per 100 microsecond interval over a 60 second trace.

Experiment H: production results. Fig. 14 shows that the 99th percentile web request service time using Fastpass is very similar to the baseline’s. The three clusters pertain to groups of machines that were assigned different load by the load-balancer. Fig. 15 shows the cluster’s load as the experiment progressed, showing gentle oscillations in load. Fastpass was able to handle the load without triggering the aggressive load-reduction.

Fastpass reduced TCP retransmissions by 2–2.5 \times (Fig. 16). We believe the remaining packet loss is due to traffic exiting the rack, where Fastpass is not used to keep switch queues low. Extending the Fastpass scheduling boundary should further reduce this packet loss.

8. DISCUSSION

8.1 Large deployments

A single arbiter should be able to handle hundreds to thousands of endpoints. At larger scales, however, a single arbiter’s computational and network throughput become bottlenecks, and several arbiters would need to cooperate.

A hierarchical approach might be useful: an arbiter within each cluster would send its demands for intra-cluster traffic to a core-arbiter, which would decide which timeslots each cluster may use, and what paths packets at these timeslots must follow. The cluster arbiters would then pass on these allocations to endpoints.

A core arbiter would have to handle a large volume of traffic, so allocating at MTU-size granularity would not be computationally feasible. Instead, it could allocate timeslots in bulk, and trust cluster arbiters to assign individual timeslots fairly among endpoints.

An alternative for large deployments could be the use of specialized hardware. An FPGA or ASIC implementation of timeslot allocation and path selection algorithms would likely allow a single arbiter to support much larger clusters.

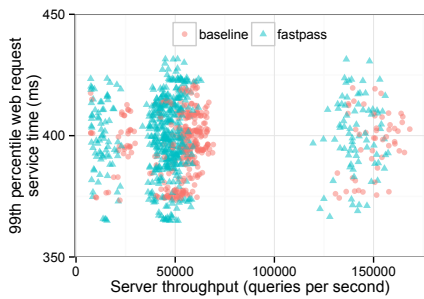


Figure 14: 99th percentile web request service time vs. server load in production traffic. Fastpass shows a similar latency profile as baseline.

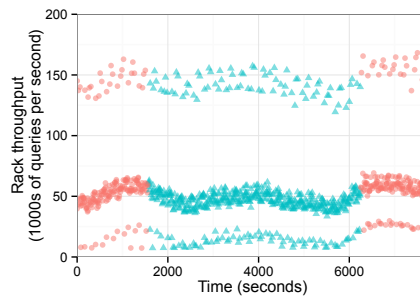


Figure 15: Live traffic server load as a function of time. Fastpass is shown in the middle with baseline before and after. The offered load oscillates gently with time.

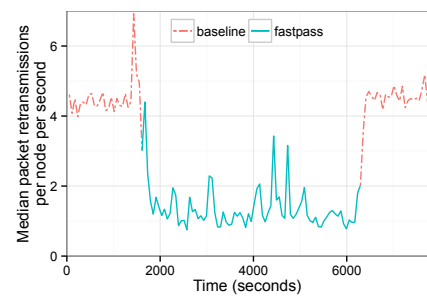


Figure 16: Median server TCP retransmission rate during the live experiment. Fastpass (middle) maintains a $2.5\times$ lower rate of retransmissions than baseline (left and right).

8.2 Routing packets along selected paths

Packets must be made to follow the paths allocated to them by the arbiter. Routers typically support IP source routing only in software, if at all, rendering it too slow for practical use. Static routing [38] and policy based routing using VLANs are feasible, but could interfere with existing BGP configurations, making them less suitable for existing clusters. Tunneling (e.g. IP-in-IP or GRE) entails a small throughput overhead, but is supported in hardware by many switch chipsets making it a viable option [18].

Finally, routing along a specific path can be achieved by what we term *ECMP spoofing*. ECMP spoofing modifies fields in the packet header (e.g., source port) to specific values that will cause each switch to route the packet to the desired egress, given the other fields in the packet header and the known ECMP hash function. The receiver can then convert the modified fields to their original values, stored elsewhere in the packet.

8.3 Scheduling support in NICs

Fastpass enqueues packets into NIC queues at precise times, using high resolution timers. This frequent timer processing increases CPU overhead, and introduces operating-system jitter (e.g., due to interrupts). These timers would not be necessary if NICs implement support for precise packet scheduling: packets could be enqueued when Fastpass receives an allocation. A “send-at” field in the packet descriptor would indicate the desired send time.

8.4 Small packet workloads

A current limitation of Fastpass is that it allocates at the granularity of timeslots, so if an endpoint has less than a full timeslot worth of data to send, network bandwidth is left unused. This waste is reduced or eliminated when an endpoint sends many small packets to the same destination, which are batched together (§6.1). Workloads that send tiny amounts of data to a large number of destinations still waste bandwidth. A possible mitigation is for the arbiter to divide some timeslots into smaller fragments and allocate these fragments to the smaller packets.

9. RELATED WORK

Several systems use centralized controllers to get better load balance and network sharing, but they work at “control-plane” granularity, which doesn’t provide control over packet latencies or allocations over small time scales.

Hedera [5] discovers elephant flows by gathering switch statistics, then reroutes elephants for better load balancing. It aims for high throughput, and has no mechanism to reduce network latency.

Datacenter TDMA [35] and Mordia [17] use gathered statistics to estimate future demand, then compute a set of matchings that are applied in sequence to timeslots of duration on the order of $100\mu\text{s}$. Both schemes target elephant flows; short flows are delegated to other means.

Orchestra [11] coordinates transfers in the shuffle stage of MapReduce/Dryad so all transfers finish roughly together, by assigning weights to each transfer. Orchestra is an application-level mechanism; the actual transfers use TCP.

SWAN [19] frequently reconfigures the network’s data plane to match demand from different services. All non-interactive services coordinate with the central SWAN controller, which plans paths and bandwidth and implements those paths by updating forwarding tables in network devices.

Distributed approaches usually set out to solve a restricted data-center problem: minimizing Flow Completion Time (FCT), meeting deadlines, balancing load, reducing queueing, or sharing the network. To our knowledge, no previous scheme provides a general platform to support all these features, and some schemes perform sub-optimally because they lack complete knowledge of network conditions.

DCTCP [6] and HULL [7] reduce switch queueing, but increase convergence time to a fair-share of the capacity, and do not eliminate queueing delay.

In MATE [15] and DARD [38], ingress nodes reroute traffic selfishly between paths until converging to a configuration that provides good load balance across paths.

In multi-tenant datacenters, Seawall [32] provides weighted fair sharing of network links, and EyeQ [22] enforces traffic constraints.

Schemes such as pFabric [8] and PDQ [20] modify the switches to reduce flow completion time, while D^3 [36] and PDQ minimize flow completion times to meet deadlines. These switch modifications raise the barrier to adoption because they need to be done across the network. PDQ and pFabric use a distributed approximation of the *shortest remaining flow first* policy, which Fastpass can implement in the arbiter.

Differentiated Services (DiffServ, RFC2474) provides a distributed mechanism for different classes of traffic to travel via distinct queues in routers. The number of DiffServ Code Points available is limited, and in practice operational concerns restrict the number of classes even further. Most commonly, there are classes for bulk traffic and latency-sensitive traffic, but not a whole lot more.

10. CONCLUSION

We showed how to design and implement a datacenter network in which a centralized arbiter determines the times at which each packet should be transmitted and the path it should take. Our results show that compared to the baseline network, the throughput penalty is small but queueing delays reduce dramatically, flows share resources more fairly and converge quickly, and the software arbiter implementation scales to multiple cores and handles an aggregate data rate of 2.21 Terabits/s.

Even with such a seemingly heavy-handed approach that incurs a little extra latency to and from the arbiter, tail packet latencies and the number of retransmitted packets reduce compared to the status quo, thanks to the global control exerted by the arbiter. Our results show that Fastpass opens up new possibilities for high-performance, tightly-integrated, predictable network applications in datacenters, even with commodity routers and switches.

We hope we have persuaded the reader that centralized control at the granularity of individual packets, achieved by viewing the entire network as a large switch, is both practical and beneficial.

Acknowledgements

We are grateful to Omar Baldonado and Sanjeev Kumar of Facebook for their enthusiastic support of this collaboration, Petr Lapukhov and Doug Weimer for their generous assistance with the Facebook infrastructure, Garrett Wollman and Jon Proulx at MIT CSAIL for their help and efforts in setting up environments for our early experiments, and David Oran of Cisco for his help. We thank John Ousterhout, Rodrigo Fonseca, Nick McKeown, George Varghese, Chuck Thacker, Steve Hand, Andreas Nowatzky, Tom Rodeheffer, and the SIGCOMM reviewers for their insightful feedback. This work was supported in part by the National Science Foundation grant IIS-1065219. Ousterhout was supported by a Jacobs Presidential Fellowship and a Hertz Foundation Fellowship. We thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing for their support and encouragement.

11. REFERENCES

- [1] Packet processing on intel architecture.
<http://www.intel.com/go/dpdk>.
- [2] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-029. March 2014.
- [3] M. Ajmone Marsan, E. Leonardi, M. Mellia, and F. Neri. On the stability of input-buffer cell switches with speed-up. In *INFOCOM*, 2000.
- [4] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
- [5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.
- [6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, C. Faster, and D. Maltz. DCTCP: Efficient Packet Transport for the Commodified Data Center. In *SIGCOMM*, 2010.
- [7] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.
- [8] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing Datacenter Packet Transport. In *HotNets*, 2012.
- [9] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-Speed Switch Scheduling for Local-Area Networks. *ACM Trans. on Comp. Sys.*, 11(4):319–352, 1993.
- [10] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.
- [11] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *SIGCOMM*, 2011.
- [12] R. Cole, K. Ost, and S. Schirra. Edge-Coloring Bipartite Multigraphs in $O(E \log D)$ Time. *Combinatorica*, 21(1):5–12, 2001.
- [13] J. Dai and B. Prabhakar. The throughput of data switches with and without speedup. In *INFOCOM*, 2000.
- [14] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. Morgan Kaufmann, 2003.
- [15] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM*, 2001.
- [16] N. Farrington and A. Andreyev. Facebook’s Data Center Network Architecture. In *IEEE Optical Interconnects Conf.*, 2013.
- [17] N. Farrington, G. Porter, Y. Fainman, G. Papen, and A. Vahdat. Hunting Mice with Microsecond Circuit Switches. In *HotNets*, 2012.
- [18] 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.
- [19] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.
- [20] Hong, C. Y. and Caesar, M. and Godfrey, P. Finishing Flows Quickly with Preemptive Scheduling. *SIGCOMM*, 2012.
- [21] F. Hwang. Control Algorithms for Rearrangeable Clos Networks. *IEEE Trans. on Comm.*, 31(8):952–954, 1983.
- [22] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, and C. Kim. EyeQ: Practical Network Performance Isolation for the Multi-Tenant Cloud. In *HotCloud*, 2012.
- [23] A. Kapoor and R. Rizzi. Edge-coloring bipartite graphs. *Journal of Algorithms*, 34(2):390–396, 2000.
- [24] N. McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Trans. on Net.*, 7(2):188–201, 1999.
- [25] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand. Achieving 100% Throughput in an Input-Queued Switch. *IEEE Trans. Comm.*, 47(8):1260–1267, 1999.
- [26] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *Supercomputing*, 2004.
- [27] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, 2009.
- [28] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.
- [29] P. Ohly, D. N. Lombard, and K. B. Stanton. Hardware Assisted Precision Time Protocol. Design and Case Study. In *LCI Intl. Conf. on High-Perf. Clustered Comp.*, 2008.
- [30] D. Shah. Maximal matching scheduling is good enough. In *GLOBECOM*, 2003.
- [31] D. Shah, N. Walton, and Y. Zhong. Optimal Queue-Size Scaling in Switched Networks. In *SIGMETRICS*, 2012.

- [32] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *NSDI*, 2011.
- [33] R. Takano, T. Kudoh, Y. Kodama, and F. Okazaki. High-Resolution Timer-Based Packet Pacing Mechanism on the Linux Operating System. *IEICE Trans. on Comm.*, 2011.
- [34] Y. Tamir and H.-C. Chi. Symmetric crossbar arbiters for VLSI communication switches. *IEEE Trans. Par. Dist. Sys.*, 4(1):13–27, 1993.
- [35] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. In *EuroSys*, 2012.
- [36] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.
- [37] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNext*, 2010.
- [38] X. Wu and X. Yang. DARD: Distributed Adaptive Routing for Datacenter Networks. In *ICDCS*, 2012.

Appendix: Theoretical Properties of the Timeslot Allocator

System model. We consider a network with N endpoints denoted by $1, \dots, N$, with each endpoint potentially having requests for any other endpoint. Time is slotted; in a unit timeslot each endpoint can transmit at most one MTU-sized packet to any other endpoint and receive at most one packet from any other endpoint. New requests arrive at an endpoint i for endpoint j with probability p_{ij} in each timeslot. An arriving request brings a random number of packets, distributed according to a distribution G_{ij} and independent of everything else.

Let $\mathbb{E}[G_{ij}] = g_{ij}$. Thus, on average $\lambda_{ij} = p_{ij}g_{ij}$ packets arrive at endpoint i for endpoint j per timeslot. The matrix $\Lambda = [\lambda_{ij}]$ denotes the net average data traffic between endpoints arriving in each timeslot. We assume a non-oversubscribed (also called full bisection bandwidth) network: the network can support any traffic where each node is paired to at most one other node per timeslot.

In the above setup, all traffic matrices, Λ , that can be served by any system architecture, must be *doubly sub-stochastic*. That is,

$$\sum_{k=1}^N \lambda_{ik} < 1, \text{ for all } i \text{ and } \sum_{k=1}^N \lambda_{kj} < 1, \text{ for all } j. \quad (1)$$

By the celebrated result of Birkhoff and von Neumann, all doubly stochastic matrices can be decomposed as a weighted sum of permutation matrices (i.e., matchings) with the sum of the weights being at most 1. Therefore, non-oversubscribed networks can support all doubly stochastic traffic matrices. A traffic matrix Λ is called *admissible* if and only if $\rho(\Lambda) < 1$ where, the system *load* $\rho(\Lambda)$ is defined as

$$\rho(\Lambda) = \max_{i,j} \left(\sum_{k=1}^N \lambda_{ik}, \sum_{k=1}^N \lambda_{kj} \right). \quad (2)$$

Finally, let $Q_{ij}(t)$ denote the total number of packets (potentially across different requests) waiting to be transferred from endpoint i to endpoint j at time t . This setup is similar to that used in literature on input-queued switches [25], enabling us to view the network as a big input-queued switch with $Q_{ij}(t)$ the Virtual Output Queue sizes.

Main result. The arbiter’s timeslot allocation algorithm of §3 is equivalent to the following: each queue (i, j) has a “priority score” associated with it. In the beginning of each timeslot, the arbiter starts processing queues in non-decreasing order of these priority scores. While processing, the arbiter allocates a timeslot to queue (i, j) if

the arbiter has not already allocated another packet starting from i or destined to j in this timeslot. Therefore, at the end of processing the timeslot, the allocations correspond to a *maximal matching* between endpoints in the bipartite graph between endpoints, where an edge is present between (i, j) if there are packets waiting at endpoint i destined for j . From the literature on input-queued switches, it is well-known that any maximal matching provides 50% throughput guarantees [13, 3]. Building upon these results as well as [30], we state the following property of our algorithm.

THEOREM 1. *For any $\rho < 1$, there exists Λ with $\rho(\Lambda) = \rho$ such that for any allocator,*

$$\liminf_t \mathbb{E} \left[\sum_{ij} Q_{ij}(t) \right] \geq \frac{N\rho}{2(1-\rho)}. \quad (3)$$

Further, let $V \geq 1$ be such that $\mathbb{E}[G_{ij}^2] \leq V\mathbb{E}[G_{ij}]$ for all i, j (bounded G_{ij}); if we allow the Fastpass arbiter to schedule (as well as transmit through the network) twice per unit timeslot,⁵ then the induced average queue-size

$$\limsup_t \mathbb{E} \left[\sum_{ij} Q_{ij}(t) \right] \leq \frac{N\rho(\rho+V)}{2(1-\rho)}. \quad (4)$$

Proof Sketch. To establish the lower bound (3) for any scheduling algorithm, it is sufficient to consider a specific scenario of our setup. Concretely, let the traffic matrix be uniform, i.e., $\Lambda = [\lambda_{ij}]$ with $\lambda_{ij} = \frac{\rho}{(N-1)}$ for all $i \neq j$ and 0 when $i = j$; $p_{ij} = 1$ for all $i \neq j$; and let G_{ij} be Poisson variables with parameter λ_{ij} . The network can be viewed as unit-sized packets arriving at each endpoint according to a Poisson arrival process of rate ρ and processed (transferred by the network) at unit rate. That is, the queue-size for each endpoint j is bounded below by that of an $M/D/1$ queue with load ρ , which is known to be bounded below by $\frac{\rho}{(2(1-\rho))}$ [31]. Therefore, the network-wide queue-size is bounded below by $\frac{N\rho}{(2(1-\rho))}$.

To establish an upper bound, we use the fact that the algorithm effectively achieves a maximal matching in the weighted bipartite graph between endpoints in each timeslot. Given this fact, and under the assumption that Fastpass can schedule as well as transfer data at twice the speed, this is effectively a *speedup* of 2 in the classical terminology of switch scheduling. Therefore, for the Lyapunov function (cf. [13]),

$$L(t) = \sum_{i,j} Q_{ij}(t) [Q_i(t) + Q_j(t)]$$

it can be shown using calculations similar to [30] that

$$\mathbb{E}[L(t+1) - L(t) | Q(t)] \leq 4(\rho - 1) \left(\sum_{i,j} Q_{ij}(t) \right) + 2N\rho^2 + 2VN\rho.$$

Telescoping this inequality for $t \geq 0$ and using the fact that the system reaches equilibrium due to ergodicity, we obtain the desired result. ■

Implications. Equation (3) says that there is some (worst case) input workload for which any allocator will have an expected aggregate queue length at least as large as $\frac{N\rho}{2(1-\rho)}$. Equation (4) says that with a speedup of 2 in the network fabric, for every workload, the expected aggregate queue length will be no larger than $\frac{N\rho(\rho+V)}{2(1-\rho)}$. Here V is effectively a bound on burst size; if it is small, say 1, then it is within a factor of 2 of the lower bound! There is, however, a gap between theory and practice here, as in switch scheduling; many workloads observed in practice seem to require only small queues even with no speedup.

⁵Equivalent to having to double the network fabric bandwidth.