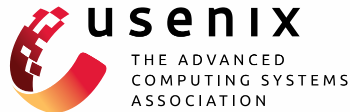


Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching

Erico Vanini and Rong Pan, ***Cisco Systems;***
Mohammad Alizadeh, ***Massachusetts Institute of Technology;***
Parvin Taheri and Tom Edsall, ***Cisco Systems***

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching

Erico Vanini* Rong Pan* Mohammad Alizadeh† Parvin Taheri* Tom Edsall*
*Cisco Systems †Massachusetts Institute of Technology

Abstract

Datacenter networks require efficient multi-path load balancing to achieve high bisection bandwidth. Despite much progress in recent years towards addressing this challenge, a load balancing design that is both simple to implement and resilient to network asymmetry has remained elusive. In this paper, we show that *flowlet switching*, an idea first proposed more than a decade ago, is a powerful technique for resilient load balancing with asymmetry. Flowlets have a remarkable *elasticity* property: their size changes automatically based on traffic conditions on their path. We use this insight to develop LetFlow, a very simple load balancing scheme that is resilient to asymmetry. LetFlow simply picks paths at random for flowlets and lets their elasticity naturally balance the traffic on different paths. Our extensive evaluation with real hardware and packet-level simulations shows that LetFlow is very effective. Despite being much simpler, it performs significantly better than other traffic oblivious schemes like WCMP and Presto in asymmetric scenarios, while achieving average flow completions time within 10-20% of CONGA in testbed experiments and 2× of CONGA in simulated topologies with large asymmetry and heavy traffic load.

1 Introduction

Datacenter networks must provide large bisection bandwidth to support the increasing traffic demands of applications such as big-data analytics, web services, and cloud storage. They achieve this by load balancing traffic over many paths in multi-rooted tree topologies such as Clos [13] and Fat-tree [1]. These designs are widely deployed; for instance, Google has reported on using Clos fabrics with more than 1 Pbps of bisection bandwidth in its datacenters [25].

The standard load balancing scheme in today's datacenters, Equal Cost MultiPath (ECMP) [16], randomly assigns flows to different paths using a hash taken over packet headers. ECMP is widely deployed due to its simplicity but suffers from well-known performance problems such as hash collisions and the inability to adapt to asymmetry in the network topology. A rich body of work [10, 2, 22, 23, 18, 3, 15, 21] has thus emerged on better load balancing designs for datacenter networks.

A defining feature of these designs is the information that they use to make decisions. At one end of the spectrum are designs that are oblivious to traffic conditions [16, 10, 9, 15] or rely only on local measurements [24, 20] at the switches. ECMP and Presto [15], which picks paths in round-robin fashion for fixed-sized chunks of data (called "flowcells"), fall in this category. At the other extreme are designs [2, 22, 23, 18, 3, 21, 29] that use knowledge of traffic conditions and congestion on different paths to make decisions. Two recent examples are CONGA [3] and HULA [21], which use feedback between the switches to gather path-wise congestion information and shift traffic to less-congested paths.

Load balancing schemes that require path congestion information, naturally, are much more complex. Current designs either use a centralized fabric controller [2, 8, 22] to optimize path choices frequently or require non-trivial mechanisms, at the end-hosts [23, 18] or switches [3, 21, 30], to implement end-to-end or hop-by-hop feedback. On the other hand, schemes that lack visibility into path congestion have a key drawback: they perform poorly in *asymmetric topologies* [3]. As we discuss in §2.1, the reason is that the optimal traffic split across asymmetric paths depends on (dynamically varying) traffic conditions; hence, traffic-oblivious schemes are fundamentally unable to make optimal decisions and can perform poorly in asymmetric topologies.

Asymmetry is common in practice for a variety of reasons, such as link failures and heterogeneity in network equipment [31, 12, 3]. Handling asymmetry gracefully, therefore, is important. This raises the question: *are there simple load balancing schemes that are resilient to asymmetry?* In this paper, we answer this question in the affirmative by developing LetFlow, a simple scheme that requires no state to make load balancing decisions, and yet it is very resilient to network asymmetry.

LetFlow is *extremely* simple: switches pick a path at random for each *flowlet*. That's it! A flowlet is a burst of packets that is separated in time from other bursts by a sufficient gap — called the "flowlet timeout". Flowlet switching [27, 20] was proposed over a decade ago as a way to split TCP flows across multiple paths without causing packet reordering. Remarkably, as we uncover in this paper, flowlet switching is also a powerful technique

for resilient load balancing.

The reason for this resilience is that flowlet sizes are *elastic* and change based on traffic conditions on different paths. On slow paths, with low per-flow bandwidth and high latency, flowlets tend to be smaller because there is a greater chance of a flowlet timeout (a large inter-packet gap for a flow). On fast paths, on the other hand, flowlets grow larger since flowlet timeouts are less likely to occur than on slow paths. This elasticity property is rooted in the fact that higher layer congestion control protocols like TCP react to traffic conditions on the flow's path, slowing down on congested paths (which leads to smaller flowlets) and speeding up on uncongested paths (which causes larger flowlets).

As a result of their elasticity, flowlets can compensate for inaccurate load balancing decisions, e.g., decisions that send an incorrect proportion of flowlets on different paths. Flowlets accomplish this by changing size in a way that naturally shifts traffic away from slow (congested) paths and towards fast (uncongested) paths. Since flowlet-based load balancing decisions need not be accurate, they do not require explicit path congestion information or feedback.

The only requirement is that the load balancing algorithm should not predetermine how traffic is split across paths. Instead, it should allow flowlets to "explore" different paths and determine the amount of traffic on each path automatically through their (elastic) sizes. Thus, unlike schemes such as Flare [27, 20], which attempts to achieve a *target* traffic split with flowlets, LetFlow simply chooses a path at random for each flowlet.

We make the following contributions:

- We show (§2) that simple load balancing approaches that pick paths in a traffic-oblivious manner for each flow [31] or packet [15] perform poorly in asymmetric topologies, and we uncover flowlet switching as a powerful technique for resilient load balancing in the presence of asymmetry.
- We design and implement LetFlow (§3), a simple randomized load balancing algorithm using flowlets. LetFlow is easy to implement in hardware and can be deployed without any changes to end-hosts or TCP. We describe a practical implementation in a major datacenter switch.
- We analyze (§4) how LetFlow balances load on asymmetric paths via detailed simulations and theoretical analysis. For a simplified traffic model, we show that flows with a lower rate are more likely to experience a flowlet timeout, and that LetFlow tends to move flows to less-congested paths where they achieve a higher rate.
- We evaluate (§5) LetFlow extensively in a small hardware testbed and large-scale simulations across a large number of scenarios with realistic traffic patterns, different topologies, and different transport protocols. We find that LetFlow is very effective. It achieves average flow completion times within 10-20% of CONGA [3] in a real testbed and 2× of CONGA in simulations under high asymmetry and traffic load, and performs significantly better than competing schemes such as WCMP [31] and an idealized variant of Presto [15].

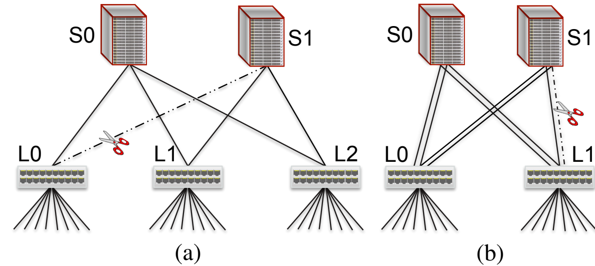


Figure 1: Two asymmetric topologies caused by link failure. All links run at 40 Gbps. Figure 1b is our baseline topology.

2 Motivation and Insights

The goal of this paper is to develop a simple load balancing scheme that is resilient to network asymmetry. In this section, we begin by describing the challenges created by asymmetry and the shortcomings of existing approaches (§2.1). We then present the key insights underlying LetFlow's flowlet-based design (§2.2).

2.1 Load balancing with Asymmetry

In asymmetric topologies, different paths between one or more source/destination pairs have different amounts of available bandwidth. Asymmetry can occur by design (e.g., in topologies with variable-length paths like BCube [14], Jellyfish [26], etc.), but most datacenter networks use symmetric topologies.[1] Nonetheless, asymmetry is difficult to avoid in practice: link failures and heterogeneous switching equipment (with different numbers of ports, link speeds, etc.) are common in large deployments and can cause asymmetry [31, 24, 3]. For instance, *imbalanced striping* [31], which occurs when the switch radix in one layer of a Clos topology is not divisible by the number of switches in an adjacent layer creates asymmetry (see [31] for details).

Figure 1 shows two basic asymmetric topologies that we will consider in this section. The asymmetry here is caused by the failure of a link. For example, in Figure 1a, the link between $L0$ and $S1$ is down, thus any $L0 \rightarrow L2$ traffic can only use the $L0 \rightarrow S0 \rightarrow L2$ path. This causes asymmetry for load balancing $L1 \rightarrow L2$ traffic.

[1] We focus on tree topologies [13, 1, 25] in this paper, since they are by far the most common in real deployments.

Scheme	Granularity	Information needed to make decisions	Handle asymmetry?
ECMP [16]	Flow	None	No
Random Packet Scatter [10]	Packet	None	No
Flare [20]	Flowlet	Local traffic	No
WCMP [31]	Flow	Topology	Partially
DRB [9]	Packet	Topology	Partially
Presto [15]	Flowcell (fixed-sized units)	Topology	Partially
LocalFlow [24]	Flow with selective splitting	Local traffic + Topology	Partially
FlowBender [18]	Flow (occasional rerouting)	Global traffic (per-flow feedback)	Yes
Hedera [2], MicroTE [8]	Flow (large flows only)	Global traffic (centralized controller)	Yes
Fastpass [22]	Packet	Global traffic (centralized arbiter)	Yes
DeTail [30]	Packet	Global traffic (hop-by-hop back-pressure)	Yes
MPTCP [23]	Packet	Global traffic (per-flow, per-path feedback)	Yes
CONGA [3]	Flowlet	Global traffic (per-path feedback)	Yes
HULA [21]	Flowlet	Global traffic (hop-by-hop probes)	Yes
LetFlow	**Flowlet**	**None (Implicit feedback via flowlet size)**	**Yes**

Table 1: Comparison of existing load balancing schemes and LetFlow. Prior designs either require explicit information about end-to-end (global) path traffic conditions, or cannot handle asymmetry.

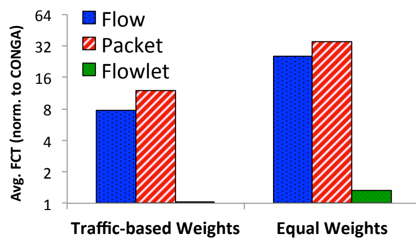


Figure 2: Load balancing dynamic traffic with asymmetry. Randomized per-flow and per-packet load balancing are significantly worse than CONGA, even with traffic-based (but static) weights; per-flowlet performs nearly as well as CONGA.

Why is asymmetry challenging? Load balancing in asymmetric topologies is difficult because the optimal split of traffic across paths in asymmetric topologies generally depends on real-time traffic demands and congestion on different paths [3]. By contrast, in symmetric topologies, splitting traffic equally across all (shortest) paths is always optimal, regardless of traffic conditions.

As an example, consider the topology in Figure 1a. Suppose the workload consists of $L0 \rightarrow L2$ and $L1 \rightarrow L2$ traffic. How should the $L1 \rightarrow L2$ traffic be split across the two paths through $S0$ and $S1$? It is not difficult to see that the ideal split depends on the amount of $L0 \rightarrow L2$ traffic. For example, if all the traffic is between $L1$ and $L2$, then we should send half of the traffic via $S0$, and half via $S1$. However, if there is 40 Gbps of $L0 \rightarrow L2$ traffic, then the $L1 \rightarrow L2$ traffic should avoid $S0$ as much as possible.

Table 1 compares several proposed load balancing schemes along two key dimensions: (1) the information they use to make load balancing decisions; and (2) the decision granularity (we discuss this aspect later). Load balancing designs that rely on *explicit* end-to-end (global) information about traffic conditions on different paths can handle asymmetry. There are many ways to collect this information with varying precision and complexity, ranging from transport-layer signals (e.g., ECN marks), centralized controllers, and in-network hop-by-hop or end-to-end feedback mechanisms. An example is CONGA [3], which uses explicit feedback loops between the top-of-rack (or "leaf") switches to collect per-path congestion information.

By contrast, schemes that are oblivious to traffic conditions generally have difficulty with asymmetry. This is the case even if different paths are weighed differently based on the topology, as some designs [31, 9, 15, 24] have proposed. Using the topology is better than nothing, but it does not address the fundamental problem of the optimal traffic splits depending on real-time traffic conditions. For instance, as we show next, knowing the topology does not help in the above scenario.

Asymmetry with dynamic workloads. Real datacenter traffic and congestion is highly dynamic [6, 7]. Since servers run at the same speed (or nearly the same speed) as network links, congestion can quickly arise as a few high rate flows start, and just as quickly dissipate as these flows end. The dynamism of real datacenter workloads makes load balancing in asymmetric topologies even more challenging, because the load balancing algorithm must adapt to fast-changing traffic conditions.

To illustrate the issue, consider again the topology in Figure 1a, and suppose that servers under switches $L0$ and $L1$ send traffic to servers under $L2$. The traffic is generated by randomly starting flows of finite length via a Poisson process, with a realistic distribution of flow sizes (details in §5). The average rate for $L0 \rightarrow L2$ and $L1 \rightarrow L2$ traffic are 20 Gbps and 48 Gbps respectively.

Ideally, the load balancer at leaf $L1$ should dynamically split traffic based on real-time $L0 \rightarrow L2$ traffic. We consider simple randomized load balancing instead, with (1) equal weights for the two paths; (2) a higher weight (roughly 2.4-to-1) for the $S1$-path compared to the $S0$-path. This weight is set statically to equalize the *average* load on the two paths. This represents a hypothetical system (e.g., a centralized controller) that optimizes the path weights using long-term traffic estimates.

Figure 2 shows the average FCT for both weight settings with load balancing decisions made on a per-flow, per-packet, and per-flowlet granularity. The results are normalized to the average FCT achieved by CONGA [3] for reference. The per-flow case is identical to ECMP and WCMP [31]. The per-packet case provides an up-

per bound on the performance of schemes like Random Packet Scatter [10] and Presto [15] that balance load on a finer granularity with static weights.[2] (We discuss the per-flowlet case later.) The results show that randomized traffic-oblivious load balancing at per-flow and per-packet granularity performs significantly worse than CONGA with both weight settings.

In summary, existing load balancing designs either explicitly use global path congestion information to make decisions or do poorly with asymmetry. Also, static weights based on the topology or coarse traffic estimates are inadequate in dynamic settings. In the next section, we describe how LetFlow overcomes these challenges.

2.2 Let the Flowlets *Flow*

Consider Figure 2 again. Remarkably, the same randomized load balancing scheme that did poorly at the per-flow and per-packet granularities performs very well when decisions are made at the level of *flowlets* [27, 20]. Picking paths uniformly at random per flowlet already does well; optimizing the path weights further improves performance and nearly matches CONGA.

Recall that flowlets are bursts of packets of the same flow that are sufficiently apart in time that they can be sent on different paths without causing packet reordering at the receiver. More precisely, the gap between two flowlets must exceed a threshold (the flowlet timeout) that is larger than the difference in latency among the paths; this ensures that packets are received in order.

Our key insight is that (in addition to enabling fine-grained load balancing) flowlets have a unique property that makes them resilient to inaccurate load balancing decisions: *flowlets automatically change size based on the extent of congestion on their path*. They shrink on slow paths (with lower per-flow bandwidth and higher latency) and expand on fast paths. This *elasticity* property allows flowlets to compensate for poor load balancing decisions by shifting traffic to less-congested paths automatically.

We demonstrate the resilience of flowlets to poor load balancing decisions with a simple experiment. Two leaf switches, $L0$-$L1$, are connected via two spine switches, $S0$-$S1$, with the path through $S1$ having half the capacity of the path through $S0$ (see topology in Figure 1b). Leaf $L0$ sends traffic — consisting of a realistic mix of short and large flows (§5) — to leaf $L1$ at an average rate of 112 Gbps (over 90% of the available 120 Gbps capacity).

We compare weighted-random load balancing on a per-flow, per-packet, and per-flowlet basis, as in the previous section. In this topology, ideally, the load balancer should pick the $S0$-path $2/3$rd of the time. To model inaccurate decisions, we vary the weight (probability) of the $S0$-path from 2% to 98% in a series of simulations. At each weight, we plot the overall average flow completion time (FCT) for the three schemes, normalized to the lowest value achieved across all experiments (with per-packet load balancing using a weight of 66% for $S0$).

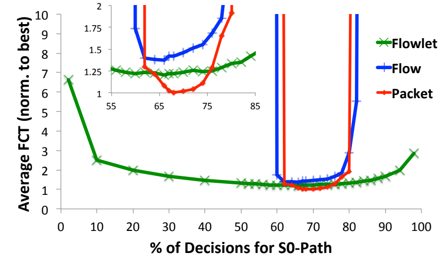


Figure 3: Flowlet-based load balancing is resilient to inaccurate load balancing decisions.

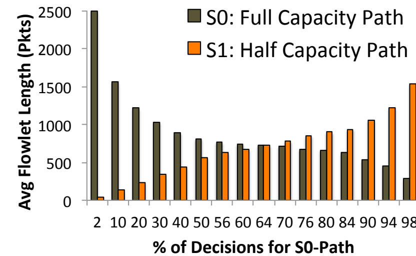


Figure 4: The flowlet sizes on the two paths change depending on how flowlets are split between them.

Figure 3 shows the results. We observe that flow- and packet-based load balancing deteriorates significantly outside a narrow range of weights near the ideal point. This is not surprising, since in these schemes, the amount of traffic on each path is directly determined by the weight chosen by the load balancer. If the traffic on either path exceeds its capacity, performance rapidly degrades.

Load balancing with flowlets, however, is robust over a wide range of weights: it is within $2\times$ of optimal for all weights between 20–95% for $S0$. The reason is explained by Figure 4, which shows how the average flowlet size on the two paths changes based on the weights chosen by the load balancer. If the load balancer uses the correct weights (66% for $S0$, 33% for $S1$) then the flowlet sizes on the two paths are roughly the same. But if the weights are incorrect, the flowlet sizes adapt to keep the actual traffic balanced. For example, even if only 2% of flowlets are sent via $S0$, the flowlets on the $S0$ path grow $\sim 65\times$ larger than those on the $S1$ path, to keep the traffic reasonably balanced at a 57% to 43% split.

This experiment suggests that a simple scheme that spreads flowlets on different paths in a traffic-oblivious manner can be very effective. In essence, due to their elasticity, flowlets implicitly reflect path traffic conditions (we analyze precisely why flowlets are elastic in §4). LetFlow takes advantage of this property to achieve good performance without explicit information or complex feedback mechanisms.

[2] Our implementation of per-packet load balancing includes an ideal reordering buffer at the receiver to prevent a performance penalty for TCP. See §5 for details.

3 Design and Hardware Implementation

LetFlow is very simple to implement. All functionality resides at the switches. The switches pick an outgoing port at random among available choices (given by the routing protocol) for each flowlet. The decisions are made on the first packet of each flowlet; subsequent packets follow the same uplink as long as the flowlet remains active (there is not a sufficiently long gap).

Flowlet detection. The switch uses a *Flowlet Table* to detect flowlets. Each entry in the table consists of a *port number*, a *valid bit* and an *age bit*. When a packet arrives, its 5-tuple header is hashed into an index for accessing an entry in the flowlet table. If the entry is active, i.e. the valid bit is set to one, the packet is sent to the port stored in the entry, and the age bit is cleared to zero. If the entry is not valid, the load balancer randomly picks a port from the available choices to forward the packet. It then sets the valid bit to one, clears the age bit, and records the chosen port in the table.

At a fixed time interval, Δ, a separate checker process examines each entry's age bit. If the age bit is not set, the checker sets the age bit to one. If the age bit is already set, the checker ages out the entry by clearing the valid bit. Any subsequent packet that is hashed to the entry will be detected as a new flowlet. This one-bit aging scheme, which was also used by CONGA [3], enables detection of flowlets without maintaining timestamps for each flow. The tradeoff is that the flowlet timeout can take any value between Δ and 2Δ. More precision can be attained by using more age bits per entry, but we find that a single bit suffices for good performance in LetFlow.

The timeout interval, Δ, plays a key role: setting it too low would risk reordering issues, but if set too high, it can reduce flowlet opportunities. We analyze the role of the flowlet timeout in LetFlow's effectiveness in §4.

Load balancing decisions. As previously discussed, LetFlow simply picks a port at random from all available ports for each new flowlet.

Hardware costs. The implementation cost (in terms of die area) is mainly for the Flowlet Table, which requires roughly 150K gates and 3 Mbits of memory, for a table with 64K entries. The area consumption is negligible ($< 0.3\%$) for a modern switching chip. LetFlow has been implemented in silicon for a major datacenter switch product line.

4 Analysis

We have seen that flowlets are resilient to inaccurate load balancing decisions because of their elasticity. In this section, we dig deeper to understand the reasons for the elasticity of flowlets (§4.1). We find that this is rooted

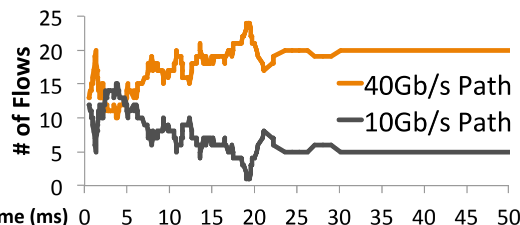


Figure 5: LetFlow balances the average rate of (long-lived) flows on asymmetric paths. In this case, 5 flows are sent to the 10 Gbps link; 20 flows to the 40 Gbps link.

in the fact that higher layer congestion control protocols like TCP adjust the rate of flows based on the available capacity on their path. Using these insights, we develop a Markov Chain model (§4.2) that shows how LetFlow balances load, and the role that the key parameter, the flowlet timeout, plays in LetFlow's effectiveness.

4.1 Why are Flowlets elastic?

To see why flowlets are elastic, let us consider what causes flowlets in the first place. Flowlets are caused by the burstiness of TCP at sub-RTT timescale [20]. TCP tends to transmit a window of packets in one or a few clustered bursts, interspersed with idle periods during an RTT. This behavior is caused by various factors such as slow-start, ACK compression, and packet drops.

A simple explanation for why flowlet sizes change with path conditions (e.g., shrinking on congested paths) points the finger at *packet drops*. The argument goes: on congested path, more packets are dropped; each drop causes TCP to cut its window size in half, thereby idling for at least half an RTT [5] and (likely) causing a flowlet.

While this argument is valid (and easy to observe empirically), we find that flowlets are elastic for a more basic reason, rooted in the way congestion control protocols like TCP adapt to conditions on a flow's path. When a flow is sent to a slower (congested) path, the congestion control protocol reduces its rate.[3] Because the flow slows down, there is a higher chance that none of its packets arrive within the timeout period, causing the flowlet to end. On the other hand, a flow on a fast path (with higher rate) has a higher chance of having packets arrive within the timeout period, reducing flowlet opportunities.

To illustrate this point, we construct the following simulation: 25 long-lived TCP flows send traffic on two paths with bottleneck link speeds of 40 Gbps and 10 Gbps respectively. We cap the window size of the flows to 256 KB, and set the buffer size on both links to be large enough such that there are no packet drops. The flowlet timeout is set to 200 μs and the RTT is 50 μs (in absence of the queueing delay).

Figure 5 shows how the number of flows on each path

[3] This may be due to packet drops or other congestion signals like ECN marks or increased delay.

changes over time. We make two observations. First, changes to the flows' paths occur mostly in the first 30 ms; after this point, flowlet timeouts become rare and the flows reach an *equilibrium*. Second, at this equilibrium, the split of flows on the two paths is ideal: 20 flows on the 40 Gbps path, and 5 flows on 10 Gbps path. This results in an average rate of 2 Gbps per flow on each path, which is the largest possible.

The equilibrium is stable in this scenario because, in any other state, the flows on one path will have a smaller rate (on average) than the flows on the other. These flows are more likely to experience a flowlet timeout than their counterparts. Thus, the system has a natural "stochastic drift", pushing it towards the optimal equilibrium state. Notice that a stochastic drift does not guarantee that the system remains in the optimal state. In principle, a flow could change paths if a flowlet timeout occurs. But this does not happen in this scenario due to TCP's ACK-clocked transmissions. Specifically, since the TCP window sizes are fixed, ACK-clocking results in a repeating pattern of packet transmissions. Therefore, once flows reach a state where there are no flowlet timeouts for one RTT, they are locked into that state forever.

In practice, there will be far more dynamism in the cross traffic and TCP's packet transmissions and inter-packet gaps. In general, the inter-packet gap for TCP depends on the window size, RTT, and degree of intra-RTT burstiness. The precise characterization of TCP inter-packet gaps is a difficult problem. But it is clear that inter-packet gaps increase as the flow's rate decreases and the RTT increases.

In search of a simpler, non-TCP-specific model, we next analyze a simple Poisson packet arrival process that provides insight into how LetFlow balances load.

4.2 A Markov chain model

Consider n flows that transmit on two paths P_1 and P_2, with bottleneck capacity C_1 and C_2 respectively.[4] Assume packet arrivals from flow i occur as a Poisson process with rate λ_i, independent of all other flows. Poisson arrivals is not realistic for TCP traffic (which is bursty) but allows us to capture the essence of the system's dynamics while keeping the model tractable (see below for more on the limitations of the model).

Assume the flowlet timeout is given by Δ, and that each new flowlet is mapped to one of the two paths at random. Further, assume that the flows on the same path achieve an equal share of the path's capacity; i.e.,

$$\lambda_i = \begin{cases} C_1/n_1 & \text{flow } i \text{ is on path } P_1 \\ C_2/n_2 & \text{flow } i \text{ is on path } P_2 \end{cases}$$

[4]For ease of exposition, we describe the model for two paths; the general case is similar.

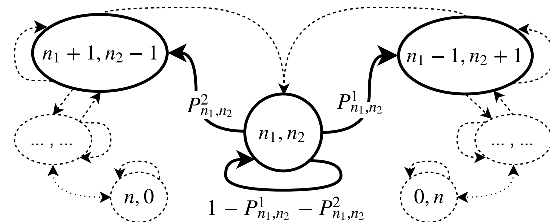


Figure 6: Markov chain model. The state $(n1, n2)$ gives the number of flows on the two paths.

where n_1 and n_2 denote the number of flows on each path. This is an idealization of the fair bandwidth allocation provided by congestion control protocols. Finally, let $\lambda_a = C_1 + C_2$ be the aggregate arrival rate on both paths.

It is not difficult to show that (n_1, n_2) forms a Markov chain. At each state, (n_1, n_2), if the next packet to arrive triggers a new flowlet for its flow, the load balancer may pick a different path for that flow (at random), causing a transition to state $(n_1 - 1,\, n_2 + 1)$ or $(n_1 + 1,\, n_2 - 1)$. If the next packet does not start a new flowlet, the flow remains on the same path and the state does not change.

Let $P^1_{n_1,n_2}$ and $P^2_{n_1,n_2}$ be the transition probabilities from (n_1, n_2) to $(n_1 - 1,\, n_2 + 1)$ and $(n_1 + 1,\, n_2 - 1)$ respectively, as shown in Figure 6. In Appendix A, we derive the following expression for the transition probabilities:

$$P^1_{n_1,n_2} = \frac{1}{2} \sum_{i \in P_1} \left[\frac{\lambda_i}{\lambda_a - \lambda_i} \left(e^{-\lambda_i \Delta} - e^{-\lambda_a \Delta} \right) + \frac{\lambda_i}{\lambda_a} e^{-\lambda_a \Delta} \right] \tag{1}$$

The expression for $P^2_{n_1,n_2}$ is similar, with the sum taken over the flows on path P_2. The transition probabilities can be further approximated as

$$P^j_{n_1,n_2} \approx \frac{C_j}{2(C_1 + C_2)} e^{-(C_j/n_j)\Delta} \tag{2}$$

for $j \in \{0, 1\}$. The approximation is accurate when $\lambda_a \gg \lambda_i$, which is the case if the number of flows is large, and (n_1, n_2) isn't too imbalanced.

Comparing $P^1_{n_1,n_2}$ and $P^2_{n_1,n_2}$ in light of Eq. (2) gives an interesting insight into how flows transition between paths. For a suitably chosen value of Δ, the flows on the path with lower per-flow rate are more likely to change path. This behavior pushes the system towards states where $C_1/n_1 \approx C_2/n_2$,[5] implying good load balancing. Notice that this is exactly the behavior we observed in the previous simulation with TCP flows (Figure 5).

The role of the flowlet timeout (Δ). Achieving the good load balancing behavior above depends on Δ. If Δ is very large, then $P^j_{n_1,n_2} \approx 0$ (no timeouts); if Δ is very small, then $P^j_{n_1,n_2} \approx C_1/2(C_1 + C_2)$. In either case, the transition probabilities don't depend on the rate of the flows and thus the load balancing feature of flowlet switching

[5]The most precise balance condition is $\frac{C_1}{n_1} - \frac{C_2}{n_2} \approx \frac{1}{\Delta} \log(\frac{C_1}{C_2})$.

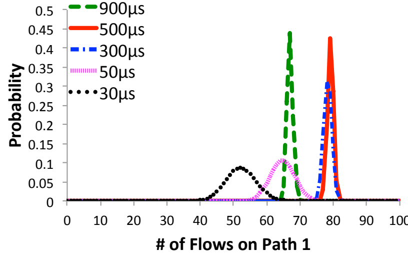


Figure 7: State probability distribution after 10^{11} steps, starting from (50,50), for different values of flowlet timeout.

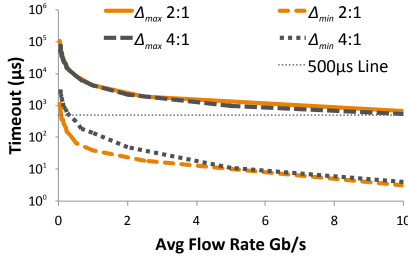


Figure 8: Flowlet timeout range vs. average flow rate.

is lost. This behavior is illustrated in Figure 7, which shows the probability distribution of the number of flows on path 1 for several values of Δ, with $n = 100$ flows, $C_1 = 40$ Gbps, and $C_2 = 10$ Gbps. These plots are obtained by solving for the probability distribution of the Markov chain after 10^{11} steps numerically, starting with an initial state of $(50,50)$.

The ideal operating point in this case is $(80,20)$. The plot shows that for small Δ (30 μs and 50 μs) and large Δ (900 μs), the state distribution is far from ideal. However, for moderate values of Δ (300 μs and 500 μs), the distribution concentrates around the ideal point.

So how should Δ be chosen in practice? This question turns out to be tricky to answer based purely on this model. First, the Poisson assumption does not capture the RTT-dependence and burstiness of TCP flows, both of which impact the inter-packet gap, hence, flowlet timeouts. Second, in picking the flowlet timeout, we must take care to avoid packet reordering (unless other mitigating measures are taken to prevent the adverse effects of reordering [11]).

Nonetheless, the model provides some useful insights for setting Δ. This requires a small tweak to (roughly) model burstiness: we assume that the flows transmit in bursts of b packets, as a Poisson process of rate λ_i/b. Empirically, we find that TCP sends in bursts of roughly $b = 10$ in our simulation setup (§5). Using this value, we run a series of simulations to obtain Δ_{min} and Δ_{max}, the minimum and maximum values of Δ for which the load balancer achieves a near-ideal split of flows across the two paths. In these simulations, we vary the number of flows, and consider two cases with 20/10 Gbps and 40/10 Gbps paths.

Figure 8 shows Δ_{min} and Δ_{max} as a function of the av-

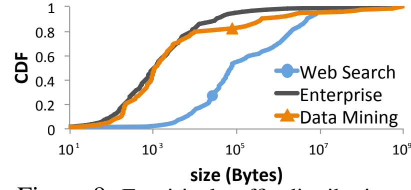


Figure 9: Empirical traffic distributions.

erage flow rate in each scenario: $(C_1 + C_2)/n$. There is a clear correlation between the average flow rate and Δ. As the average flow rate increases, Δ needs to be reduced to achieving good load balancing. A flowlet timeout around 500 μs supports the largest range of flow rates. We adopt this value in our experiments (§5).

5 Evaluation

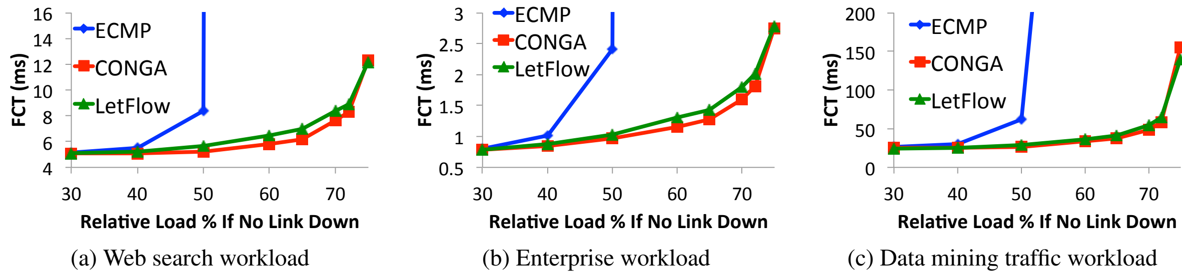
In this section, we evaluate LetFlow's performance with a small hardware testbed (§5.1) as well as large-scale simulations (§5.2). We also study LetFlow's generality and robustness (§5.3).

Schemes compared. In our hardware testbed, we compare LetFlow against ECMP and CONGA [3], a state-of-the-art adaptive load balancing mechanism. In simulations, we compare LetFlow against WCMP [31], CONGA, and Presto*, an idealized variant of Presto [15], a state-of-the-art proactive load balancing scheme that sprays small fixed-sized chunks of data across different paths and uses a reordering buffer at the receivers to put packets of each flow back in order. Presto* employs per-packet load balancing and a perfect reordering buffer that knows exactly which packets have been dropped, and which have been received out-of-order.

Note: Presto* cannot be realized in practice, but it provides the best-case performance for Presto (and other such schemes) and allows us to isolate performance issues due to load balancing from those caused by packet reordering. CONGA has been shown to perform better than MPTCP in scenarios very similar to those that we consider [3]; therefore, we do not compare with MPTCP.

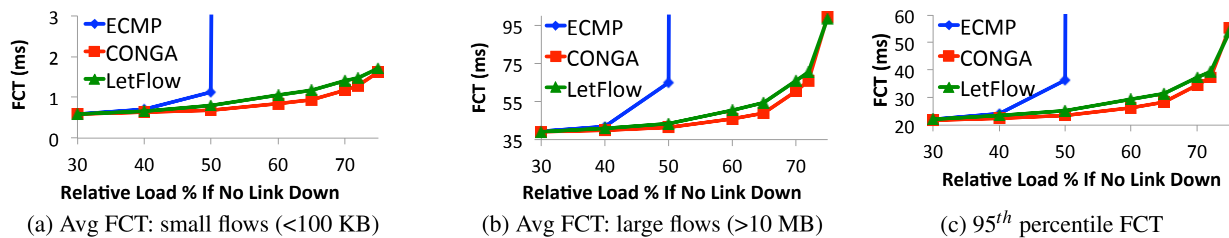
Workloads. We conduct our experiments using realistic workloads based on empirically observed traffic patterns in deployed data centers. We consider the three flow size distributions shown in Figure 9, from (1) production clusters for web search services [4]; (2) a typical large enterprise datacenter [3]; (3) a large cluster dedicated to data mining services [13]. All three distributions are heavy-tailed: a small fraction of the flows contribute most of the traffic. Refer to the papers [4, 3, 13] that report on these distributions for more details.

Metrics. Similar to prior work, we use flow completion time (FCT) as the primary performance metric. In certain experiments, we also consider packet latency and traffic distribution across paths.



(a) Web search workload (b) Enterprise workload (c) Data mining traffic workload

Figure 10: Testbed results: overall average FCT for different workloads on baseline topology with link failure (Fig. 1b).



(a) Avg FCT: small flows (<100 KB) (b) Avg FCT: large flows (>10 MB) (c) 95th percentile FCT

Figure 11: Several FCT statistics for web search workload on baseline topology with link failure.

Summary of results. Our evaluation spans several dimensions: (1) different topologies with varying asymmetry and traffic matrices; (2) different workloads; (3) different congestion control protocols with varying degrees of burstiness; (4) and different network and algorithm settings. We find that:

1. LetFlow achieves very good performance for realistic datacenter workloads in both asymmetric and symmetric topologies. It achieves average FCTs that are significantly better than competing traffic-oblivious schemes such as WCMP and Presto, and only slightly higher than CONGA: within 10-20% in testbed experiments and 2× in simulations with high asymmetry and traffic load.
2. LetFlow is effective in large topologies with high degrees of asymmetry and multi-tier asymmetric topologies, and also handles asymmetries caused by imbalanced and dynamic traffic matrices.
3. LetFlow performs consistently for different transport protocols, including "smooth" congestion control algorithms such as DCTCP [4] and schemes with hardware pacing such as DCQCN [32].
4. LetFlow is also robust across a wide range of buffer size and flowlet timeout settings, though tuning the flowlet timeout can improve its performance.

5.1 Testbed Experiments

Our testbed setup uses a topology that consists of two leaf switches and two spine switches as shown in Figure 1b. Each leaf switch is connected to each spine switch with two 40 Gbps links. We fail one of the links between a leaf and spine switch to create asymmetry, as indicated by the dotted line in Figure 1b. There are 64 servers that are attached to the leaf switches (32 per leaf) with 10 Gbps links. Hence, we have a 2:1 over-subscription at the leaf level, which is typical in modern data centers. The servers have 12 core Intel Xeon X5670 2.93 Ghz CPUs and 96 GB of RAM.

We use a simple client-server program to generate traffic. Each server requests flows according to a Poisson process from randomly chosen servers. The flow size is drawn from one of the three distributions discussed above. All 64 nodes run both client and server processes. To stress the fabric's load balancing, we configure the clients under each leaf to only send to servers under the other leaf, so that all traffic traverses the fabric.

Figure 10 compares the overall average FCT achieved by LetFlow, CONGA and ECMP for the three workloads at different levels of load. We also show a breakdown of the average FCT for small (<100 KB) and large (>10 MB) flows as well as the 95th percentile FCT for the web search workload in Figure 11. (The results for the other workloads are similar.)

In these experiments, we vary the traffic load by changing the arrival rate of flows to achieve different levels of load. The load is shown relative to the bisectional bandwidth without the link failure; i.e., the maximum load that the fabric can support with the link failure is 75% (see Figure 1b). Each data point is the average of ten runs on the testbed.

The results show that ECMP's performance deteriorates quickly as the traffic load increases to 50%. This is not surprising; since ECMP sends half of the traffic through each spine, at 50% load, the $S1 \rightarrow L1$ link (on the path with reduced capacity) is saturated and performance rapidly degrades. LetFlow, on the other hand, performs very well across all workloads and different flow size groups (Figure 11), achieving FCTs that are only 10-20% higher than CONGA in the worst case. This demonstrates the ability of LetFlow to shift traffic

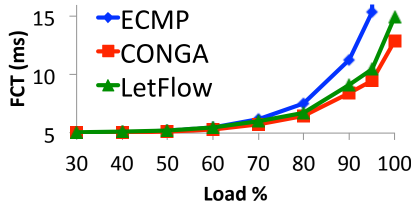


Figure 12: Overall average FCT for web search workload on symmetric testbed experiments.

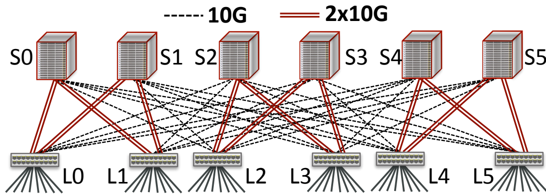


Figure 13: Large-scale topology with high path asymmetry.

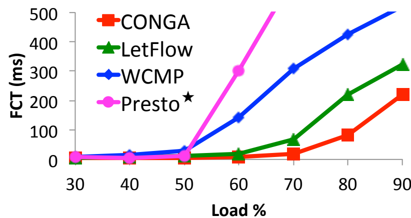


Figure 14: Average FCT comparison (web search workload, large-scale asymmetric topology).

away from the congested link for realistic dynamic traffic loads, without any explicit knowledge of path traffic conditions or complex feedback mechanisms.

Symmetric topology. We repeat these experiments in a symmetric topology without link failures. Figure 12 shows the results for the web search traffic workload. As expected, all three schemes perform similarly when the load is low, but CONGA and LetFlow both outperform ECMP at higher levels of load. Despite its simplicity, LetFlow is within 10% of CONGA at all levels of loads.

5.2 Large Scale Simulations

In this section, we evaluate LetFlow's performance via simulations in larger scale topologies with high degrees of asymmetry and multi-tier asymmetric topologies. We also validate LetFlow's ability to handle challenging traffic scenarios which require different load balancing decisions for traffic destined to different destinations. Our simulations use OMNET++ [28] and the Network Simulation Cradle [17] to port the actual Linux TCP source code (from kernel 2.6.26) as our simulator.

Large-scale topology with high asymmetry. In order to evaluate LetFlow in more complex topologies, we simulate an asymmetric topology with 6 spines and 6 leaf switches, shown in Figure 13. This topology is from the WCMP paper [31]. Spine and leaf switches that belong to the same pod are connected using two 10 Gbps links while inter-pod connections use a single 10 Gbps link. Such asymmetric topologies can occur as a result of *imbalanced striping* — a mismatch between switch radices at different levels in a Clos topology [31].

Each leaf is connected to 48 servers. We generate web search traffic uniformly at random from servers at each leaf to servers at the other five leaves. We compare the performance of LetFlow against Presto*, WCMP and CONGA. Presto* and WCMP take into account the path asymmetry by using static weights (based on the topology) to make load balancing decisions, as described in the WCMP [31] and Presto [15] papers.

Figure 14 shows that WCMP and Presto* fail to achieve the performance of CONGA and LetFlow. In this scenario, Presto* cannot finish flows quickly enough and is unstable at 80% and 90% load. CONGA has the best performance overall. But LetFlow also performs quite well: it achieves average FCTs within 50% of CONGA, even at 90% load. This result shows that static topology-dependent weights are inadequate for handling asymmetry with dynamic workloads, and LetFlow is able to balance load well in complex asymmetric topologies.

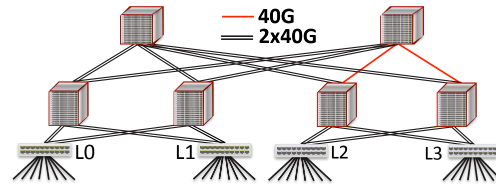


Figure 15: Multi-tier topology with link failure.

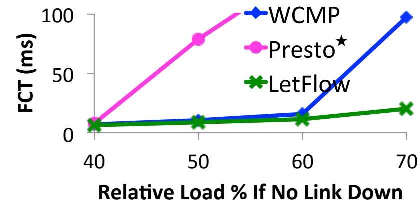
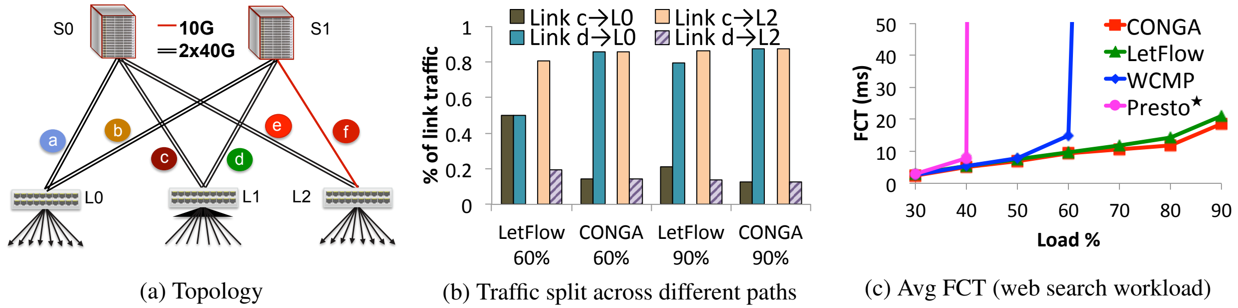


Figure 16: Average FCT (web search workload, asymmetric multi-tier topology).

Multi-tier topology. In order to evaluate LetFlow's scalability, we simulate a 3-tier topology with asymmetry as shown in Figure 15. We aim to validate that LetFlow is able to detect this asymmetry and react accordingly. We focus on the performance of traffic from $L0$-$L1$ to $L2$-$L3$.

We do not consider CONGA in this scenario since it was designed for two-tier topologies [3]. We compare LetFlow's performance against WCMP and Presto* for the web search workload. Figure 16 shows the average flow completion time at 40%-70% load. The results show that LetFlow handles asymmetry in the multi-tier topology very well. While Presto* is unstable when the load is larger than 50% and WCMP degrades severely when the load reaches 70%, LetFlow performs similarly to that in the two-tier network.



(a) Topology (b) Traffic split across different paths (c) Avg FCT (web search workload)

Figure 17: Multi-destination scenario with varying asymmetry.

	Destined to $L0$	**Destined to** $L2$
Traffic on *Link c*	11.1%	88.9%
Traffic on *Link d*	88.9%	11.1%

Table 2: Ideal traffic split for multi-destination scenario.

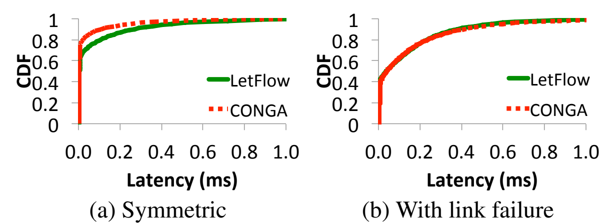
Multiple destinations with varying asymmetry. Load balancing in asymmetric topologies often requires splitting traffic to different destinations differently. For this reason, some load balancing schemes like CONGA [3] take into account the traffic's destination to make load balancing decisions. We now investigate whether LetFlow can handle such scenarios despite making random decisions that are independent of a packet's destination.

We simulate the topology shown in Figure 17a. One leaf switch, $L1$, sends traffic uniformly at random to the other two leaf switches, $L0$ and $L2$, using the web search workload. Note that the two paths from $L1$ to $L0$ are symmetric (both have 80 Gbps of capacity) while the paths between $L1$ to $L2$ are asymmetric: there is only one 10 Gbps link between $S1$ to $L2$. As a result, leaf $L1$ must send more $L1 \rightarrow L2$ traffic via spine $S0$ than $S1$. This in turn creates (dynamic) bandwidth asymmetry for the $L1 \rightarrow L0$ traffic.

The ideal traffic split to balance the maximum load across all fabric links is given in Table 2. We should split the traffic between *Link e* and *Link f* with a ratio of 8:1; i.e., 88.9% of the traffic destined to $L2$ should be carried on *Link c* while 11.1% should be carried on *Link d*. To counter the traffic imbalance on *Link c* and *Link d*, the traffic to $L0$ should ideally be split in the opposite ratio, 1:8, on *Link c* and *Link d*.

Can LetFlow achieve such an ideal behavior? Figure 17b shows the traffic split for LetFlow and CONGA when the total traffic load from $L1$ is 48 Gbps (60%) and 72 Gbps (90%) respectively. We omit ECMP results since it is unstable in this scenario.

We observe that when the traffic load is relatively low (60% for *Link c* and *Link d*), the traffic split with LetFlow for the $L1 \rightarrow L0$ traffic is simply 50%-50%, not ideal. However, notice that at this low load, *Link c* has spare capacity, even with a 50%-50% split for $L1 \rightarrow L0$ traffic. Thus, flowlets destined to $L0$ do not find either link to be particularly better than the other. For traffic destined to $L2$, however, the paths via *Link e* and *Link f* are vastly different. Thus LetFlow's flowlets automatically shift traffic away from the congested link, and the traffic split ratio is 81%/19%, close to ideal.



(a) Symmetric (b) With link failure

Figure 18: Network latency CDF with LetFlow and CONGA on the baseline topology (simulation).

Now when the load increases to 90%, evenly splitting traffic destined to $L0$ on *Link c* and *Link d* would cause *Link c* to be heavily congested. Interestingly, in this case, LetFlow avoids congestion and moves more $L1 \rightarrow L0$ traffic to *Link d* and achieves close to the ideal ratio.

LetFlow's good performance is also evident from the average FCT in this scenario, shown in Figure 17c. LetFlow achieves similar performance as CONGA (at most 10% worst), which achieves close-to-ideal traffic splits for both $L0$- and $L2$-bound traffic.

Impact on Latency. Finally, we compare LetFlow and CONGA in terms of their impact on fabric latency. We omit the results for ECMP and Presto* which perform much worse in asymmetric topologies, as to examine closely the differences between CONGA and LetFlow. We measure a packet's fabric latency by timestamping it when it enters the ingress leaf switch and when it enters the egress leaf switch. This allows us to measure the fabric latency without including the latency incurred at the egress switch, which is not influenced by load balancing.

Figure 18 shows the CDF of fabric latency for the web search workload at 60% average load in the topology shown in Figure 1b, with and without asymmetry. In the heavily congested asymmetric scenario, LetFlow has similar fabric latency to CONGA. In the symmetric topology, however, CONGA achieves lower fabric latency. The reason is that CONGA can detect subtle traffic imbalances on the two paths and proactively balance load by choosing the path that is least congested. LetFlow, however, is reactive and needs to cause congestion

(e.g. packet loss, increased delay) for TCP to "kick in" and the flowlet sizes to change. Hence, its reaction time is slower than CONGA. This distinction is less relevant in the asymmetric scenario where congestion is high and even CONGA can't avoid increasing queuing delay.

5.3 Robustness

We have shown that LetFlow can effectively load balance traffic under various conditions. In this section, we evaluate LetFlow's robustness to different factors, particularly, different transport protocols, flowlet time timeout periods, and buffer sizes.

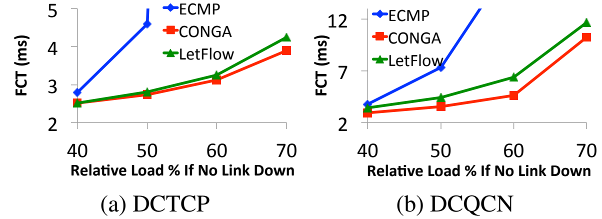
Different transport protocols. As explained in §4, LetFlow's behavior depends on the nature of inter-packet gaps. A natural question is thus how effective is LetFlow for other transport protocols that have different burstiness characteristics than TCP? To answer this question, we repeat the previous experiment for two new transports: DCTCP [4] and DCQCN [32].

DCTCP is interesting to consider because it adjusts its window size much more smoothly than TCP and largely avoids packet drops that can immediately cause flowlet timeouts. To use DCTCP, we enable ECN marking at the switches and set the marking threshold to be 100 KB. Since DCTCP reduces queuing delay, we also set the flowlet table timeout period to 50 μs. We discuss the impact of the flowlet timeout parameter further in the following section. Figure 19a compares the overall average flow completion time using CONGA and LetFlow (with DCTCP as the transport). Similar to the case with TCP, we observe that LetFlow is able to achieve performance within 10% of CONGA in all cases.

Next, we repeat the experiment for DCQCN [32], a recent end-to-end congestion control protocol designed for RDMA environments. DCQCN operates in the NICs to throttle flows that experience congestion via hardware rate-limiters. In addition, DCQCN employs Layer 2 Priority Flow Control (PFC) to ensure lossless operations.

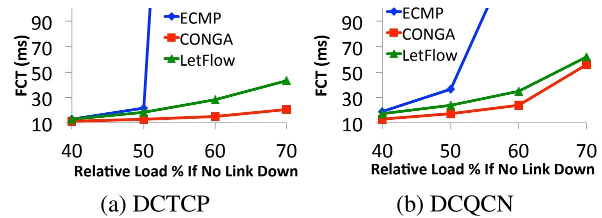
We use the same flowlet table timeout period of 50 μs as in DCTCP's simulation. Figure 19b shows that, while ECMP's performance becomes unstable at loads above 50%, both CONGA and LetFlow can maintain stability by moving enough traffic away from the congested path. LetFlow achieves an average FCT within 47% of CONGA. The larger gap relative to CONGA compared to TCP-based transports is not surprising since DCQCN generates very smooth, perfectly paced traffic, which reduces flowlet opportunities. Nonetheless, as our analysis (§4) predicts, LetFlow still balances load quite effectively, because DCQCN adapts the flow rates to traffic conditions on their paths.

At the 95th percentile, LetFlow's FCT is roughly within 2× of CONGA (and still much better than ECMP) for DCTCP, as shown in Figure 20a. The degradation for DCQCN is approximately 1.5× of CONGA (Figure 20b). This degradation at the tail is not surprising, since unlike CONGA which proactively balances load, LetFlow is reactive and its decisions are stochastic.



(a) DCTCP (b) DCQCN

Figure 19: Overall average FCT with different transport protocols (web search workload, baseline topology).



(a) DCTCP (b) DCQCN

Figure 20: 95th percentile FCT with different transport protocols (web search workload, baseline topology).

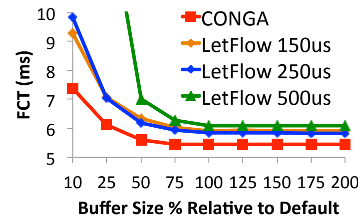


Figure 21: Effect of buffer size and flowlet timeout period on FCT (web search workload at 60% load, baseline topology).

Varying flowlet timeout. The timeout period of the flowlet table is fundamental to LetFlow: it determines whether flowlets occur frequently enough to balance traffic. The analysis in §4 provides a framework for choosing the correct flowlet timeout period (see Figure 8). In Figure 21, we show how FCT varies with different values of the flowlet timeout (150 μs, 250 μs and 500 μs) and different buffer sizes in the testbed experiment. These experiments use the web search workload at 60% load and TCP New Reno as the transport protocol. The buffer size is shown relatively to the maximum buffer size (10 MB) of the switch.

When the buffer size is not too small, different timeout periods do not change LetFlow's performance. Since all three values are within the minimum and maximum bounds in Figure 8, we expect all three to perform well. However, for smaller buffer sizes, the performance improves with lower values of the flowlet timeout. The reason is that smaller buffer sizes result in smaller RTTs, which makes TCP flows less bursty. Hence, the burst parameter, b, in §4.2 (set to 10 by default) needs to be

reduced for the model to be accurate.

Varying buffer size. We also evaluate LetFlow's performance for different buffer sizes. As shown in Figure 21, the FCT suffers when the buffer is very small (e.g., 10% of the max) and gradually improves as the buffer size increases to 75% of the maximum size. Beyond this point, the FCT is flat. LetFlow exhibits similar behavior to CONGA in terms of dependency on buffer size.

6 Related Work

We briefly discuss related work, particularly for datacenter networks, that has inspired and informed our design.

Motivated by the drawbacks of ECMP's flow-based load balancing [16], several papers propose more fine-grained mechanisms, including flowlets [20], spatial splitting based on TCP sequence numbers [24], per-packet round robin [9], and load balancing at the level of TCP Segmentation Offload (TSO) units [15]. Our results show that such schemes cannot handle asymmetry well without path congestion information.

Some schemes have proposed topology-dependent weighing of paths as a way of dealing with asymmetry. WCMP [31] adds weights to ECMP in commodity switches, while Presto [15] implements weights at the end-hosts. While weights can help in some scenarios, static weights are generally sub-optimal with asymmetry, particularly for dynamic traffic workloads. LetFlow improves resilience to asymmetry even without weights, but can also benefit from better weighing of paths based on the topology or coarse estimates of traffic demands (e.g., see experiment in Figure 2).

DeTail [30] propose per-packet adaptive load balancing, but requires priority flow control for hop-by-hop congestion feedback. Other dynamic load balancers like MPTCP [23], TeXCP [19], CONGA [3], and HULA [21] load balance traffic based on path-wise congestion metrics. LetFlow is significantly simpler compared to these designs as its decisions are local and purely at random.

The closest prior scheme to LetFlow is FlowBender [18]. Like LetFlow, FlowBender randomly reroutes flows, but relies on explicit path congestion signals such as per-flow ECN marks or TCP Retransmission Timeout (RTO). LetFlow does not need any explicit congestion signals or TCP-specific mechanisms like ECN, and can thus support different transport protocols more easily.

Centralized load balancing schemes such as Hedera [2] and MicroTE [8] tend to be slow [23] and also have difficulties handling traffic volatility. Fastpass [22] is a centralized arbiter which can achieve near-ideal load balancing by determining the transmission time and path for every packet, but scaling a centralized per-packet arbiter to large-scale datacenters is very challenging.

7 Final Remarks

Let the flowlets flow!

The main thesis of this paper is that flowlet switching is a powerful technique for simple, resilient load balancing in the presence of network asymmetry. The ability of flowlets to automatically change size based on real-time traffic conditions on their path enables them to effectively shift traffic away from congested paths, without the need for explicit path congestion information or complex feedback mechanisms.

We designed LetFlow as an extreme example of this approach. LetFlow simply picks a path uniformly at random for each flowlet, leaving it to the flowlets to do the rest. Through extensive evaluation in a real hardware testbed and large-scale simulations, we showed that LetFlow has comparable performance to CONGA [3], e.g., achieving average flow completion times within 10-20% of CONGA in our testbed experiments and 2× of CONGA in challenging simulated scenarios.

LetFlow is not an optimal solution or the only way to use flowlet switching. By its reactive and stochastic nature, LetFlow cannot prevent short-time-scale traffic imbalances that can increase queuing delay. Also, in symmetric topologies, schemes that balance load proactively at a more fine-grained level (e.g., packets or small-sized chunks [15]) would perform better.

LetFlow is, however, a significant improvement over ECMP and can be deployed today to greatly improve resilience to asymmetry. It is trivial to implement in hardware, does not require any changes to end-hosts, and is incrementally deployable. Even if only some switches use LetFlow (and others use ECMP or some other mechanism), flowlets can adjust to bandwidth asymmetries and improve performance for all traffic.

Finally, LetFlow is an instance of a more general approach to load balancing that randomly reroutes flows with a probability that decreases as a function of the flow's rate. Our results show that this simple approach works well in multi-rooted tree topologies. Modeling this approach for general topologies and formally analyzing its stability and convergence behavior is an interesting avenue for future work.

Acknowledgements

We are grateful to our shephard, Thomas Anderson, the anonymous NSDI reviewers, Akshay Narayan, and Srinivas Narayana for their valuable comments that greatly improved the clarity of the paper. We are also thankful to Edouard Bugnion, Peter Newman, Laura Sharpless, and Ramanan Vaidyanathan for many fruitful discussions. This work was funded in part by NSF grants CNS-1617702 and CNS-1563826, and a gift from the Cisco Research Center.

References

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

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[4] M. Alizadeh et al. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[5] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. In *SIGCOMM*, 2004.

[6] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *SIGCOMM*, 2010.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[8] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[9] J. Cao et al. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *CoNEXT*, 2013.

[10] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM*, 2013.

[11] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 20. ACM, 2016.

[12] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, 2011.

[13] A. Greenberg et al. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[15] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. 2015.

[16] C. Hopps. Analysis of an equal-cost multi-path algorithm, 2000.

[17] S. Jansen and A. McGregor. Performance, Validation and Testing with the Network Simulation Cradle. In *MASCOTS*, 2006.

[18] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 149–160. ACM, 2014.

[19] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[20] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic Load Balancing Without Packet Reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, Mar. 2007.

[21] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. ACM Symposium on SDN Research*, 2016.

[22] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 307–318. ACM, 2014.

[23] C. Raiciu et al. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM*, 2011.

[24] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, Optimal Flow Routing in Datacenters via Local Link Balancing. In *CoNEXT*, 2013.

[25] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hoelzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Sigcomm '15*, 2015.

[26] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[27] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[28] A. Varga et al. The OMNeT++ discrete event simulation system. In *ESM*, 2001.

[29] P. Wang, H. Xu, Z. Niu, D. Han, and Y. Xiong. Expeditus: Congestion-aware load balancing in clos data center networks. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 442–455, New York, NY, USA, 2016. ACM.

[30] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*, 2012.

[31] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. Wcmp: weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page 5. ACM, 2014.

[32] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. *SIGCOMM Comput. Commun. Rev.*, 45(4):523–536, Aug. 2015.

A Derivation of Equation (1)

Suppose n flows share a link, and flow i transmits packets as a Poisson process with a rate λ_i, independent of the other flows. Let $\lambda_a = \sum_{i=1}^{n} \lambda_i$ be the aggregate packet arrival rate. Notice that the packet arrival process for all flows besides flow i is Poisson with rate $\lambda_a - \lambda_i$.

Consider an arbitrary time instant t. Without loss of generality, assume that flow i is the next flow to incur a flowlet timeout after t. Let τ_i^{-1} be the time interval between flow i's last packet arrival and t, and τ_i be the time interval between t and flow i's next packet arrival. Also, let $\tau_{a\backslash i}$ be the time interval until the next packet arrival from any flow other than flow i. Figure 22 shows these quantities.

The probability that flow i incurs the next flowlet timeout, $\mathbb{P}_i$, is the joint probability that the next packet arrival after time t is from flow i, and its inter-packet gap, $\tau_i^{-1}+\tau_i$, is larger than the flowlet timeout period Δ:

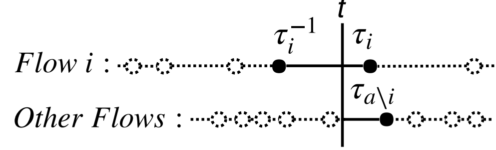


Figure 22: Packet arrival processes for flow i and other flows, and their relationship to each other

$$\begin{aligned}\mathbb{P}_i &= \mathbb{P}\left\{\tau_i < \tau_{a\backslash i}, \tau_i + \tau_i^{-1} > \Delta\right\} \\ &= \int_0^\infty \lambda_i e^{-\lambda_i t} \mathbb{P}\left\{\tau_{a\backslash i} > t, \tau_i^{-1} > \Delta - t\right\} dt. \quad (3)\end{aligned}$$

Since packet arrivals of different flows are independent, we have:

$$\mathbb{P}\left\{\tau_{a\backslash i} > t, \tau_i^{-1} > \Delta - t\right\} = \mathbb{P}\left\{\tau_{a\backslash i} > t\right\} \mathbb{P}\left\{\tau_i^{-1} > \Delta - t\right\}.$$

Also, it follows from standard properties of Poisson processes that

$$\begin{aligned}\mathbb{P}\left\{\tau_{a\backslash i} > t\right\} &= e^{-(\lambda_a - \lambda_i)t} \\ \mathbb{P}\left\{\tau_i^{-1} > \Delta - t\right\} &= \begin{cases} e^{-\lambda_i(\Delta - t)} & t \le \Delta \\ 1 & t \ge \Delta. \end{cases} \quad (4)\end{aligned}$$

Therefore, we obtain

$$\begin{aligned}\mathbb{P}_i &= \int_0^\Delta \lambda_i e^{-\lambda_i \Delta} e^{-(\lambda_a - \lambda_i)t} dt + \int_\Delta^\infty \lambda_i e^{-\lambda_a t} dt \\ &= \frac{\lambda_i}{\lambda_a - \lambda_i}\left(e^{-\lambda_i \Delta} - e^{-\lambda_a \Delta}\right) + \frac{\lambda_i}{\lambda_a} e^{-\lambda_a \Delta}. \quad (5)\end{aligned}$$

Now, assume that there are two paths P_1 and P_2, and let n_1 and n_2 denote the number of flows on each path. Also, assume that the total arrival rate on both paths is given by λ_a. Following a flowlet timeout, the flow with the timeout is assigned to a random path. It is not difficult to see that (n_1, n_2) forms a Markov chain (see Figure 6).

Let $P^1_{n_1,n_2}$ and $P^2_{n_1,n_2}$ be the transition probabilities from (n_1, n_2) to $(n_1 - 1, n_2 + 1)$ and $(n_1 + 1, n_2 - 1)$ respectively. To derive $P^1_{n_1,n_2}$, notice that the probability that the next flowlet timeout occurs for one of the flows on path P_1 is given by $\sum_{i \in P_1} \mathbb{P}_i$, where the notation $i \in P_1$ indicates that the sum is over the flows on path P_1, and $\mathbb{P}_i$ is given by the expression in Eq. (5). The flow that times out will change paths with probability $1/2$. Therefore:

$$P^1_{n_1,n_2} = \frac{1}{2} \sum_{i \in P_1} \left[\frac{\lambda_i}{\lambda_a - \lambda_i}\left(e^{-\lambda_i \Delta} - e^{-\lambda_a \Delta}\right) + \frac{\lambda_i}{\lambda_a} e^{-\lambda_a \Delta}\right], \quad (6)$$

which is Equation (1) in §4. The derivation for $P^2_{n_1,n_2}$ is similar.