Balancing on the edge: transport affinity without network state

João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek and Raul Landa
Fastly

Abstract

Content delivery networks and edge peering facilities have unique operating constraints which require novel approaches to load balancing. Contrary to traditional, centralized datacenter networks, physical space is heavily constrained. This limitation drives both the need for greater efficiency, maximizing the ability to absorb denial of service attacks and flash crowds at the edge, and seamless failover, minimizing the impact of maintenance on service availability.

This paper introduces *Faild*, a distributed load balancer which runs on commodity hardware and achieves graceful failover without relying on network state, providing a cost-effective and scalable alternative to existing proposals. Faild allows any individual component of the edge network to be removed from service without breaking existing connections, a property which has proved instrumental in sustaining the growth of a large global edge network over the past four years. As a consequence of this operational experience, we further document unexpected protocol interactions stemming from misconfigured devices in the wild which have significant ramifications for transport protocol design.

1 Introduction

While economies of scale have increasingly centralized compute and storage, user expectations dictate that content and even logic be pushed further towards the edge of the network. This centrifugal relationship has increased the relevance of Points of Presence (POPs) as an intermediary between cloud hosted services and end-users.

As a design pattern, POPs were pioneered by Content Delivery Networks (CDNs) intent on improving performance for traditional HTTP caching and DNS resolution. They have since evolved to encompass a much wider set of application layer services such as media processing (*e.g.* video and image optimization), security (*e.g.* application layer firewalls) and business logic (*e.g.* authentication and authorization; paywalls; request routing). The common thread uniting these *edge cloud* services is that they are latency sensitive and must therefore be geographically distributed across POPs rather than merely centralized within availability regions.

Today, edge cloud providers deploy hundreds of POPs [31, 39], with individual POPs able to deliver upwards of a terabit per second of bandwidth whilst handling millions of requests per second. How traffic is distributed across available hosts within a POP has a significant impact on the performance and availability of a large number of Internet services. Load balancing in this context differs significantly from traditional datacenter environments, and as a result existing solutions [15, 16, 17, 22, 27] are not readily applicable.

The defining constraint in the architecture of a POP is that physical space is at a premium [39]. POPs are typically set up in colocation facilities, often in regions where few alternatives exist, if any. The resulting inelastic price dynamics impose a strong economic incentive to minimize POP hardware in an effort to reduce capital expenditure. Load balancing under such constraints exacerbates the following concerns:

Efficiency. The physical build of POPs must be designed to maximize the number of service requests that a given number of hosts can process. Proposals that rely on dedicated hardware appliances or VM instances [15, 16, 17, 27] are not cost-efficient, since they consume scarce resources without increasing the number of requests that a POP can service.

Resilience. POPs provide a critical service at a fixed capacity, and are therefore attractive targets for denial-of-service attacks. Load balancing proposals which rely on flow state or incur significant per-flow overhead [15, 16, 17, 22, 27] are vulnerable to exhaustion attacks, and pose a threat to business continuity.

Gracefulness. Owing to the higher processing density of POPs, individual components within a POP represent a much larger proportion of total system capacity

when compared with traditional cloud environments. It is therefore vital that all system components can be brought in and out of service gracefully, without disrupting active flows. To our knowledge, no existing load balancer ensures seamless addition and removal of every element of its architecture, including hosts, switches and peers.

The primary contribution of this paper is to describe the design and implementation of a load-balancer which achieves all of the above goals. *Faild* provides *transport affinity* (seamless transport-layer failover) with minimal processing overhead. It synthesizes two distinct approaches leveraging hardware processing on commodity switches where possible, and pushing out flow handling to efficient software implementations when necessary.

A load balancing solution operating further down the network stack would be oblivious to how packets relate to ongoing transport connections, and therefore unable to ensure *graceful draining*, *i.e.* bringing a system component out of production without affecting service traffic. On the other hand, operating above the transport layer requires per-session state tracking, which is at odds with our requirement for robustness against resource exhaustion attacks. A key insight in this paper is that since end hosts track flows themselves, this trade-off is not binding: by performing host mapping on switches and re-purposing the connection tracking functions that end-hosts already perform, a load-balancer can retain transport affinity without the burden of maintaining network state. Although many of the specific techniques used by Faild have been used in isolation in other contexts (see Sec. 7), Faild brings them together in a novel way that directly addresses the needs of edge clouds and CDNs.

Our final contribution is to provide insight into the trials and tribulations of operating Faild over the past four years at a large[1] edge cloud provider. We document where reality fell short of our original design assumptions - from network equipment shortcomings to unexpected middlebox behavior - and how these quirks may affect future protocols to come.

The remainder of this paper is organized as follows. Sec. 2 expands upon the engineering requirements of edge load balancing. Sec. 3 describes the design of Faild, and Sec. 4 details its current implementation. Sec. 5 evaluates the system in practice, and Sec. 6 describes some of the lessons we learned by developing it and deploying it on production. We present relevant related work in Sec. 7, and conclude with Sec. 8.

2 Background and motivation

While the architecture of a POP bears superficial resemblance to traditional datacenter environments, their goals differ, and as a result the set of constraints they impose diverge. This section expands upon the requirements presented in Sec. 1 and highlights the idiosyncrasies of load balancing within POPs.

High request processing density. Ideally, all available power and space in the POP would be devoted exclusively to *hosts*. This proves unfeasible except for the very smallest POPs. As the number of hosts in the POP increases, directly connecting them to upstream providers becomes 1) logistically impractical for providers; 2) prohibitively expensive for intra-POP traffic; and eventually 3) physically impossible due to lack of ports. Network devices are therefore required to both reduce the number of interconnects required towards providers and ensure connectivity between hosts[2]. Our preferred *POP topology* maximizes power and space cost-benefit for hosts by collapsing all load balancing functions into just two layers: one for switches and one for hosts. A POP hence consists of a minimal number of switches, each one connected to all hosts and providing them with consolidated BGP sessions and Internet connectivity. This deviates from a common design pattern in datacenters, which relies on Clos-like topologies [6, 33].

Traditional hardware solutions *e.g.* [1, 3] are undesirable both due to their low power/space efficiency and poor horizontal scalability. Software-based solutions [15, 27] on the other hand perform poorly given the lower throughput and higher latency of packet processing on general purpose hardware. For example, Maglev [15] claims a 10 Gbps limit per load balancing host. Our most common POP build has 4 switches fully meshed with 32 hosts using 25 Gbps NICs. Since each host has a rated capacity of 40 Gbps in order to ensure operational stability, this results in a reference POP throughput of 1.28 Tbps. This alone would require over 100 Maglev hosts to load balance, which greatly exceeds the number of target hosts. Even if future advancements made it possible to substantially reduce the number of software load balancers required per target host, deploying load balancing functions in hosts would prevent using the full bisection bandwidth offered by the physical POP topology, further impacting request processing density.

Traffic surges. POP load balancers must be designed to gracefully withstand traffic surges of hundreds of times their usual loads, as well as DDoS attacks.

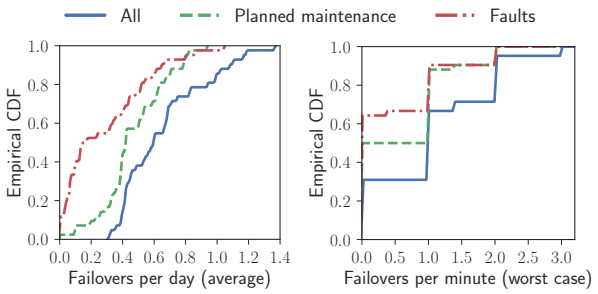
Highly optimized stateful solutions such as SilkRoad [22] can scale to ten million simultaneous connections by using hashing to scale per-connection state management and storing this state in the ASIC SRAM memory of switches with a programmable data plane. In our experience however POPs are routinely subjected to SYN floods largely exceeding this number, and the intensity and frequency of these attacks is suppressed only

[1] 40+ POPs globally; 7+ million RPS and 4+ Tbps of client traffic.

[2] Facebook EdgeFabric [31] and Google Espresso [39] rely on similar architectures to provide a capacity-aware, SDN-based *edge peering*.

when they fail to inflict economic harm. While software-based solutions like Maglev [15] and Duet [17] are less memory-constrained than SilkRoad, they are still subject to degraded performance as the connection count rises.

A commonly deployed alternative to stateful load balancing is to rely on Equal Cost Multipath (ECMP) [18, 38] to simply hash inbound connections to active hosts. While ECMP results in connection resets when re-hashing, many operators tolerate this given they more often operate under high load than high churn. Flashcrowds and DDoS in particular make statelessness an engineering necessity for edge load balancing, rather than a design choice.



(a) Average host drain event rate per POP per day

(b) Worst case host drain event rate per POP per minute

Figure 1: Drain events for one month, for all POPs

Host churn. While horizontally scaled services in a datacenter may comprise tens or hundreds of thousands of instances, POPs will typically have tens of nodes. As a result, the relative service impact of removing and adding hosts is much greater. Similarly, POPs are directly exposed to a much more diverse set of peers (via IXPs, Internet exchange points).

Without support for graceful failover, churn across the set of hosts and providers can disrupt traffic in a manner which is observable by customers, consequently increasing the operational expense for support and deployment. The need to remove hosts for software upgrades in particular is further exacerbated in edge networks due to their pivotal role in securing cloud services. Often edge networks will provide TLS termination and interface with a much wider set of clients, which increases the churn due to important software upgrades. Fig. 1 shows the average and worst case rate for hosts being drained across POPs over the course of a month. The average daily drain rate is not neglible given the size of a POP - this will include both planned, automated maintenances and software or hardware faults. Multiple concurrent drain events, however are relatively rare, since automated upgrades will be halted in the presence of unplanned events. In our environment switches are almost as frequently upgraded as hosts, given we run a custom control plane stack and there are frequent adjustments to the set of BGP peers.

Software load balancers [15, 16, 17, 27] can gracefully drain service hosts, but not the load balancers themselves since they maintain per-flow state. Hence, it is normally not possible to remove a load balancer instance from service without disrupting ongoing connections unless state is synchronized across all instances, which in itself is non-trivial and hence rarely supported. Furthermore, routing changes within a POP or peer churn may cause flows to be hashed to different load balancer instances, thereby disrupting existing flows.

3 Design

Building on the stated goals from previous sections, we now turn our attention to designing an efficient, stateless and graceful load balancing system. *Faild* attains these goals by relying on the socket information that hosts are required to maintain, allowing network devices to remain oblivious to TCP connection state. Similarly to other load balancing architectures [15, 17, 19, 27], Faild uses ECMP to hash flow tuples and load balance *service* traffic. Correspondingly, services are application layer abstractions advertised to the Internet using sets of *virtual IP addresses (VIP sets)*. Sec. 3.1 explores how Faild couples ECMP and MAC address rewriting to approximate *consistent hashing*. We then describe how this can be leveraged to enable graceful host failover in Sec. 3.2 and discuss host-side packet processing in Sec. 3.3.

3.1 Consistent hashing

We define *consistent hashing* as the ability to change from load balancing over a *baseline* host set B to balancing over a *failover* host set F with only traffic destined to hosts in a *removal* host set R of hosts in B but not in F being affected. Only flows terminating on R (the set of hosts being *drained*) are affected; ongoing flows terminating on hosts in the *common* set C of hosts found in both B and F remain unaffected.

In Faild, switches implement platform agnostic consistent hashing by maintaining a *fixed* set of virtual nexthops, forcing the switch to perform an ARP lookup instead. By not manipulating the routing table, we avoid rehashing events which would otherwise reset existing connections. Faild maps each service (set of VIPs) to a set of ECMP nexthops, and each ECMP nexthop to a MAC address. It is this transitive association between services and MAC addresses that determines the load balancing configuration.

Draining a switch can be achieved by instructing it to withdraw route advertisements for all services from all BGP sessions with its upstream providers. This will redirect traffic flowing through it towards neighboring switches, which would still advertise the withdrawn prefixes. The only requirement for this process to be grace-

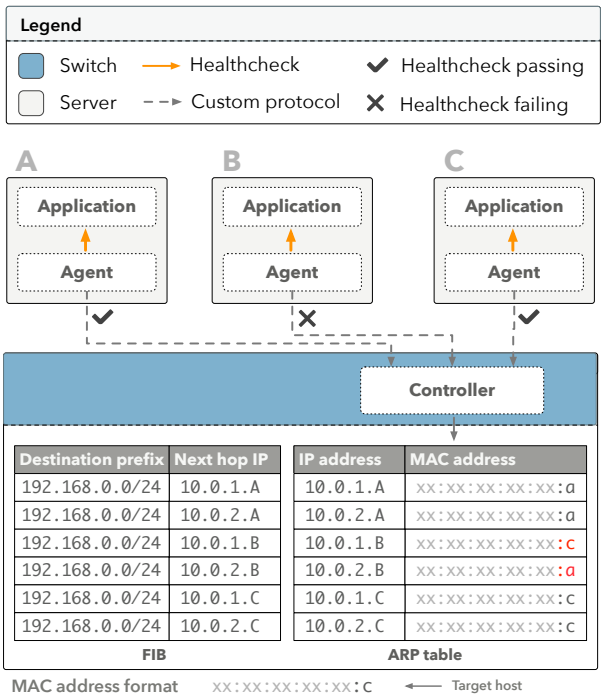


Figure 2: Custom routing protocol based on ARP table manipulation. The routing table remains static while the ARP table is adjusted to point at healthy hosts. Additional nexthops ensure even distribution of traffic when some hosts are unavailable.

ful is that all switches are configured to hash identically, so that all switches allocate flows to the same hosts.

A constraint of implementing consistent hashing in this manner is that the granularity with which traffic can be re-balanced is now directly tied to the number of nexthops used. If we only allocate one nexthop per host, removing a host from service would potentially double the amount of traffic towards a healthy neighbor. To avoid this, we can use more nexthops to provide finer-grained control, as illustrated in Fig. 2. In this example having two virtual nexthops per host is enough ensure that healthy hosts in the POP have an equal number of ARP entries directed at them when withdrawing B from production. In our implementation, we greedily reassign ARP entries to the hosts with the fewest mapped entries. Faild can approximate uniform load balancing by allocating a large number of ECMP nexthops to a smaller number of hosts. In practice we are constrained by the maximum number of ECMP nexthops supported by the underlying hardware (see Sec. 6.2).

Although switch vendors have recently started to provide consistent hashing natively [2, 4], Faild provides its own implementation for two important benefits. Firstly, by not relying on vendor implementations we lower the entry-level cost of eligible switches for use in our POP designs. More importantly, controlling the consistent hashing logic allows us to signal to target hosts whether the connections hashed onto them may have previously been hashed onto different hosts. This is crucial to use host TCP connection state to achieve transport affinity, as we show in Sec. 3.2.

In the example in Fig. 2, if the ECMP nexthops associated with B are re-allocated to other hosts in the pool, all ongoing connections towards host B will be terminated. Within large datacenters this problem is usually dealt with by using application-aware load balancers that track flow state and map new connections to healthy hosts, maintaining this mapping until completion. This approach runs counter to our goal of maximizing overall system capacity. In order to retain efficiency, we must push the equivalent functionality down towards the hosts.

3.2 Encoding failover decisions

Host draining cannot be implemented on switches alone, since they are layer 2/3 entities with no visibility of what flows are in progress towards each host at any given time. Instead, Faild distributes the responsibility for host draining across both the switch controller and hosts.

On the switch, a controller is responsible for using consistent hashing to steer traffic towards hosts in the failover set F. Hosts in F redirect traffic for ongoing connections back to their original destination in R, so that they continue to be served until their natural conclusion. Rather than relying on proprietary mechanisms, Faild implements detour routing when draining by extending the semantics of MAC addresses to encode load balancing state and its associated routing decisions, in addition to network interface identification.

Any host f in F can forward ongoing connections to their original host r in R if they ascertain the identity of this host. Conceptually, this can be done by annotating all drained traffic sent to f with the host r responsible for handling connections which were active before the drain episode started.

On baseline conditions all traffic for host r will be sent to MAC addresses with `r:r` suffix, and all hosts will receive the load assigned to them by the baseline distribution. Hence, in the baseline state the last two octets for all MAC addresses installed in the switch ARP table will be identical, signaling that all flows should be processed by their baseline host. In practice, switches forward frames to hosts based on the last identifier only.

Now consider host r being drained. Fig. 3 revisits the ARP table for the example in Fig. 2, this time using MAC addresses which reflect a host being drained from service, rather than simply removed. Faild will update all MAC addresses in the switch forwarding table that have suffixes of the form `r:X` by re-writing the penultimate octet to denote the *failover host* f. Hence, ECMP

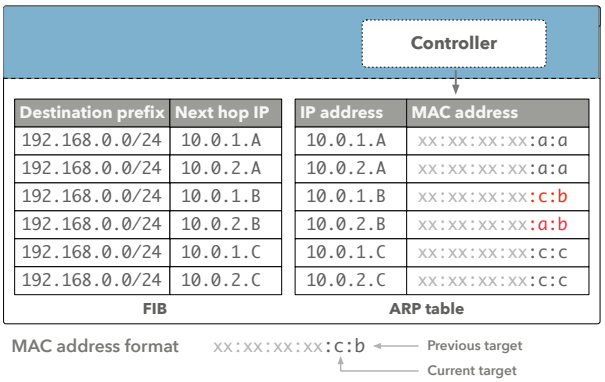


Figure 3: ARP table during draining of host B. The destination MAC address now encodes the drained host alongside the failover host.

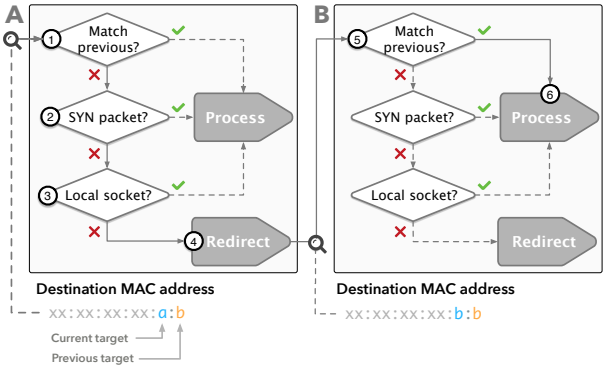


Figure 4: Receive-side packet processing example for traffic draining from host B towards host A. Packets filtered through host A are only accepted if they belong to a new connection, or if they match a local socket.

nexthops previously mapped to MAC addresses with an `r:r` suffix will now have an `f:r` suffix and traffic will be failed over to the correct host f in F.

By rewriting the ARP table in the switch, Faild can specify, for each IP nexthop mapped to a host r in R, the failover host f in F that will process new incoming requests. This improves upon alternative techniques such as switch-native consistent hashing: by appropriately selecting MAC addresses, Faild can inform a given host b which flows correspond to its baseline ECMP nexthops (if $b \in C$ for a given nexthop, the MAC address used will have a `b:b` suffix) and which ones correspond to failed-over ECMP nexthops (if $r \in R$ and $f \in F$ for a given nexthop, the MAC address used will have a `f:r` suffix). Such an explicit signaling is not achievable with switch-based implementations of consistent hashing.

3.3 Host-side processing

The method by which Faild steers traffic means we can no longer rely on routing protocols such as BGP and OSPF. Faild removes this responsibility from the routing layer, instead pushing it down to the data link layer. This is done using a controller that exchanges information with agents running on hosts and manipulates the ARP table on the switch. A host must therefore run an agent which is responsible for health checking local services, and is able to drain hosts if their associated service becomes degraded, non-responsive or placed under maintenance. Since this agent provides intelligent control over MAC-to-IP bindings, Faild switches do not use either ARP or IPv6 ND protocols.

Having received failover state information down from the switch, hosts can decide whether to process traffic locally or whether to forward it to a drained host. This not only removes the need for maintaining flow state within the network, but also distributes the computational cost of load balancing across a set of nodes which are more numerous and possess greater processing capabilities than switches. This computational cost is further reduced by implementing all of the receive-side processing as a single purpose kernel module, which efficiently processes inbound packets according to the destination MAC address (see Fig. 4).

When processing a frame, the kernel module receive handler at host h first determines whether the MAC address used is of the form `h:h` where h matches its own identifier. If so, packet processing is handed over to the local network stack, since we are operating as a baseline host ($h \in B$). Otherwise, the MAC address in the frame will be of the form `h:r`, where $h \in F$ and $r \in R$ (h is the failover host for r for this ECMP nexthop). The module must then verify whether the packet inside the frame belongs to a new connection, as signaled by the SYN flag in the TCP header (step 2), or to an existing connection, which can be verified by performing a lookup against the local socket table (step 3). If none of these conditions are met, the packet is redirected to the drain target r of the request by rewriting its destination MAC address to have a `r:r` suffix and returning the packet to a Faild switch. The same processing logic is applied at r (step 5). In this case, a `r:r` MAC address suffix indicates a locally maintained connection, and the frame will be immediately accepted. Outbound packets always follow the direct path towards their destination, resulting in asymmetrical packet forwarding during draining.

4 Implementation

This section details implementation specific nuances of Faild on both the switch and host.

4.1 Switch controller

The switch controller is implemented in approximately 3500 LOC of Python code and runs as a userspace dae-

mon on the control plane of a commodity switch built on merchant silicon. While we use a proprietary vendor API to access and configure data plane lookup tables, most of the functionality could be implemented in a portable way using OpenFlow [21], P4 [9] or SAI [5]. The controller operates on three tables:

- the **routing table**, by configuring VIP prefixes for each of the services to be load balanced and mapping them to the corresponding group of ECMP next-hops;
- the **ARP table**, by mapping each nexthop to the appropriate virtual MAC address as hosts are added or removed from the service pool; and
- the **bridging table**, by mapping each virtual MAC address to an outbound interface. Given the MAC addresses are virtual, there is no opportunity for learning what egress ports they map to. Instead, the controller must statically construct this mapping in order to avoid flooding traffic over all ports. This mapping can be provided via configuration, or autoconfigured using a custom protocol (*e.g.* LLDP discovery).

The switch controller also has the responsibility of keeping track of agent health checks, as well as ensuring that load is evenly distributed across available hosts. Despite its statelessness, a controller can use its logically centralized view of network health to implement a variety of load balancing strategies. Currently, agents encode three possible states for a health checked service: *up*, *down*, *disabled*. Operationally, there is an important distinction to be made between the *disabled* and *down* service states, since the former denotes an intentional withdrawal from service. While the underlying mechanics used to drain the host are the same in either case, a rapid sequence of downed hosts may be a symptom of a cascading failure, at which point a controller may decide to lock itself and fall back to standard ECMP.

Routing a packet of a load balanced connection requires up to two sequential lookups. The first is the routing table lookup, normally performed in TCAM, to resolve the destination IP address to a group of ECMP nexthops. The second involves the computation of a hash on the packet five tuples and the resolution to an entry of the nexthop group, normally stored on on-chip SRAM.

ECMP lookup table size does not adversely affect available space in the TCAM given that exact match lookups are executed on on-chip SRAM [10]. Hence, whereas supporting more services or fragmenting the VIP prefixes used will lead to an increase in TCAM footprint, adding more hosts or assigning more MAC addresses to each host will only require an increase in SRAM footprint. Faild leverages the lower cost and greater abundance of SRAM over TCAM to achieve improved horizontal scalability and decouple it from memory limitations.

4.2 Host agent

The host runs both a kernel module (1250 LOC in C) and a userspace daemon (2000 LOC in Python). The userspace daemon is responsible for configuring VIPs locally, executing healthchecks and relaying service health upstream to switch controllers. The kernel module is responsible for processing incoming packets with a Faild virtual MAC address as a destination. Depending on the address and the local socket table, the module will either deliver the packet locally or redirect the packet towards the alternate host encoded in the destination MAC.

In addition to the processing described in Fig. 4, the kernel module must add each of the secondary MAC addresses to the NIC unicast address filter using a standard kernel network driver API function. The NIC driver implements this function by adding the MAC address to the NIC's unicast perfect match filter. If this filter table is full, the NIC will resort to either using a hash-based filter, or by enabling unicast promiscuous mode, depending on the particular NIC model in use.

A further implementation nuance of the kernel module is in ensuring correct SYN cookie support. SYN cookies [8, 14] are vital in defending against large scale SYN floods. The difficulty when we are sending SYN cookies is that returning ACK packets will not match a local socket. As a result, we cannot determine whether they belong to a 3-way handshake for a connection that we ourselves sent a SYN cookie for and hence should be delivered locally, or whether they are regular ACKs for a drained host connection and hence should be forwarded.

Fortunately, there is a solution for this that is both simple and elegant. The Linux kernel enables SYN cookies automatically upon listen queue overflow. We check whether the listening socket on the local host has recently seen a listen queue overflow, and if so, we execute a SYN cookie validation on the ACK field. If the validation succeeds we deliver the packet locally and we forward the packet otherwise. Since a SYN cookie is a hash of a set of connection-related fields as well as some secret data and is designed to be nontrivial to forge, we are unlikely to accept an arbitrary ACK value as a valid SYN cookie. We are also unlikely to accept a SYN cookie generated by another host in the POP as a valid SYN cookie since each host uses a distinct secret hash seed.

5 Evaluation

In this section we evaluate the efficiency, resilience and gracefulness of Faild (see Sec. 1). In particular, we show that Faild can drain any system component without impacting end-to-end traffic (Sec. 5.1); can drain hosts in a timely manner (Sec. 5.2); does not induce a significant latency increase when detouring drained connections (Sec. 5.3); does not impose significant CPU over-

head on hosts (Sec. 5.4); and can achieve good load distributions across available hosts (Sec. 5.5).

All results presented were collected by means of active and passive measurements on our smallest production POP deployments, composed of only two switches and eight hosts. Despite occupying only half a rack, this particular instantiation of our POP design can scale up to 400 Gbps of throughput and handle up to 320k requests per second (RPS). Larger POPs can span up to four switches and 64 hosts.

5.1 Graceful failover

Sec. 3.1 claimed Faild maintains transport affinity when draining switches, allowing switches to be removed from operation without impacting established TCP connections. This is demonstrated in Fig. 5, that shows off-peak draining and refill of a Faild switch in a production POP. This POP only comprises two switches, both multi-homed to the same set of providers. The drain and refill events are visible in the graph at the top, in which all inbound traffic from one switch is initially shifted to the other and then shifted back. The other graphs highlight that both the number of requests served by POP hosts and the rate of reset packets (RST) and retransmissions they generate remain unchanged. The graceful removal and addition of one switch confirms that both switches must be applying the same hashing function.

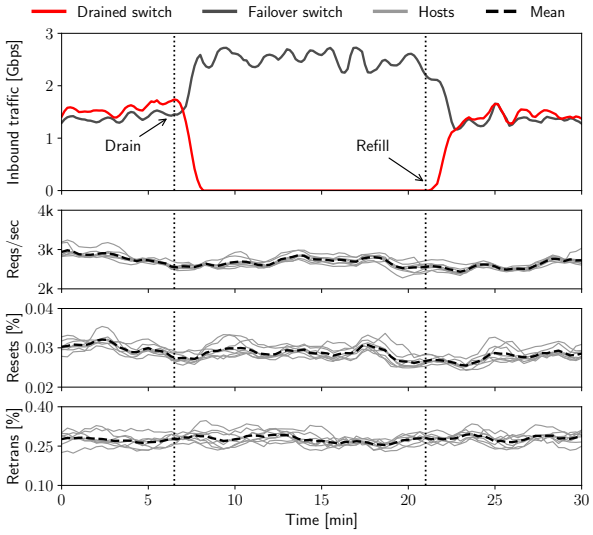


Figure 5: Graceful draining and refill of a switch

We proceed to demonstrate host draining. With all other components in steady state, the withdrawal of a single host from service should trigger Faild to redistribute traffic equally across all seven neighbors. This is observable in the top graph of Fig. 6, which shows the rate of requests handled by each host of a POP as a host is drained and refilled. We can make two important observations. First, upon disabling a host, the rate of VIP-related requests decreases rapidly and eventually converges to zero, on a timespan depending on the distribution of flow sizes active on the host when drained. Similarly, upon the host being re-enabled, the rate of requests rapidly converge to pre-draining values. Second, enabling and disabling a host does not cause any increase in RST and retransmission rates on any host. This validates that both events did not cause packets being delivered to incorrect hosts or dropped.

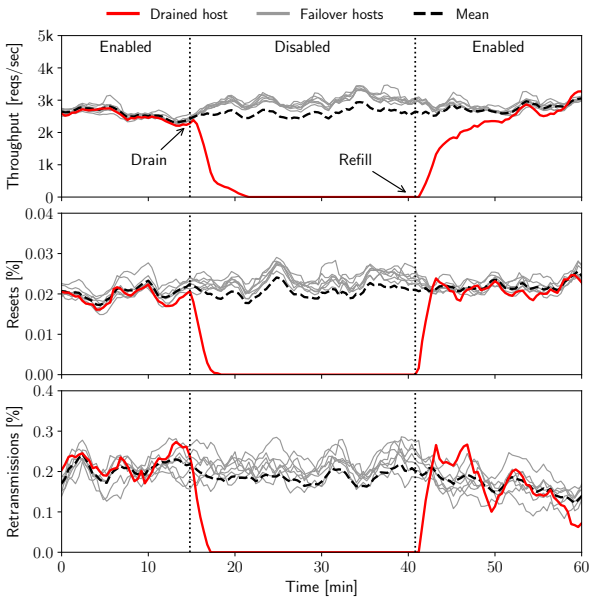
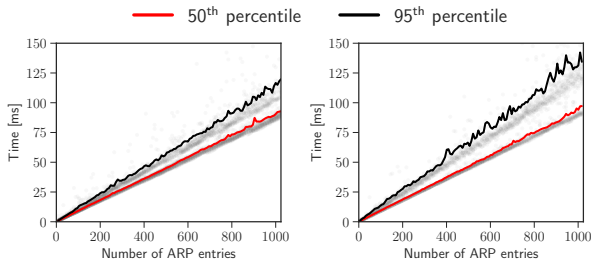


Figure 6: Host failover. Host d, in red, is drained and then refilled; traffic is shifted towards failover hosts $f \in F$, in grey, and then shifted back to d.

5.2 Switch reconfiguration time

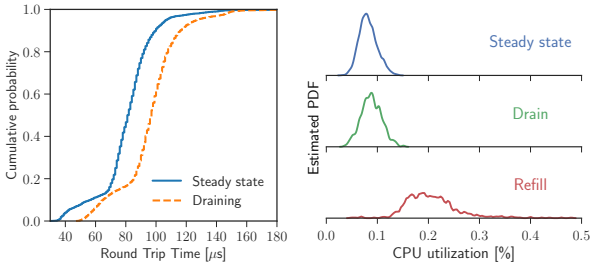
As discussed in Sec. 3.1, Faild implements host draining by means of a user space application running on the switch that updates the ARP table. For benchmarking purposes, we repurposed this application to perform batch ARP table updates of varying sizes. This was carried out multiple times on two models of production switches, referred to as switch A and B, equipped with different ASICs. Fig. 7 plots the measured time as a function of the number of ARP entries to be updated.

Roughly, we observe that overall reconfiguration time scales linearly with the number of updates. Even in the worst case scenario, requiring 1024 ARP entries to be updated, the 95th percentile is 119ms for switch A and 134ms for switch B. These values are low enough to ensure that, for the foreseeable future, Faild does not hinder our ability to react to host liveness in a timely manner. This is particularly true given ARP updates are atomic, and therefore service traffic is not disrupted during reconfiguration.



(a) Switch A (b) Switch B

Figure 7: ARP table reconfiguration time



(a) CDF of RTT (b) PDF of CPU utilization

Figure 8: Draining microbenchmarks

5.3 Detour-induced latency

Unfortunately, common debugging tools (*e.g.* *ping*, *traceroute* and *nc*) are not useful to measure the latency induced by detour routing during draining. This is because TCP SYN packets are never detour routed, and ICMP echo request packets do not trigger a lookup in the host socket table, which is crucial to obtain a realistic approximation of system behavior.

To measure detour routing latency we built a simple active measurement tool that runs on a non-Faild server q. The tool generates a stream of non-SYN TCP packets having a destination MAC address suffix `f:r` that specifies a node r as the drained host and f as its failover host. Since these are not TCP SYN packets, they trigger a lookup in the socket table when received by f. Since these packets do not correspond to an ongoing TCP connection, the lookup will not return a match, resulting in a detour through host r. When r attempts to process the packet it will not find a matching socket either, and will subsequently generate an RST packet and send it to q. By measuring the time between the generation of the non-SYN and RST packets q can measure the round trip latency. The direct round trip latency between q and f is measured using the same tool, but in this case the destination MAC address is set with two identical last octets, which causes f to reply directly with a RST packet.

The subsequent results are plotted in Fig. 8a, that shows the empirical CDF of RTT in steady-state and draining phases. Introducing a single hop detour induces a very low increase in end-to-end latency: the 50th percentile of the latency differential is 14 μs, the 95th percentile is 14.6 μs, and the 99th percentile is 19.52 μs. It should be noted that this small additional latency is observed only during a host draining phase and only by flows terminated at hosts being drained. In contrast, software load balancers add a latency between 50μs and 1ms [17, 15] to all packets they process.

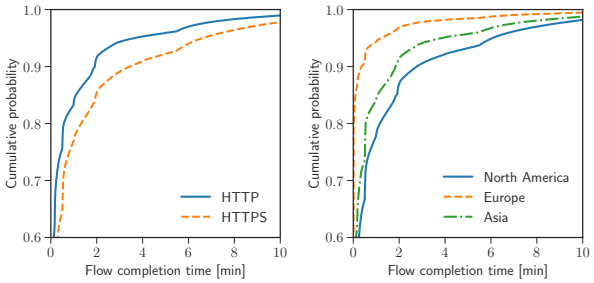
5.4 Host overhead

In order to measure the overhead incurred by the kernel component of Faild, we instrumented hosts to periodically collect stack traces on each CPU and count how many of those traces included function calls attributed to Faild. These counts are then normalized using the total number of traces collected. Since hosts run on GNU/Linux, we collected traces at 999 Hz to avoid synchronization with the 1 kHz timer interrupt frequency.

Fig. 8b shows the estimated probability density for the CPU utilization of the Faild kernel module calculated using Gaussian kernel density estimation [32]. We model the distribution for each one of three phases: steady-state, draining and refill. Data referring to host draining and host refill was collected over the first two minutes after the transition was triggered.

In steady-state we can observe that CPU utilization is very low, averaging approximately 0.1%. The utilization remains approximately constant as draining begins, and during the drain event itself. A noticeable increase occurs when we revert the draining operation and shift traffic back to the drained host, which we denote as *refilling*. At this point the kernel module receives a large number of packets in which the last two octets of the destination MAC address differ, which requires a lookup in the socket table to be able to trigger detour forwarding in case of a miss. In spite of the increased workload, the average CPU utilization remains remarkably low, peaking at 0.5% for a very small subset of samples. In general, the expected utilization during refill is 0.22%.

Our two minute measurement cut-off is justified because flows usually terminate quickly after a drain event starts. To verify this, we measured the distribution of flow completion times from different vantage points on our network. The resulting distributions for three of our POPs in distinct geographic regions are plotted in Fig. 9. While these results are biased towards our customer base and the configuration settings for their applications, they provide us with a feel for the underlying properties of flows in flight. Our analysis shows that most flows are short: between 60% and 70% of flows last less than 10 seconds and between 78% and 85% last less than 1 minute. Host overhead is therefore not only small to begin with, but also decays rapidly over time.

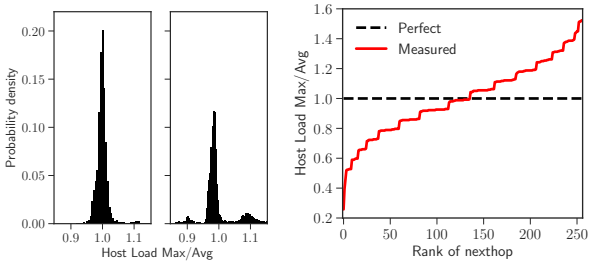


(a) By service (b) By region

Figure 9: Distribution of flow completion time

5.5 Load balancing accuracy

In order to achieve good load distributions, 1) switches must provide a good enough hardware implementation of ECMP hashing; and 2) the mix of inbound traffic must be such that ECMP will provide sufficiently homogeneous traffic balancing. We validate these two requirements.



(a) Switch A and B (b) Switch C

Figure 10: Granularity of ECMP load balancing

To measure load balancing quality we sample two hosts, each one from a POP equipped with a different switch model, and assign the same number of nexthops to each. By measuring the traffic volume served by either host at 5 minute intervals over a 24h period, we can calculate the actual fraction of traffic volume served by each host normalized by the average amount of traffic served by all hosts for each interval. Fig. 10a shows the distribution of this fraction for both switch models; the density concentration around unity shows that load balancing quality is good for both switches used in production. As shown in Sec. 6.3, this does not always hold (*e.g. Switch C*, Fig. 10b).

6 Operational experience

Faild derives much of its elegance from a core set of simplifying assumptions. This section revisits our assumptions in light of our operational experience and highlights their practical implications.

6.1 Recursive draining and POP upgrades

A limitation of the stateless architecture of Faild is that hosts that are actively forwarding drained traffic cannot be drained themselves. If any nexthops have MAC addresses with a `f:r` suffix (indicating that traffic for r is already being failed over to f), failing over host f to host g would imply updating these suffixes to have the form `g:f`, and the previous draining indirection from r to f would be lost. If f were still forwarding ongoing connections for r at this point, they would be reset by g as they would not have an associated open socket. This type of *recursive draining* fails because it cannot be effectively represented with our current MAC address encoding, and this means that Faild is unable to recover when a host that is already forwarding drained traffic fails.

While trivial to address by further overloading MAC address semantics, in practice this shortcoming is not a concern. For one, it is mitigated trivially by waiting for drained traffic to decay naturally. As long as the average time between failover reconfigurations is greater than the time needed for ongoing connections to end naturally once a draining episode is started, no customer traffic will be affected. Since edge traffic is preponderantly composed of small objects, flow completion time is low and drained traffic decay is fast, as observed in Fig. 9. Additionally, the probability for overlapping draining episodes is correspondingly low. As shown in Fig. 1, every one of our POPs had a worst-case draining rate of 5 events per minute or lower at the 99.9$^{\text{th}}$ percentile, irrespective of underlying event cause. In our experience, the benefits of recursive draining do not justify the resulting increase in operational complexity.

Faild can implement seamless addition of hosts simply by draining an appropriate number of ECMP nexthops to the new hosts being deployed. However, instead of updating all MAC addresses with suffixes `r:r` associated with a drained host r, the ECMP nexthops to be reallocated are chosen from the entire forwarding table in such a way that the resulting configuration is balanced. After a host f is inserted, and once draining traffic has subsided, addresses with `f:X` suffixes can be re-labeled as `f:f` to distinguish the change as permanent. In order to simplify the addition of hosts to a POP, Faild switches are configured to use the maximum number of nexthops supported by the hardware, which are then allocated to the initial number of hosts in a balanced manner.

6.2 Scalability challenges

When scaling Faild to larger POPs, the main bottleneck in a single-layer topology will be the port density of switches; this can be mitigated by using alternative topologies with multiple switch layers. However, a challenge arises when performing ARP table updates on switches running Faild but not directly connected to

hosts: bridging table changes need to be synchronized. This can be achieved with either custom agents running on switches or using *e.g.* LLDP discovery.

Although our MAC address encoding limits POP size to 256 hosts, we have yet to come close to this limit. The tight latency bounds crucial for edge clouds push for geographically diverse POPs with a small number of high-end servers, counterbalancing the economies of scale driving conventional datacenter deployments. It would however be simple to extend the host encoding in MAC addresses to support larger POPs.

A further challenge arises when the maximum number of ECMP nexthops supported by the underlying switch hardware is insufficient to provide adequate load balancing. Recent commodity switches provide 2048+ nexthops, which we have found to be sufficient for our needs.

The current layer 2 architecture is attractive because it is efficient and widely supported and tested. Furthermore, IPv6 is trivially supported and kernel packet processing on the hosts is simpler than alternative encapsulation methods. If necessary however Faild can operate in the network layer without significant modifications. By simply using IP-in-IP ECMP nexthops bound with host IP addresses and with statically configured ARP tables, all the benefits that we have described become available in a layer 3 architecture.

6.3 ECMP hashing assumptions

While ECMP hashing plays an important role in most load balancing solutions [17, 27], it is central to Faild. Our original design and subsequent analysis make numerous assumptions which were empirically validated in Sec. 5. This, however, is a reflection of our own selection bias rather than representative for all switch models. While evaluating various hardware options, we came across numerous implementation flaws.

Uneven hashing. Some switches under evaluation were incapable of hashing evenly. The worst example was a switch that, when configured with 256 ECMP nexthops for a given destination, hashes traffic according to the ranked load distribution in Fig. 10b. For this particular switch model, the most and least heavily loaded of the 256 ECMP nexthops differ in allocated traffic share by a factor of approximately six. The impact of this cannot be understated, given that it is uncommon for customers to rigorously evaluate hashing, and instead rely on vendor claims to drive capacity planning. As a result, it is likely that many commercial networks reliant on ECMP are far more prone to overloading capacity than they realize.

Unusable nexthops. Some switches also have odd restrictions on the number of usable ECMP nexthops for any given destination. In particular, one model we tested appears to only support ECMP nexthop set sizes that are of the form $\{1,2,\ldots,15\} \times 2^n$, presumably because of hardware limitations. Configuring an ECMP route with a nexthop count not of this form will result on the next lower number of nexthops of this form to be used. For example, configuring this switch with an ECMP route with 63 nexthops will cause only 60 (*i.e.* 15×2^2) of the nexthops to be used, and 3 of the nexthops will thus receive no traffic at all. As a result, a switch may hash evenly amongst internal buckets, and yet still lead to a skew in load distribution because of a mismatch in how these buckets map to nexthops.

Hash polarization. For a Faild instance with multiple ingress switches, each switch should be configured to hash in the same manner. If this is not the case, an external routing change may divert a flow through a different switch, which in turn may hash the flow onto a different host leading to TCP connection resets.

Vendors however can make such hash polarization impossible to achieve in practice. ECMP hash polarization is often configured unintentionally by omission, and can lead to poor performance in networks employing multiple levels of ECMP routing, as it leads to correlation between ECMP nexthop decisions made at different levels of the network hierarchy. Some vendors have addressed this potential misconfiguration by introducing additional sources of entropy into their ECMP hashing functions.

Unfortunately we found that many switch models include the index of the ingress interface of a packet in that packet's ECMP nexthop hash computation, and in one particular case, we found that line-card boot order was used to seed the ECMP hashing function. If the hardware vendor does not provide a knob to disable such behavior, hash polarization is rendered impossible, which has dire consequences for our use case.

6.4 Protocol assumptions

One might expect that, in the absence of routing changes, packets belonging to the same flow follow a single network path. This however assumes that any load balancing along the path, including that applied by Faild itself, is consistent across all packets within a flow. This section reviews cases where this does not hold.

Inbound fragmentation. If we receive a TCP segment that has been fragmented by either the originating client or an intermediate router, its additional fragments will not contain the TCP port numbers for the connection, which will cause the receiving switch to hash the initial fragment and the set of additional fragments to different hosts in the POP.

IPv4-speaking clients that use IPv4 Path MTU Discovery [23] transmit TCP segments that have the IPv4 Don't-Fragment bit set, and will therefore not be fragmented by intermediate routers. Anecdotally, virtually all IPv4 clients we see traffic from implement Path MTU Discovery and transmit TCP segments that are unfragmented

and have the IPv4 Don't-Fragment bit set. In IPv6 packets are never fragmented by intermediate routers [12], and fragmented IPv6 TCP segments belonging to established connections are exceedingly rare. Considering this, we continue to allow switches to use port numbers for hashing, as this does not appear to cause operational problems. Nevertheless, a possible mitigation for this problem is to configure Faild switches to exclude the received segment's TCP port numbers in its packet hash computation. This would allow successful defragmentation and processing of received TCP segments, but could harm load balancing evenness.

Outbound fragmentation. Inevitably we will receive ICMPv4 Fragmentation Needed or ICMPv6 Packet Too Big messages in response to outgoing TCP segments destined for a network (or link) that uses an MTU lower than the segment size. As noted in [11], such ICMP messages, and ICMP errors in general, are not guaranteed to ECMP-hash back to the same host as the one that was handling the corresponding TCP session. Faild handles this by having hosts that receive certain ICMP messages from upstream switches rebroadcast these messages to all hosts in the local POP, and by processing received ICMP messages that were rebroadcast in this manner as if they were unicast to the local host. This ensures that the host that was handling the TCP session corresponding to an inbound ICMP message will receive and process a copy of that ICMP message. Given this is a potential denial-of-service vector, we also implemented a mechanism for rate limiting broadcasts.

ECN. In 2015 we experienced an increase in reports of connection resets coinciding with Apple enabling Explicit Congestion Notification [28] by default on iOS and OS X. Further investigation revealed that this did not affect all users; it was dependent on network path.

Prior to the deprecation of the IPv4 Type of Service field and its redefinition as the Differentiated Services field by RFC 2474 [24], and the reservation of the last two bits of the DS octet for ECN by RFC 3168 [28], it was explicitly permitted by the IPv4 router requirements RFC [7] to involve the second-to-last bit of the TOS/DS octet in routing decisions, and at least one major operator in the US and several operators outside the US appeared to have deployed network devices which take this bit into account in ECMP nexthop selection. Given that this can cause packets for a single ECN-using flow to follow different paths, there is the potential for trouble if such a flow hits a POP where Faild is operating across multiple switches which do not support hash polarization.

Given we are still phasing out switches which do not support hash polarization from our network, we decided instead to disable ECN negotiation entirely. The expectation that all packets within a flow follow the same path however is likely to be broken for ECN for a much longer period of time, as devices hashing on ECN bits are embedded in networks with long hardware refresh cycles.

SYN proxies. More recently, we uncovered a similar lack of packet affinity between our POPs and a major cloud provider which had deployed a SYN proxy for their enterprise platform. In this case, a software proxy undertakes the responsibility of establishing outbound connections, and then passes the established flow to the proxied host. In practice, this results in separate route lookups - one for the SYN handshake, and another for the subsequent data. While this corner case appears similar on the surface to ECN, it is much harder to work around. After extended discussions with the cloud provider in question, it was decided that it would be simpler to use BGP to pin an ingress path until we could upgrade our contingent switches to support hash polarization.

7 Related work

We now review how existing proposals fall short of the load balancing requirements for POP deployments. A comparative summary is provided in Table 1.

Consistent hashing ECMP provided by recent switches (*e.g.* [2, 4]) is a typical solution adopted by CDNs to achieve stateless load balancing but does not make it possible to gracefully add and remove hosts.

Ananta [27] and Maglev [15] propose the use of software load balancers, with the latter improving packet throughput through batch processing, poll mode NIC drivers and zero copy operations (also adopted by [13, 29]). Despite these improvements, both require per-flow state to provide connection persistence.

Duet [17] and Rubik [16] combine the ECMP capability of commodity switches with a software load balancer to address the performance bottleneck of general purpose hardware. They configure routing for heavy hitting VIPs directly on switches, and relegate less popular VIPs towards software load balancers. Although both add cursory support for migrating flow state between switches and software load balancers, this proves impractical to implement. For one, the approach assumes that the ECMP hash function and seed used on the switches can be replicated on load balancer instances. This however is typically proprietary knowledge which equipment manufacturers are unwilling or unable to disclose [30]. Even if this were not the case, the resulting implementation can still lead to flow disruption, as demonstrated in motivating the design decisions of SilkRoad [22].

SilkRoad [22] implements connection tracking in programmable switches by storing compressed flow state in SRAM memory, which can be potentially overrun. It also supports graceful draining of hosts without the high latency and low throughput of software load balancers, at the cost of requiring the switch control plane to execute a

Table 1: L4 load balancers: suitability comparison for POP deployment

System	Low latency	No dedicated HW	Stateless	Host draining	Switch draining	In production
Consistent ECMP [2, 4]	✓	✓	✓	✗	✓	✓
Ananta [27], Maglev [15]	✗	✗	✗	✓	✗	✓
Duet [17], Rubik [16]	✓	✗	partial	✓	partial	✗
SilkRoad [22]	✓	✓	✗	✓	✗	✗
Beamer [25]	✗	✗	✓	✓	✓	✗
Faild	✓	✓	✓	✓	✓	✓

table insertion for each received SYN. As a result of both these features, SilkRoad is vulnerable to resource exhaustion attacks and therefore not an appropriate match for load balancing within a POP. SilkRoad eliminates the need for load balancer instances to be deployed on hosts, but in doing so hinders the ability to drain switches.

The closest approach to Faild is Beamer [26, 25]. Beamer also implements consistent hashing by mapping a fixed number of buckets to hosts and uses detouring to drain hosts without keeping per-flow state, but is predated by Faild on both counts [34, 35, 36, 37]. However, whereas Faild does not require any additional hardware, Beamer requires dedicated hosts to run controllers and load balancer instances, making it unsuitable for deployment within a POP. Finally, the design of Faild is informed by years of operational experience. In addition to the benefits provided by Beamer, it provides explicit support for SYN cookies and maintains correct operation when facing anomalous cross-layer protocol interactions (*e.g.* inbound/outbound packet fragmentation).

A number of techniques used in Faild have also been used singularly albeit in different contexts and for different purposes. Extending MAC address semantics to implement switch-to-host signaling was used in [20] to allow switch forwarding based on application layer information. Mapping a large number of virtual nexthops to a much smaller number of physical nexthops was used in [40] to implement weighted multipath forwarding. Faild is unique in using these mechanisms to achieve graceful, stateless load balancing and having been validated extensively in production.

8 Conclusion

This paper revisits load balancing in the context of edge clouds, which are orders of magnitude more dense than datacenter-based cloud computing environments. Stripped of virtually unbounded physical space, even individual failures can have a noticeable impact on availability and cost becomes primarily driven by efficiency.

In light of these issues, this paper proposed Faild, a stateless, distributed load balancing system which supports *transport affinity*, hence allowing the graceful failover of any individual component. Constrained to commodity hardware, Faild redefines the semantics of existing network primitives and rethinks the allocation of load balancing functions across network components. We demonstrate that commodity switches can be leveraged to perform fast, stateless load balancing, while retaining the ability to signal failover forwarding information toward the hosts.

In optimizing our design for cost, we inadvertently implemented all functions necessary to achieve graceful failover. A key insight in this paper is that all state needed for graceful failover is readily accessible within the host kernel itself. Previous load balancers rely on additional per-flow state to track the allocation of flows to hosts: a costly, unscalable design pattern which is prone to resource exhaustion attacks. Faild instead inspects existing socket state to determine whether to failover traffic to neighboring hosts. Our results show that this is seamless to clients while incurring negligible CPU overhead and minimal delay. By keeping a tight focus on solving the engineering challenges of edge clouds, Faild combines well-known techniques in a novel way to achieve graceful addition and removal of any component without requiring per-flow state beyond that already present in the service hosts themselves.

This paper reflects our collective experience in scaling Faild over the past four years to handle in excess of seven million requests per second for some of the most popular content on the Internet. While the core design principles of Faild have fared remarkably well, we have strived to document cases where our assumptions proved overly optimistic. In exposing hardware limitations and unintuitive protocol interactions, we hope that with time these issues may begin to be addressed by a wider community.

Acknowledgments

We would like to thank our shepherd KyoungSoo Park and the anonymous reviewers for their feedback.

Faild was originally conceived by Artur Bergman and Tyler McMullen, and owes much to the many engineers at Fastly who contributed to its implementation and operation over the past four years.

References

[1] A10 Networks. https://www.a10networks.com/.

[2] Broadcom Smart-Hash. https://docs.broadcom.com/docs/12358326.

[3] F5 Networks. https://f5.com/.

[4] Juniper Networks - Understanding the Use of Resilient Hashing to Minimize Flow Remapping in Trunk/ECMP Groups. http://www.juniper.net/techpubs/en_US/junos15.1/topics/concept/resilient-hashing-qfx-series.html.

[5] Switch Abstraction Interface. https://github.com/opencomputeproject/SAI/.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '08)*.

[7] F. Baker. Requirements for IP Version 4 Routers. IETF RFC 1812 (Proposed Standard), June 1995.

[8] D. J. Bernstein. SYN cookies. http://cr.yp.to/syncookies.html.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*.

[11] M. Byerly, M. Hite, and J. Jaeggli. Close Encounters of the ICMP Type 2 Kind (Near Misses with ICMPv6 Packet Too Big (PTB)). IETF RFC 7690 (Informational), Jan. 2016.

[12] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. IETF RFC 2460 (Draft Standard), Dec. 1998.

[13] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[14] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations. IETF RFC 4987 (Informational), Aug. 2007.

[15] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*.

[16] R. Gandhi, Y. C. Hu, C. kok Koh, H. Liu, and M. Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *USENIX Annual Technical Conference (USENIX ATC '15)*.

[17] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '14)*.

[18] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. IETF RFC 2992 (Informational), Nov. 2000.

[19] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '15)*.

[20] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, (NSDI '16)*.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[22] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful Layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[23] J. Mogul and S. Deering. Path MTU discovery. IETF RFC 1191 (Draft Standard), Nov. 1990.

[24] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. IETF RFC 2474 (Proposed Standard), Dec. 1998.

[25] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[26] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox '16)*.

[27] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*.

[28] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. IETF RFC 3168 (Proposed Standard), Sept. 2001.

[29] L. Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*.

[30] L. Saino. Hashing on broken assumptions. In NANOG '70. https://www.nanog.org/sites/default/files/1_Saino_Hashing_On_Broken_Assumptions.pdf, June 6th, 2017.

[31] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[32] B. W. Silverman. *Density estimation for statistics and data analysis*. Chapman and Hall, London, 1986.

[33] 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. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '15)*.

[34] J. Taveira Araújo. Communication continuation during content node failover. US Patent 9,678,841. Filed May 30th, 2014, Issued Jun. 13th, 2017.

[35] J. Taveira Araújo. Failover handling in a content node of a content delivery network. US Patent 9,569,318. Filed May 30th, 2014, Issued Feb. 14th, 2017.

[36] J. Taveira Araújo. Scaling networks through software. In *USENIX SREcon '15*.

[37] J. Taveira Araújo, L. Saino, and L. Buytenhek. Building and scaling the Fastly network, part 2: balancing requests. `https://www.fastly.com/blog/building-and-scaling-fastly-network-part-2-balancing-requests/`, Dec 2016.

[38] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. IETF RFC 2991 (Informational), Nov. 2000.

[39] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global Internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[40] 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 (EuroSys '14)*.