6LB: Scalable and Application-Aware Load Balancing with Segment Routing

Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, Thomas Heide Clausen

6LB: Scalable and Application-Aware Load Balancing with Segment Routing

Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, Thomas Clausen

***Abstract*—Network load-balancers generally either do not take application state into account, or do so at the cost of a centralized monitoring system. This paper introduces a load-balancer running exclusively within the IP forwarding plane, *i.e.* in an application protocol agnostic fashion – yet which still provides application-awareness and makes real-time, decentralized decisions. To that end, IPv6 Segment Routing is used to direct data packets from a new flow through a chain of candidate servers, until one decides to accept the connection, based solely on its local state. This way, applications themselves naturally decide on how to fairly share incoming connections, while incurring minimal network overhead, and no out-of-band signaling. A consistent hashing algorithm, as well as an in-band stickiness protocol, allow for the proposed solution to be able to be reliably distributed across a large number of instances.**

Performance evaluation by means of an analytical model and actual tests on different workloads (including a Wikipedia replay as a realistic workload) show significant performance benefits in terms of shorter response times, when compared to a traditional random load-balancer. In addition, this paper introduces and compares kernel bypass high-performance implementations of both 6LB and a state-of-the-art load-balancer, showing that the significant system-level benefits of 6LB are achievable with a negligible data-path CPU overhead.

***Index Terms*—Load-balancing, Segment Routing (SR), IPv6, Application-aware, Consistent hashing, Performance evaluation.**

I. Introduction

Virtualization and containerization has enabled scaling of application performance by way of (i) running multiple instances of the same application within a (distributed) data center, and (ii) employing a load-balancer for dispatching queries between these instances.

For the purpose of this paper, it is useful to distinguish between two categories of load-balancers:

1. Network-level load-balancers, which operate below the application layer – a simple approach being to rely on Equal Cost Multi-Path (ECMP) [1] to homogeneously distribute network flows between application instances. This type of load-balancer typically does not take application state into account, which can lead to suboptimal server utilization.

2. Application-level load-balancers, which are bound to a specific type of application or application-layer protocol, and make informed decisions on how to assign servers to incoming requests. This type of load-balancer typically incurs a cost from monitoring the state of each application instance, and sometimes also terminates network connections (such as is the case for an HTTP proxy).

A desirable load-balancer combines the best of these categories: *(i) be application or application-layer protocol agnostic* (*i.e.* operate at below the application layer) and *(ii) incur no monitoring overhead* – yet *(iii) make informed dispatching decisions depending on the state of the applications*.

Furthermore, data centers are more and more utilized to run virtualized network functions alongside traditional applications. In light of this, network load-balancers are more and more running as virtual functions (running *e.g.* in virtual machines). This allows for the load-balancers themselves to take full advantage of the flexibility and redundancy of a virtualized data center: for resiliency, to allow a faulty load-balancer instance to be safely removed and replaced, without incurring a service outage, or for scalability, by growing (and shrinking) the number of load-balancers to be able to accommodate different daily traffic demands and/or unexpected traffic peaks. The resulting architecture will thus distribute incoming flows between an edge router and several load-balancer instances, each of which will redistribute the flows to application instances [2], [3]. A challenge arising from this architecture is to provide a consistent service when traffic for a given flow is directed by the edge router to a different load-balancer instance. This can happen *e.g.* when a load-balancer instance is added or removed, causing the corresponding ECMP mapping between the edge router and the load-balancers to be updated correspondingly.

Thus in addition to the three desired properties exposed above can be added: *(iv) be able to be fully distributed*, providing the same service regardless of whether traffic is directed to different load-balancer instances within the lifetime of a flow.

These four objectives may appear irreconcilable: operating below the application layer makes it hard to take application state into account, and balancing by state rather than deterministically ties a flow to a given load-balancer instance. This paper aims at providing a solution satisfying these four objectives, by challenging the traditional network paradigm wherein a packet is deterministically assigned only one destination.

A. Statement of Purpose

The purpose of this paper is to propose 6LB, a load-balancing approach that is application server load aware, yet is both application and application-layer protocol independent and does not rely on centralized monitoring or transmission of application state.

Y. Desmouceaux, P. Pfister, J. Tollet and M. Townsley are with Cisco Systems Paris Innovation and Research Laboratory (PIRL), 92782 Issy-les-Moulineaux, France; emails {ydesmouc,ppfister,jtollet,townsley}@cisco.com.

Y. Desmouceaux, M. Townsley and T. Clausen are with École Polytechnique, 91128 Palaiseau, France; emails {yoann.desmouceaux,mark.townsley,thomas.clausen}@polytechnique.edu.

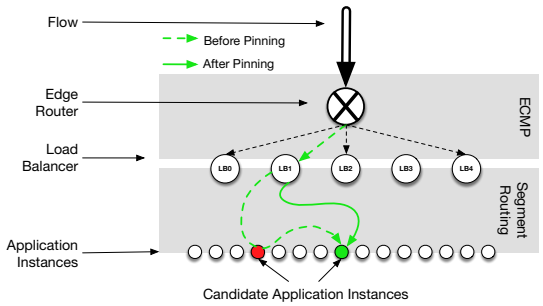


Figure 1. 6LB architecture: load-balancers assign a flow to a set of candidate instances, through which the connection is passed until one accepts the connection (section II). The flow is then pinned to the server having accepted the connection (section IV).

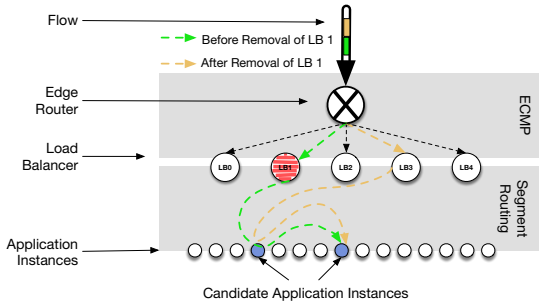


Figure 2. 6LB consistent hashing: when a flow is rebalanced to another load-balancer instance by the edge router, consistent hashing (section III) allows the flow to re-browse the same set of candidate instances, then to be re-pinned to the one that had accepted the connection (section IV).

A key argument behind this design goal is that an application instance itself is best positioned to know if it should be accepting an incoming query, or if doing so would degrade performance. Thus, 6LB disregards any design by which queries are unconditionally assigned to an application instance by the load-balancer. Rather, 6LB offers a received query to several candidate application instances, ensuring that exactly one instance accepts and processes the query.

The architecture behind 6LB is as follows, and as illustrated in figure 1: the edge router receives and uses ECMP to assign each incoming flow to a load-balancer. Each load-balancer selects a set of candidate application instances, to which it forwards (in order) the flow, using IPv6 Segment Routing (SR) [4], which permits directing data packets through an (ordered) set of intermediaries (see section I-C). In this way, 6LB enables that flow acceptance decisions are made strictly locally by an application instance, based on its real-time state information about itself, only – *i.e.* without any centralized monitoring.

Once a flow is accepted by an application instance, it is "pinned" to it by the load-balancer: subsequent packets in that flow are all forwarded directly to that application instance. As depicted in figure 3, this is accomplished by the load-balancer inspecting the TCP handshake and establishing a mapping between a flow and the application instance serving it. A mechanism is provided to permit a load-balancer to recover

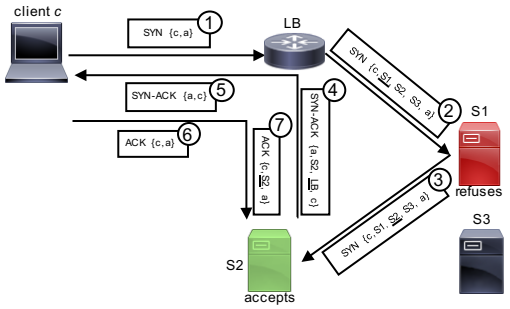


Figure 3. TCP Connection pinning from client c to application a with 3 instances s_1, s_2, s_3. The path (source, segments, destination) is indicated between curly braces. The active segment is underlined.

this mapping, if for some reason it is lost[1].

The final mechanism in 6LB is consistent hashing, as illustrated in figure 2: a given flow will be assigned to the same set of candidate application instances, regardless of by which 6LB instance (past, current, or future, in case the pool of instances changes) it is assigned.

While the algorithms developed in this paper are generally applicable for any number of candidate application instances, the concept of *power of two choices* [5] applies: selecting two candidate application instances ensures a low network footprint while providing a significant load-balancing improvement – with diminishing returns beyond that. Building on the work of [6], the novel contributions of this paper are fivefold: (i) a consistent hashing algorithm, (ii) an in-band stickiness protocol (the union of which allows to scale the number of load-balancer instances for reliability and performance), (iii) an analytic model analyzing the performance of SR-based load-balancing, (iv) experiments with the combined 6LB load-balancing architecture conducted on a larger testbed, and finally (v) kernel-bypass implementations of 6LB and a state-of-the-art load-balancer (Maglev [2]) with performance comparison of the two implementations.

B. Related Work

Among existing load-balancing approaches below the application layer, Maglev [2] and Ananta [7] aim at providing a software load-balancer instance that can be scaled at will, and make use of ECMP to distribute flows between those instances. In addition to a flow stickiness table, they also make use of consistent hashing [8], [9], [10], for ensuring that data packets within a given flow are directed to the same application instance – regardless of the selected load-balancer instance forwarding a data packet, and with minimal disruption when the set of application servers changes. However, flows are distributed to application instances regardless of their current load. This is taken one step further in Duet [11] and Rubik [12], by moving the load-balancing function to hardware instances, while handing traffic over to software instances in case of failure. Conversely, [13], [14] use Software Defined Networking (SDN) on a controller, to monitor the

[1]For example, if a load-balancer is removed, and another load-balancer takes over the active flows it was serving.

Next Header	Ext Hdr Length	Routing Type = 4	Segments Left
First Segment	Flags		Reserved
Segments...			
Optional TLVs...			

Figure 4. IPv6 Segment Routing Header

application instance load and network load – and then install network rules to direct flows to these application instances. Other frameworks, such as [15], can be used to gather precise monitoring information on the network load.

Simple application-aware load-balancing policies (random, shortest queue, threshold) have been introduced in [16]; in [5], [17], it is shown that performing a load-balancing decision based on two random servers is sufficient to exponentially decrease the response time as compared to a random strategy. This concept has been used by [18] for peer-to-peer applications. Another similar idea, proposed in [19], [20], consists of duplicating queries among several replicas, and serving the quickest reply to the client. In [21], SR is used to duplicate traffic through two different disjoint paths, so as to decrease latency and packet loss.

In [22], three load-balancing techniques are listed, which can be used for dispatching queries among Web servers: DNS round-robin, dispatchers that perform NAT or destination IP rewrite, and redirect-based approaches. Application-aware load-balancing mechanisms for static Web content include [23], [24], [25], which assign queries as a function of their estimated size so that each application instance becomes equally loaded. In [26], a feedback approach is used to estimate the parameters of a queuing model representing the system, before making a load-balancing decision.

Application-layer protocol aware load-balancers, *e.g.* HAProxy [27], also propose application-awareness by estimating the load on each application instance and assigning new queries accordingly. Load estimates are obtained either by tracking open connections through the load-balancer to the backend servers (and thus do not consider other loads), or by periodically probing the backends for load information (and thus suffer from polling delay and incur network overhead). Another issue with application-level load-balancers is that network connection is reset when a failure causes a flow to be migrated from one load-balancer instance to another. The load-balancer introduced in [28] aims at solving this by keeping per-flow TCP state information in a distributed store.

C. Segment Routing

IPv6 Segment Routing (SR) [4] permits directing data packets through an (ordered) set of intermediaries, and instructing these intermediaries to perform a specific function [29]. For example, one instruction could be *"process the contained query, if you are not too busy"*. As SR is a network layer service, segments are expressed by way of IPv6 addresses, and the simplest possible sequence of segments interprets into "forward the packet to A, then B, then C" – *i.e. source routing*. The SR information is expressed as an IPv6 Extension Header, defined in [30] (figure 4), comprising a list of segments and a counter `SegmentsLeft` – indicating the number of remaining segments to be processed. Although [30] specifies that the last segment in the header is the first segment to be processed (*i.e.* the order of segments in the packet is reversed), for the purpose of readability this paper will use the convention that $(s_1, \ldots, s_n)$ represents an SR header indicating s_1 as a first segment to be traversed.

D. Paper Outline

The remainder of this paper is organized as follows. Section II describes the use of Segment Routing for performing load-balancing. Section III describes a consistent hashing algorithm, which allows to distribute the load-balancing function into different instances, for scalability. Section IV describes how "stickiness" can be established and recovered between load-balancer instances and server instances, using an in-band channel. An analytical performance model of 6LB is derived in section V. 6LB is then experimentally evaluated in section VI, by means of a synthetic workload and a realistic workload consisting of a Wikipedia replica; and the performance of the implementation in terms of packet forwarding capabilities is evaluated. Finally, section VII concludes this paper.

II. Service Hunting with Segment Routing

A. Description

Service Hunting uses SR to direct network packets from a new flow through a set of *candidate application instances* until one accepts the connection. It assumes that applications are identified by *virtual IP addresses* (VIPs), and can be replicated among several servers, identified by their *topological addresses*. Servers run a *virtual router* (*e.g.* VPP [31]), which dispatches packets between physical NICs and application-bound virtual interfaces. Finally, a load-balancer within the data center advertises routes for the VIPs.

When a new flow[2] for a VIP arrives at the load-balancer, it will select a set of candidate application instances from a pool, and insert an SR header identifying this set into the IPv6 data packet. The SR header will contain a list of segments, each indicating that the query can be processed by either of these application instances, and with the VIP as the last segment. Different policies can be used to select the list of candidate application instances to include in the SR header. It has been shown in [5] that selecting two random candidate application instances is enough to greatly improve load-balancing fairness, with a decreasing marginal benefit when using more than two instances. Thus, 6LB assigns each new flow to **two pseudo-randomly chosen** application instances – by way of a consistent hashing scheme, described in section III.

When the flow reaches a candidate application instance, the corresponding segment in the SR header indicates that the virtual router may either forward the packet (*i.e.* start processing the next segment), or may directly deliver it to the virtual interface corresponding to the application instance. This purely local decision to accept query, or not, is based on a policy shared *only* between the virtual router and the application instance, running on the same compute node. To guarantee satisfiability, however, the penultimate segment indicates that the application instance must not refuse a query.

[2]Typically, a TCP SYN packet as part of a connection request.

Algorithm 1 Static Connection Acceptance Policy $\mathbf{SR_c}$

```
for each SYN packet do
    b ← number of busy threads
    if b < c or SegmentsLeft = 1 then
        SegmentsLeft ← 0, forward packet to application
    else
        SegmentsLeft ← SegmentsLeft − 1
        forward packet to next application instance in SR list
    end if
end for
```

Algorithm 2 Dynamic Connection Acceptance Policy $\mathbf{SR_{dyn}}$

```
accepted ← 0, attempt ← 0
c ← 1    ▷ or other initial value
ϵ ← 0.1    ▷ or other increment value
windowSize ← 50    ▷ or other window size
for each SYN packet with SegmentsLeft = 2 do
    attempt ← attempt + 1
    if attempt = windowSize then
        ▷ end of window reached, adapt c if needed and reset window
        if accepted/windowSize < 1/2 − ϵ and c < n then
            c ← c + 1
        else if accepted/windowSize > 1/2 + ϵ and c > 0 then
            c ← c − 1
        end if
        attempt ← 0, accepted ← 0
    end if
    SRc_policy()    ▷ use SRc policy with current value of c
    if SRc succeeded then
        accepted ← accepted + 1
    end if
end for
for each SYN packet with SegmentsLeft = 1 do
    SegmentsLeft ← 0, forward packet to application
end for
```

B. Connection Acceptance Policies

An application agent informs the virtual router if the local application instance wishes to accept a flow. The application agent may make this decision based on whatever information it has locally available – from the operating system, or from real-time application metrics, if exposed. If information is exchanged through shared memory, this incurs no system calls or synchronization, thus imposes a negligible run-time cost.

This section describes two simple policies for deciding if to accept new flows, or not. They assume that the application uses a standard master-slave thread architecture. Section VI-A will illustrate the application of these policies, in case of an HTTP server such as Apache. Table I summarizes the notation used throughout this paper to designate the different policies.

1) Static: With n worker threads in the application instance, and a threshold parameter c between 0 and $n+1$, Algorithm 1 describes a policy, $\mathbf{SR_c}$, where an application instance accepts the flow if and only if strictly less than c worker threads are busy (except for the last in the SR list, which must always accept). Thus, an application instance which is "too busy" will be assigned a connection only if all previous application instances in the list are also "too busy". The choice of the parameter c directly influences the global system behavior: small values of c yield better results under light loads, and high values yield better results under heavy loads. As extreme examples, when $c = 0$, all requests are satisfied by the last application instance of their SR lists; when $c = n + 1$, all requests are satisfied by the first: both cases reduce to a random load-balancing scheme. If the chosen value of c is too small as compared to the load, almost all connections are treated by the last application instance of their SR lists, and vice-versa.

If the typical load is known, the value of c can be configured statically – otherwise, a dynamic policy can be employed.

2) Dynamic: When the typical load is unknown, the policy $\mathbf{SR_{dyn}}$ adapts c to maintain a rejection ratio of each application instance of $\frac{1}{2}$, as detailed in Algorithm 2. Previous acceptance decisions are recorded over a fixed window of queries. When the end of the window is reached, if the number of accepted queries is significantly below (or above) $\frac{1}{2}$, the value of c is incremented (or decremented).

$\mathbf{SC}$	Single-Choice policy (baseline)
$\mathbf{SR_c}$	Static acceptance policy (Algorithm 1) with threshold c *e.g.* $\mathbf{SR_4}$ is the policy of Algorithm 1 with $c = 4$
$\mathbf{SR_{dyn}}$	Dynamic acceptance policy (Algorithm 2)

Table I
NOTATION

Protocol	new flow	pinned flow
IPv6 SR insert	72	56
IPv6 SR encap	96	80
IPv6 GRE Tunnel	88	44
IPv6 VXLAN Tunnel	140	70

Figure 5. Protocol overhead (in bytes) for different steering mechanisms, towards two (*new flow*) or one (*pinned flow*) application instances.

C. Protocol Overhead

Inserting an IPv6 SR header to steer a connection through multiple application instances has an impact in terms of packet size overhead. To quantify this, figure 5 depicts the number of extra bytes needed to steer a packet through two (*new flow*) or one (*pinned flow*) servers, for different protocols. As compared to other equivalent solutions allowing to steer a request through a set of instances (by sticking several successive tunneling headers), the proposed approach has the lowest overhead. After flow pinning, the overhead incurred by using SR to steer packets is of 12 bytes as compared to GRE.

D. Reliability

The solution described in this section (and more generally in this paper) focuses on the data-plane: it is assumed that a controller takes care of installing the mapping between a VIP and the set of addresses of servers capable of serving that VIP. Notably, as in other distributed load-balancing approaches [2], [7], the controller should take care of health-checking the backend servers, and removing them from the set of available servers if unresponsive.

Since connection establishment packets go through a chain of servers rather than a single one, the properties of 6LB when facing failures need to be considered. Two scenarios can be distinguished. First, if a whole machine goes down (*critical failure*), new flows whose first candidate application instance is hosted on this machine will fail to be established, and new flows whose second candidate application instance is hosted on this machine will fail only if the first instance rejected them. This incurs a $p_R\%$ failure overhead as compared to single-choice load-balancing approaches, where p_R is the

percentage of connections being rejected by a first instance (*e.g.* 50% with $\mathbf{SR_{dyn}}$). However, this happens only during the short amount of time before the controller detects that the machine is down and updates the backend pool on load-balancers accordingly[3]. Second, if an application instance goes down (crashes or becomes unresponsive) but the machine hosting it still remains up (*non-critical failure*), the virtual router on that machine will be able to forward connection establishment packets to the next instance in the SR list, for new flows whose first candidate instance is failing. Thus, for non-critical failures, 6LB increases the reliability of the system as compared to single-choice approaches, with a $(100-p_R)\%$ failure reduction.

III. Horizontal Scaling with Consistent Hashing

Elastic scaling of the number of load-balancer instances is required, in order to accomodate dynamic data center loads and configurations. When a load-balancer instance is added or removed, the ECMP function (see figure 1) performed by the edge router(s) may rebalance existing flows between remaining load-balancer instances. Thus, it is necessary to ensure that the mapping from flows to lists of candidate application instances is consistent across all load-balancers. This is achieved by the use of *consistent hashing*, depicted in figure 2 – which must also be resilient to modifications to the set of applications instances: adding or removing an application instance must have minimal impact on the mapping of existing flows.

Consistent hashing for load balancing is used *e.g.* in [2], which proposes an algorithm mapping an incoming flow to one application instance. This section introduces a new consistent hashing algorithm (generalizing the one from [2]) to allow each flow to map to an ordered list of application instances. Although 6LB uses 2 choices, the mechanism presented in this section is agnostic to this value, and is therefore presented for lists of C instances.

A. Generating Lookup Tables

With M *buckets* and N *application instances*, and where $N \ll M$, a pseudo-random permutation $p[i]$ of $\{0,\dots,M-1\}$ is generated for each application instance $i \in \{0,\dots,N-1\}$ – *e.g.* by listing the multiples of the i-th generator of the group $(\mathbb{Z}_M, +)$. These permutations are then used to generate a lookup table $t : \{0,\dots,M-1\} \to \{0,\dots,N-1\}^C$, mapping each bucket to a list of C application instances, following the procedure described in Algorithm 3. This table t is then used to assign SR lists of application instances to flows: each network flow will be assigned an SR list by hashing its network 5-tuple into a bucket j and taking the corresponding list $t[j]$.

Generating the lookup table t is done by browsing through the set of application instances in a circular fashion, making them successively "skip" buckets in their permutation until finding one that has not yet been assigned C application instances. Once each bucket has been assigned C application instances, the algorithm terminates. This process is illustrated in figure 6a, for $C = 2$ choices, with $N = 4$ application instances and $M = 7$ buckets. For each application instance i, the corresponding permutation table $p[i]$ is shown, where a circled number $\textcircled{j}_n$ means that bucket j has been assigned to that application instance at step n. For each bucket j, the lookup table $t[j]$ returned by the algorithm is also shown. For instance, bucket 3 is assigned to instance 1 (at step 5) and instance 2 (at step 6), thus the lookup table for bucket 3 is $(1, 2)$. The "skipping" behavior occurs *e.g.* at step 9, where bucket 5 is skipped in $p[1][j]$ because it was already assigned two application instances.

[3]A simple way to improve the reliability of the system during this short amount of time would be to monitor, in-band, the responsiveness of the servers (*e.g.* by gathering information about packet retransmissions or return traffic), and to rotate the order of SR lists for packets whose first instance is deemed unresponsive. This would increase reliability while ensuring that the browsed set of instances remains the one returned by consistent hashing.

Algorithm 3 Consistent Hashing

```
nextIndex ← [0, ..., 0]
C ← 2    ▷ or another size for SR lists
t ← [(−1, −1), ..., (−1, −1)]
n ← 0
while true do
    for i ∈ {0, ..., N − 1} do
        if nextIndex[i] = M then    ▷ permutation exhausted
            continue
        end if
        c ← p[i][nextIndex[i]]    ▷ advance in i's permutation
        ▷ skip buckets for which the SR list is already filled
        while t[c][C−1] ≥ 0 do
            nextIndex[i] ← nextIndex[i] + 1
            if nextIndex[i] = M then    ▷ permutation exhausted
                continue 2    ▷ continue the upper loop
            end if
            c ← p[i][nextIndex[i]]
        end while
        ▷ c is now the first bucket with SR list not filled
        choice ← 0
        while t[c][choice] ≥ 0 do
            choice ← choice + 1
        end while
        ▷ choice is now the first available position in the SR list
        t[c][choice] ← i
        nextIndex[i] ← nextIndex[i] + 1
        n ← n + 1
        if n = M × C then return t
        end if
    end for
end while
```

B. Analysis

1) Resiliency: Figure 6b illustrates how this scheme is resilient to changes to the pool of application instances, by showing how removing application instance 0 modifies the tables $t[j]$ from the example of figure 6a. Assuming that flows are assigned to the first or second application instance in their SR lists with equal probability (as with the $\mathbf{SR_{dyn}}$ policy), the question is how flows mapped to a non-removed application instance ($1, 2, 3$ in this example) are affected by the table recomputation. For each bucket, one failure is counted for each non-removed application instance appearing in the lookup table before recomputation, but not after. In the example of figure 6, the only failure is induced by bucket 4, as the second entry of its lookup table, 1, does not appear in its newly computed lookup table, $(3, 2)$. With 10 non-removed flows, the failure rate in this example is 10%.

Permutation tables $p[i]$ for each Application Instance i:

j	0	1	2	3	4	5	6
$p[0][j]$	④$_0$	⑤$_4$	⑥$_8$	0	1	②$_{12}$	3
$p[1][j]$	①$_1$	③$_5$	5	⓪$_9$	2	④$_{13}$	6
$p[2][j]$	⑤$_2$	③$_6$	①$_{10}$	6	4	2	0
$p[3][j]$	⑥$_3$	⓪$_7$	1	②$_{11}$	3	4	5

and lookup table t:

Bucket j	0	1	2	3	4	5	6
SR list $t[j]$	(3,1)	(1,2)	(3,0)	(1,2)	(0,1)	(2,0)	(3,0)

(a) Before removal of Application Instance 0

Permutation tables $p[i]$ for each Application Instance i:

j	0	1	2	3	4	5	6
$p[1][j]$	①$_0$	③$_3$	⑤$_6$	⓪$_9$	②$_{12}$	4	6
$p[2][j]$	⑤$_1$	③$_4$	①$_7$	⑥$_{10}$	④$_{13}$	2	0
$p[3][j]$	⑥$_2$	⓪$_5$	1	②$_8$	3	④$_{11}$	5

and lookup table t:

Bucket j	0	1	2	3	4	5	6
SR list $t[j]$	(3,1)	(1,2)	(3,1)	(1,2)	(3,2)	(2,1)	(3,2)

(b) After removal of Application Instance 0

Figure 6. Example permutation tables $p[i]$ and lookup table t ($C = 2, M = 7, N = 4$), before and after removal of application instance 0

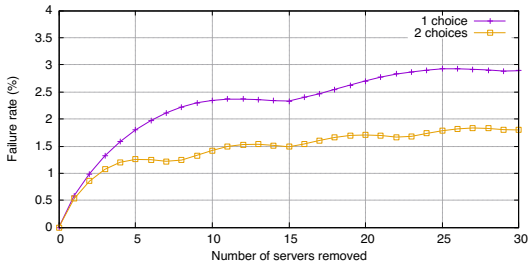


Figure 7. Resiliency of consistent hashing to application instance removals: 1 choice (no SR) vs 2 choices (6LB)

Intuitively, mapping flows to two application instances, instead of just to one, increases resiliency: it is less likely that the SR lists of a bucket before and after recomputation have empty intersection – for this to happen, a single bucket would need to be re-assigned twice.

The resiliency of algorithm 3 is studied by way of a simulation. An initial lookup table was computed. Then, k application instances were removed and the lookup table was recomputed – which allowed computing the previously introduced failure rate. The parameters were $N = 1000$ servers, $M = 65537$ buckets, and 20 experiments were performed, for each value of k from 0 to 30.

Figure 7 reports the failure rate as a function of the number of removed instances. First, with $C = 1$ (*i.e.* mapping each flow onto a single application server), results identical to figure 12 in [2] are obtained, confirming that the algorithm reduces to the algorithm from [2] in this case. Using algorithm 3 for mapping each flow to two application instances ($C = 2$) shows up to 44% fewer failures (when $k = 8$) – or, to put it differently, 44% fewer TCP connections being reset.

2) Fairness: Each application instance picks the same number of buckets (as first or second entry), except potentially one in the last round. Assuming a probability of acceptance of $\frac{1}{2}$ (as with $\mathbf{SR_{dyn}}$), this guarantees that traffic is equally spread between application instances. Note that a given application instance is not assigned the same number of first-choice buckets and second-choice buckets; this is nonetheless compensated by the fact that application instances do balance the load between themselves – this is evaluated in section VI-B4.

3) Complexity: If permutations $p[i]$ are randomly distributed, this algorithm is a variant of the *coupon collector's problem*, and is expected to terminate in $M \log M + \mathcal{O}(M)$ steps for $C = 1$ [32], and in $M \log M + M \log \log M + \mathcal{O}(M)$ steps for $C = 2$ [33]. Hence, choosing two, rather than one, application instances requires only $1 + \frac{\log \log M}{\log M} \leq 1.368$ times more steps.

In comparison to the naïve algorithm consisting in building two uncorrelated lookup tables for the first and second application instances in the SR list, the benefit of using Algorithm 3 is twofold: the generation time is smaller, and jointly building the two entries make the scheme more resilient to changes as shown in figure 7.

IV. In-band Stickiness Protocol

A load-balancer instance should, for each flow it handles, have knowledge of the application instance which has accepted the flow. First, this allows packets to be directly steered to the handling instance, without hopping through the chain of candidates. Second, this ensures that, when the consistent hashing table is recomputed (*e.g.* due to changes in the pool of applications), existing connections are protected against potential changes in the lookup table.

Thus, a signaling mechanism is required between the load-balancer and the application instances. Four properties should be satisfied: (i) no external control traffic should be generated, (ii) deep packet inspection should be minimized, (iii) incoming packets should go directly to the application instance handling the flow, and (iv) outgoing packets should not transit through the load-balancer.

To satisfy (i) and (ii), SR headers are inserted into packets part of the accepted flow, *i.e.* a set of SR *functions* are used for communicating between the load-balancer and the application instance. Objective (iii) is accomplished by having the appliation instance signal to the load-balancer when it has accepted a flow (*i.e.* by adding an SR header to, typically, the TCP SYN+ACK), and (iv) by making other traffic (*i.e.* packets other than, typically, the TCP SYN+ACK) bypass the load-balancer and be sent directly from the application instance to the client.

A. SR Functions

SR functions are used to encode actions, which are to be taken by a node, directly in the SR header. This is closely linked to how IPv6 addresses are assigned: since each compute node is assigned a (typically, 64 bit [34]) IPv6 prefix, it is possible to use the lower-order bytes in this prefix to designate different functions, as recommended by the SR draft specification [29]. These functions will also depend on the address of the first segment in the SR list (the "sender" of the function). In practice, when a node whose physical prefix is s receives a packet with SR header $(x, \ldots, s::f, \ldots)$, it will trigger a function f with argument x, which will be denoted by `s.f(x)`. In terms of a state machine, each SR function will thus (i) move the node from one state to another and (ii) trigger an action on the packet containing the SR function.

B. Handshake Protocol

When a (TCP) flow is initiated, SR functions are added to the TCP handshake, so as to inform the load-balancer candidate application instance has accepted the flow - thus establishing a handshake protocol between the load-balancer and the application instance handling a flow. This handshake protocol is formally described as state machines in tables II and III (in the appendix), and detailed below:

1. Upon receipt of a flow (typically, a TCP SYN packet) from a client c for an application whose VIP is d, the load-balancer lb will insert an SR header $(lb, s_1 :: ca, s_2 :: cf, d)$ comprising the physical addresses of the two candidate application instances s_1, s_2 as given by the hashing function, and the original VIP. The suffix ca in the addresses indicate a function `connectAvail`, whereas cf represents a function `connectForce`. The first application instance in the list will make a local decision on whether to accept the flow. In case of refusal, the packet will be forwarded to the second application instance, which will have to forcefully accept the flow.

2. The application instance s_i ($i \in \{1, 2\}$) that has accepted the connection enters a *waiting* state for this flow. While in this state, it will temporarily steer traffic from the application towards the load-balancer, so that the latter can learn which application instance has accepted the connection. To do so, it inserts an SR header $(s_i, lb :: cs, c)$ in packets coming from the application, where cs is the `createStickiness` function.

3. Upon receipt of such a packet, the load-balancer enters a *steering* state, during which traffic from the client to the application is sent using $(lb, s_i :: as, d)$ as an SR header, as standing for a function `ackStickiness`. This permits both steering the traffic directly to the correct application instance, and acknowledging the creation of a stickiness entry.

4. Then, when s_i receives such a packet, it enters a *direct return* state for this flow. As s_i has acknowledged the creation of the stickiness entry on the load-balancer, it thus does not need to send traffic through it anymore. Subsequent traffic sent by s_i will therefore be sent directly towards the client, without using SR.

5. Finally, when s_i receives a connection termination packet from the application (typically, a TCP FIN or RST), it will insert an SR header $(s_i, lb :: rms, c)$, where rms designates a `removeStickiness` function. This allows explicitly signaling connection termination to the load-balancer. When receiving this packet, the load-balancer will start a small timer, at the expiration of which it will remove the corresponding stickiness entry – using a small timer ensures that packets are correctly steered to the rightful application instance while the transport layer connection teardown is happening. In addition to this explicit connection termination process, periodic garbage collection is used to remove stale entries from the load-balancer.

C. Failure Recovery

When adding or removing a load-balancer instance, traffic corresponding to a given flow might be redirected to a different load-balancer instance from the one over which it was initiated.

In order to recover state, when a new load-balancer instance receives a flow for which it does not have any state, incoming data packets corresponding to an unknown flow are added an SR header $(lb, s_1 :: rs, s_2 :: rs, d)$, where rs is an SR function `recoverStickiness`. Consistent hashing ensures that $\{s_1, s_2\}$ is the same instance set as the one used by the previous load-balancer. When receiving a packet for this SR function, the application instance that, in the past, has accepted the flow will re-enter the *steering* state, so as to notify the load-balancer. Conversely, an application instance that has previously not accepted the flow will simply forward the packet to the next application instance in the SR list.

V. Performance Analysis

In this section, an analytic model describing the performance of 6LB with the $\mathbf{SR_c}$ policy (Algorithm 1) is derived. By way of this model, 6LB is compared to a **S**ingle-**C**hoice random flow assignment approach (**SC**), which reflects the behavior of standard consistent-hashing approaches (*e.g.* [2]).

A. System Model

It is assumed that the system contains N application instances, with $N \to +\infty$, and that consistent hashing uses enough buckets such that the SR list associated with a flow is uniformly chosen amongst the N^2 possible lists.

In this section, the expected response time for the static acceptance policy $\mathbf{SR_c}$ described in Algorithm 1 is derived, for c an integer threshold parameter. While a similar model has been formulated in [35], the contribution in this paper is that the model developed allows deriving the associated expected response time, the fairness index, and the response time distribution. This paper also validates the model against a real deployment experiment.

Incoming flows are assumed distributed according to a Poisson process of rate $\Lambda = N\lambda$, and each application instance offers an exponentially distributed response time, with a processing rate normalized to $\mu = 1$. For stability, the arrival rate must verify $\lambda < 1$. For $i \geq 0$, s_i is the fraction of application instances for which there are i or more pending flows (with $s_0 = 1$). This allows writing (as in [35]):

$$\begin{cases} \frac{\mathrm{d}s_i}{\mathrm{d}t} = \lambda(1 + s_c)(s_{i-1} - s_i) - (s_i - s_{i+1}), \forall 1 \leq i \leq c \\ \frac{\mathrm{d}s_i}{\mathrm{d}t} = \lambda s_c(s_{i-1} - s_i) - (s_i - s_{i+1}), \forall i > c \end{cases} \tag{1}$$

When $i \leq c$, the probability of a flow being sent to an application instance which already handles $(i-1)$ other flows (i) directly is $(s_{i-1} - s_i)$ and (ii) after having being rejected by an application instance which already handles c or more flows is $s_c(s_{i-1} - s_i)$. This yields a total probability of $(1 + s_c)(s_{i-1} - s_i)$. The same reasoning applies for $i > c$, where the probability of a flow being sent to an application instance which already handles $(i-1)$ flows ($i \geq 1$) is the probability of having been rejected by a first application instance already handling c or more flows, before having been sent to an accepting application instance, yielding $s_c(s_{i-1} - s_i)$. The probability of a flow leaving an application

instance already handling i flows is $(s_i - s_{i+1})$. Since the per-application-instance arrival rate is λ and the processing rate is $\mu = 1$, this yields equation (1).

To study the behaviour of the system once in equilibrium, it is necessary to find a fixed point to the differential system (1). Setting $\frac{\mathrm{d}s_i}{\mathrm{d}t} = 0$ yields the following system of equations (where $s_0 = 1$):

$$\begin{cases} 0 = \lambda(1 + s_c)(s_{i-1} - s_i) - (s_i - s_{i+1}), \forall 1 \leq i \leq c \\ 0 = \lambda s_c(s_{i-1} - s_i) - (s_i - s_{i+1}), \forall i > c \end{cases} \tag{2}$$

For $1 \leq n \leq c$, summing equation (2) over $i = n$ to $+\infty$ gives $s_n = \lambda(1 + s_c)(s_{n-1} - s_c) + \lambda s_c^2$, or: $s_n = \lambda[s_{n-1}(1 + s_c) - s_c]$. Since $s_0 = 1$, this gives $s_1 = \lambda$. More generally, solving this recursion yields:

$$s_n = \frac{(1 - \lambda)(\lambda(1 + s_c))^n - \lambda s_c}{1 - \lambda(1 + s_c)}, \forall 1 \leq n \leq c \tag{3}$$

Then, for $n > c$, summing equation (2) over $i = n$ to $+\infty$ gives $s_n = \lambda s_c s_{n-1}$, *i.e.* :

$$s_n = (\lambda s_c)^{n-c} s_c, \forall n \geq c \tag{4}$$

Plugging $n = c$ into equation (3) allows formulating an implicit equation for s_c:

$$s_c(1 - \lambda s_c) = (1 - \lambda)[\lambda(1 + s_c)]^c \tag{5}$$

This polynomial equation can be solved explicitly for $c \leq 5$, and numerically otherwise. Using this solution in equations (3) and (4) allows obtaining the value of s_n for any n, *i.e.* the distribution of the number of flows awaiting being handled by an arbitrary application instance.

B. Expected Response Time

The expected number X of flows in the system can be computed, given that the probability that an application instance handles n flows is $(s_n - s_{n+1})$: $\mathbb{E}(X) = N \sum_{n=0}^{+\infty} n(s_n - s_{n+1}) = N \sum_{n=1}^{+\infty} s_n$.

According to Little's law [36], the expected time T spent in the system by a flow is: $\mathbb{E}(T) = \frac{\mathbb{E}(X)}{\Lambda} = \frac{1}{\lambda} \sum_{n=1}^{+\infty} s_n$. Summing s_n, as obtained in equations (3) and (4), and using equation (5) gives:

$$\sum_{n=1}^{c-1} s_n = \frac{\lambda - \lambda s_c(c-1) - s_c}{1 - \lambda(1 + s_c)}, \sum_{n=c}^{+\infty} s_n = \frac{s_c}{1 - \lambda s_c}$$

Hence:

$$\mathbb{E}(T) = \frac{1 - c s_c(1 - \lambda s_c) - \lambda s_c(1 + s_c)}{(1 - \lambda s_c)(1 - \lambda(1 + s_c))} \tag{6}$$

Figure 8a depicts the value of $\mathbb{E}(T)$ for different $\mathbf{SR_c}$ policies, and for $\lambda \in [0, 1)$. As a reference, this value is compared to $\frac{1}{1-\lambda}$, the expected response time for $\mathbf{SC}$, when clients are randomly assigned to one server. It can be observed that the $\mathbf{SR_c}$ policies uniformly yield an improvement over $\mathbf{SC}$. When c is small, lower values of λ yield the highest gain; when c is important, higher values of λ yield the highest gain. For example, choosing $\mathbf{SR_8}$ offers an improvement over $\mathbf{SR_4}$ only when $\lambda \geq 0.983$, and thus might rarely be suitable.

C. Additional Forwarding Delay

In cases where the server-to-server forwarding delay is significant as compared to the job duration, forwarding a query to the second application instance in an SR list incurs an additional cost. If $\delta > 0$ denotes the network delay, multiplying this by the probability of being rejected by a first application instance gives the expected additional delay. Thus, the response time including delay, $\hat{T}$, verifies:

$$\mathbb{E}(\hat{T}) = \mathbb{E}(T) + \delta \times s_c \tag{7}$$

The following theorem states the conditions under which $\mathbf{SR_c}$ does not degrade performance as compared to $\mathbf{SC}$:

Theorem 1. *As long as the network delay δ is smaller than the job duration $1/\mu$, the response time including delay with $\mathbf{SR_c}$ is better than with $\mathbf{SC}$:* $\mathbb{E}(\hat{T}) \leq \frac{1}{1-\lambda}, \forall \delta \leq 1$.

The proof is given in the appendix. Figure 8d gives the expected response time including delay, in the "worst-case" in which the network delay equals the job duration ($\delta = 1$).

D. Fairness Index

Jain's fairness index [37], defined as $F = \frac{\mathbb{E}(X)^2}{\mathbb{E}(X^2)} \in [0, 1]$, is a measure for how even a load is distributed in a system: the closer it is to one, the more evenly is the load distributed.

Computing F requires computing $\mathbb{E}(X^2)$, the second moment of the number of flows in the system:

$$\mathbb{E}(X^2) = N \sum_{n=0}^{+\infty} n^2(s_n - s_{n+1}) = N \sum_{n=1}^{+\infty} (2n - 1) s_n$$

Using equations (3), (4) and (5), this yields:

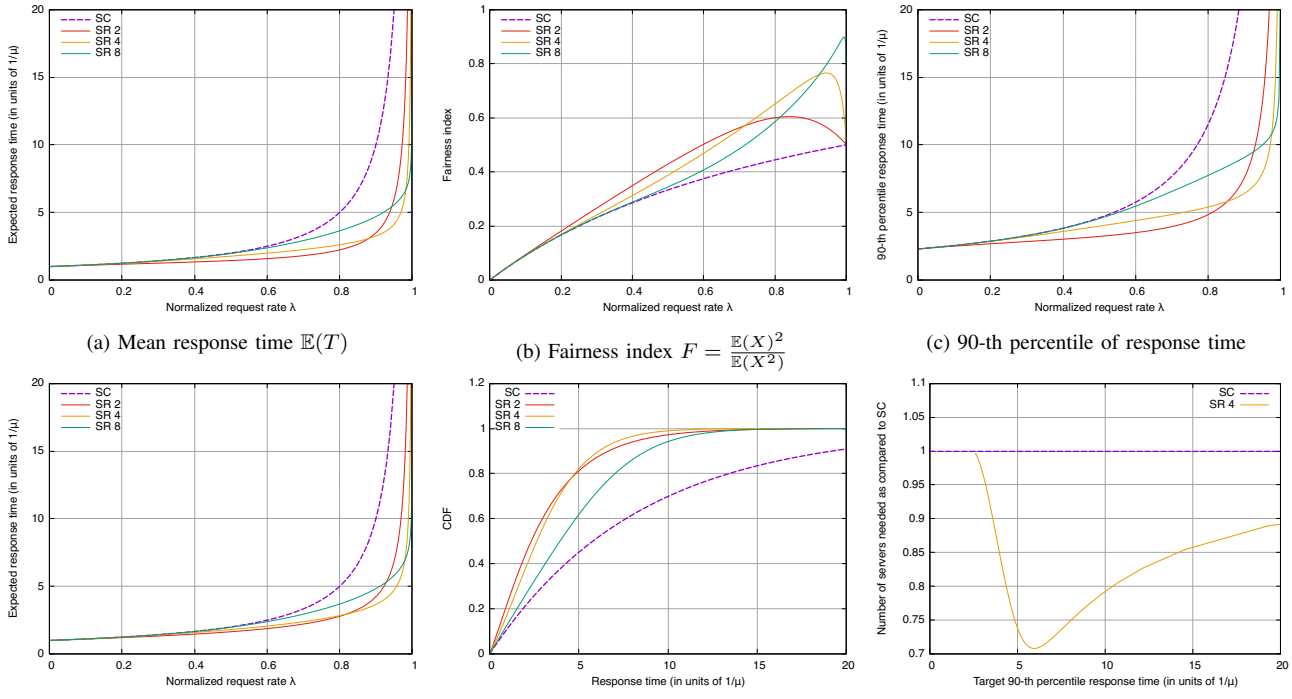
$$\sum_{n=1}^{c-1} (2n-1)s_n = \frac{1}{(1-\lambda(1+s_c))^2} \Big[\lambda^2(s_c+1)((c-1)^2 s_c + 1) + \lambda s_c(2cs_c - 3s_c + 2c - c^2 - 2) - 2cs_c + \lambda + s_c\Big],$$

$$\sum_{n=c}^{+\infty} (2n-1)s_n = \frac{s_c(c(2 - 2\lambda s_c) + 3\lambda s_c - 1)}{(1 - \lambda s_c)^2}$$

Combining those expressions with the expression for $\mathbb{E}(X)$ from section V-B allows to compute F. Figure 8b depicts F, for different $\mathbf{SR_c}$ policies, and for $\lambda \in [0, 1)$ – and compares with $\frac{\lambda}{1+\lambda}$, the fairness index for the $\mathbf{SC}$ policy. It can be observed that $\mathbf{SR_c}$ policies provide a better fairness than the reference $\mathbf{SC}$ policy, and that low values of c are more suitable for a low rate of new flow arrivals whereas high values of c are preferable for higher rates of new flow arrivals.

E. Wrongful Rejections

As has been shown in section V-C, using the $\mathbf{SR_c}$ policy yields better performance than $\mathbf{SC}$, because an overloaded application instance can offload a query to another random instance. However, the proposed mechanism is not fully equivalent to the canonical "power-of-two-choices" scheme [5], wherein the least loaded of two random instances is chosen. Suboptimal decisions happen when a first instance handling $n \geq c$ flows rejects the connection to a second instance



(a) Mean response time $\mathbb{E}(T)$

(b) Fairness index $F = \frac{\mathbb{E}(X)^2}{\mathbb{E}(X^2)}$

(c) 90-th percentile of response time

(d) Worst-case response time with delay $\mathbb{E}(\hat{T})$

(e) CDF of response time for $\lambda = 0.88$

(f) Reduction in number of servers when using $\mathbf{SR_4}$ vs $\mathbf{SC}$, for different 90-th percentile SLAs

Figure 8. Performance analysis of 6LB: $\mathbf{SC}$ vs $\mathbf{SR_2}$, $\mathbf{SR_4}$ and $\mathbf{SR_8}$.

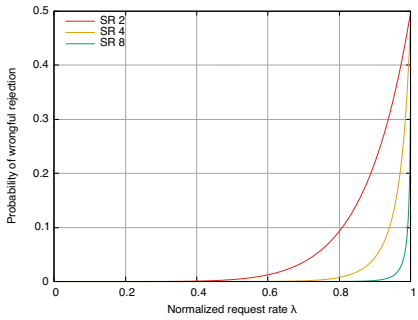


Figure 9. Probability of *wrongful rejection* for different $\mathbf{SR_c}$ policies.

with strictly more than n flows. It is possible to estimate the quantity of such *wrongful rejections*: the probability of hitting such a pair of instances is $(s_n - s_{n+1})s_{n+1}$. Using equation (4), the probability p_w of wrongful rejection can be expressed as:

$$p_w = \sum_{n=c}^{+\infty}(s_n - s_{n+1})s_{n+1} = \frac{\lambda s_c^3}{1+\lambda s_c} \tag{8}$$

In order to quantify this, figure 9 shows the probability of wrongful rejection for different $\mathbf{SR_c}$ policies. For example, with $\mathbf{SR_4}$, wrongful rejections happen with probability lower than 4.5% when $\lambda \leq 0.9$.

F. Response Time Distribution

The model also allows deriving the distribution of the time that a flow exists in the system. Knowing this distribution allows, for example, characterizing the performance of 6LB for Service Level Agreement (SLA) metrics, of the form *"No more than $x\%$ of clients experience a response time $\geq y$"*.

The distribution of the time T a flow waits, will be derived by computing its characteristic function $\varphi_T(\theta) = \mathbb{E}(e^{i\theta T})$ (for $\theta \in \mathbb{R}$). Assume that the system is at its equilibrium given by equation (2), and that application instances use a FIFO policy with exponential response times. When a flow is being directed to an application instance that is already handling $(k-1)$ flows ($k \geq 1$), its waiting time will be distributed as the sum of k independent and identically distributed (i.i.d.) exponential random variables $(\mathcal{E}_1, \ldots, \mathcal{E}_k)$ of parameter $\mu = 1$. The characteristic function of one such variable is $\mathbb{E}(e^{i\theta\mathcal{E}_1}) = \frac{1}{1-i\theta}$, hence $\mathbb{E}(e^{i\theta T}|k-1 \text{ clients}) = \left(\frac{1}{1-i\theta}\right)^k$.

When a flow arrives at the system, it will, with probability $(1_{k\leq c} + s_c)(s_{k-1} - s_k) = \frac{1}{\lambda}(s_k - s_{k+1})$, be directed to an application instance which is already handling $(k-1)$ other flows. Based on this, it is possible to express the characteristic function of the waiting time of an arbitrary flow:

$$\mathbb{E}(e^{i\theta T}) = \sum_{k=1}^{+\infty}\frac{1}{\lambda}(s_k - s_{k+1})\left(\frac{1}{1-i\theta}\right)^k$$

Using equations (3), (4) and (5), this can be expressed as:

$$\mathbb{E}(e^{i\theta T}) = \frac{(1-\lambda)(1+s_c)}{1-\lambda(1+s_c)-i\theta} - \frac{s_c(1-\lambda s_c)}{(1-\lambda s_c - i\theta)(1-\lambda(1+s_c)-i\theta)(1-i\theta)^{c-1}} \tag{9}$$

which can be inverted to find p_T, the probability density of T, using $p_T(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-i\theta t}\mathbb{E}(e^{i\theta T})\mathrm{d}\theta$.

Figure 8e depicts the CDF of this probability distribution, for $\lambda = 0.88$, and for various $\mathbf{SR_c}$ policies. At this high load,

the distribution for the $\mathbf{SR_c}$ policies exhibit lower response times and less variance as compared to $\mathbf{SC}$.

Integrating this probability density allows finding π_x, the x-th percentile of response time, defined as the number satisfying:

$$Pr(T \leq \pi_x) = \int_0^{\pi_x} p_T(t)\mathrm{d}t = \frac{x}{100} \tag{10}$$

Figure 8c depicts π_{90}, the 90-th percentile of response time, for various $\mathbf{SR_c}$ policies and for $\lambda \in [0, 1)$, and is compared to $\frac{\ln(10)}{1-\lambda}$, the same metric for the $\mathbf{SC}$ policy. Similarly as in figure 8a, the response time with $\mathbf{SR_c}$ is lower than with $\mathbf{SC}$, and small values of c are more suitable for low request rates.

G. Reducing the Number of Servers

The developed model allows estimating the gain, in terms of how many fewer application instances are required to attain a certain SLA, when using 6LB as compared to when using "plain" $\mathbf{SC}$.

Assuming that a system faces a daily request rate profile with a peek rate Λ_0, and the goal is to provide a given SLA μ_0 on the 90-th percentile of response time: no more than 10% of clients should receive a target response time greater than μ_0 (*i.e.* $\pi_{90} = \mu_0$).

With a simple $\mathbf{SC}$ load-balancer, the system faces a normalized request rate of $\lambda_0 = \Lambda_0/N$, and the 90-th percentile of response time is $\pi_{90} = \frac{\ln(10)}{1-\Lambda_0/N}$ – thus, requiring deploying $N = \frac{\Lambda_0}{1-\ln(10)/\mu_0}$ application instances to meet the SLA.

As per equation (10), let $\pi(\lambda)$ be the function giving the 90-th percentile of response time π_{90} as a function of λ, when using 6LB with the $\mathbf{SR_c}$ policy. In order to meet the SLA, *i.e.* to ensure that $\pi(\Lambda_0/N) = \mu_0$, $N' = \frac{\Lambda_0}{\pi^{-1}(\mu_0)}$ application instances must be deployed.

Comparing $\mathbf{SC}$ and 6LB with $\mathbf{SR_c}$ yields:

$$\frac{N'}{N} = \frac{1 - \ln(10)/\mu_0}{\pi^{-1}(\mu_0)} \tag{11}$$

Figure 8f depicts this reduction in number of servers, as a function of the target SLA μ_0, between $\mathbf{SC}$ and $\mathbf{SR_4}$. If the SLA requires that no more than 10% of clients experience a response time greater than *e.g.* $\mu_0 = 6$, then if that is met by a deployment of *e.g.* $N = 100$ application instances when using $\mathbf{SC}$, only $N' = 71$ application instances are required if using 6LB with the $\mathbf{SR_4}$ policy.

VI. Evaluation

A. Experimental Platform

The experimental platform used for evaluating 6LB is composed of a load-balancer and a server agent for the Apache HTTP server.

1) Load-Balancer: The load-balancer performing consistent hashing, SR header insertion and flow steering is implemented as a VPP plugin [31]. Having kernel-bypass capabilities and embedding an IPv6 Segment Routing stack, VPP is a suitable choice to build a performing implementation. As a reference, Maglev [2] was also implemented to evaluate the single-choice consistent hashing flow assignment policy $\mathbf{SC}$.

2) Apache HTTP Server Agent: A server agent for the Apache HTTP server [38] has been implemented as a VPP plugin, accessing Apache's *scoreboard* shared memory[4] to allow the virtual router to access the state of the application instance. Apache uses a *worker thread* model: a pool of worker threads is started in advance, and received queries are dispatched to those threads. Thus, a simple exposed metric is the state of each worker thread, allowing to count the number of busy/idle threads, and use this to decide on connection acceptance, using one of the policies described in section II-B.

3) System platform: The experiments, described in sections VI-B and VI-C, are conducted on a common platform. An edge router and two load-balancer instances are deployed as 2-core VMs residing in one physical machine. $N = 48$ application instances of an Apache HTTP server reside each in a 2-core VM, all of which are hosted across 4 physical machines (distinct from the one hosting the edge router/load-balancers). The edge router is configured to split traffic for the application across the two load-balancer instances, by way of ECMP, as in figure 1. VPP instances running in the edge router VM, in the load-balancer VMs, and in each of the VMs of the application instances, are on the same Layer-2 link, with routing tables statically configured. Each physical machine has a 24-core Intel Xeon E5-2690 CPU.

The size of the consistent hashing table of the load-balancer instances was set to $M = 65536$ (except for the experiments of sections VI-B4 and VI-B5), and the Apache servers were configured to use the `mpm_prefork` module, each with 32 worker threads and with a TCP backlog of 128.

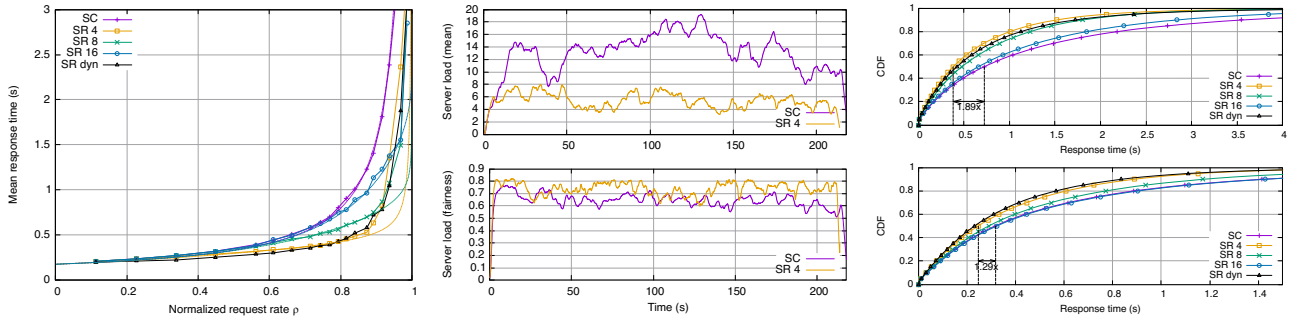
The `tcp_abort_on_overflow` parameter of the Linux kernel was enabled, triggering a TCP RST when the backlog of TCP connections exceeds queue capacity, rather than silently dropping the packet and waiting for a SYN retransmit. Thus under heavy load, it is application response delays that are measured, and not possible TCP SYN retransmit delays.

B. Poisson Traffic

1) Traffic and Workload Patterns: To evaluate the efficiency of the connection acceptance policies from section II-B under different loads, 6LB was tested against a simple CPU-intensive web application, consisting of a PHP script running a `for` loop with an exponentially distributed number of iterations, and whose duration is 190 ms in average. Using such a distribution ensures that job durations exhibit reasonable variance (the standard deviation of the exponential distribution is equal to its mean). A traffic generator sends a Poisson stream of queries (HTTP requests), with rate λ. A bootstrap step consisted of identifying λ_0, the max rate sustainable by the 48-servers farm, *i.e.* the smallest value of λ for which some TCP connections were dropped.

2) Connection Acceptance Policies Evaluation: With $\rho = \lambda/\lambda_0$ as the normalized request rate, for 20 values of ρ in the range $(0, 1)$, a Poisson stream of 80000 queries with rate ρ was injected in the load-balancers, using the policies $\mathbf{SR_4}$, $\mathbf{SR_8}$, $\mathbf{SR_{16}}$, $\mathbf{SR_{dyn}}$. As baseline, the same tests were run with

[4]This shared memory, internal by default, can be exposed as a named file by specifying the `ScoreBoardFile` directive in the server configuration.

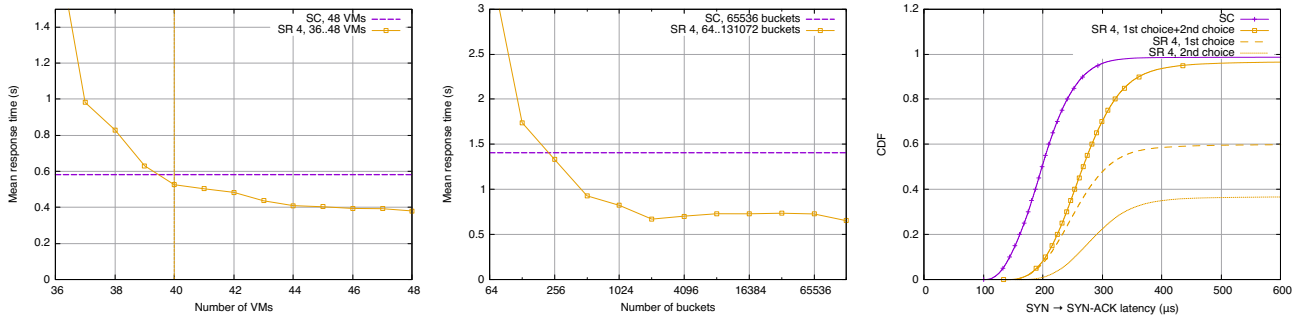


(a) Average page load time

(b) Instantaneous server load, $\rho = 0.89$.

(c) CDF of page load time: $\rho = 0.71$ (top) and $\rho = 0.89$ (bottom).

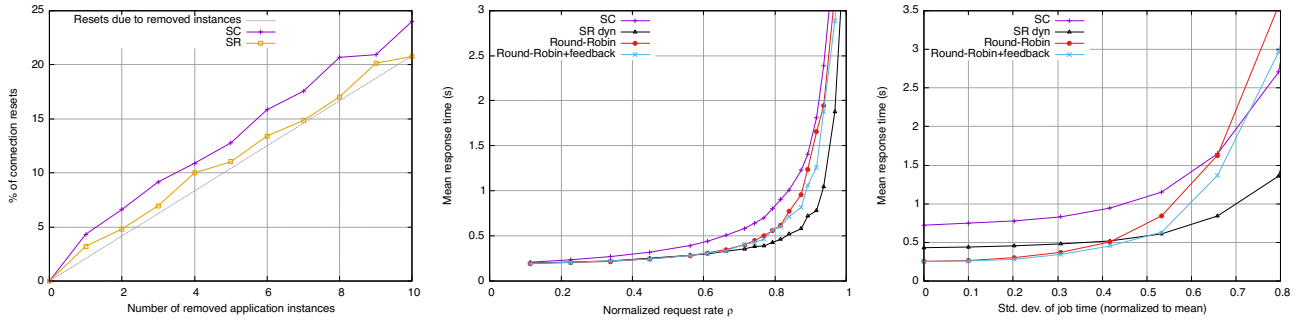
Figure 10. Connection acceptance policies evaluation: $\mathbf{SC}$ vs $\mathbf{SR_4}$, $\mathbf{SR_8}$, $\mathbf{SR_{16}}$, $\mathbf{SR_{dyn}}$.



(a) Average page load time while decreasing the number of VMs: $\mathbf{SR_4}$, $\rho = 0.71$.

(b) Influence of the consistent hashing table size M: $\mathbf{SR_4}$, $\rho = 0.89$.

(c) SYN → SYN-ACK latency, $\rho = 0.71$.



(d) Connection resets when removing x application instances and simultaneously switching to another 6LB instance

(e) Average page load time for centralized policies

(f) Average page load time for different variances of job times, $\rho = 0.71$.

Figure 11. Poisson workload evaluation.

a policy $\mathbf{SC}$ where queries are pseudo-randomly assigned to one server, without Service Hunting, using the single-choice consistent hashing algorithm of Maglev [2].

Figure 10a depicts mean response times for each tested request rate and for each policy, and show that, among those, $\mathbf{SR_4}$ yields the best response time profile, up to $2.3\times$ better than $\mathbf{SC}$ for $\rho = 0.87$. $\mathbf{SR_8}$ and $\mathbf{SR_{16}}$ likewise perform better than $\mathbf{SC}$ for all loads, but with a lesser impact. $\mathbf{SR_{dyn}}$ offers results close to the best tested static policy. In order to validate the analytical model introduced in section V, the response time as obtained from equation (6) is displayed in dotted lines alongside the experimental results[5]: it can be observed that the model accurately fits the data, as long as $\rho < 0.9$. After that, the assumptions (steady state, infinite number of servers) do not hold anymore.

Figure 10c shows the CDF of the page response time for the 80000 queries batch with $\rho = 0.89$, for each policy. $\mathbf{SC}$ exhibits a very dispersed distribution of response times, whereas the different $\mathbf{SR_c}$ policies yield lower, and less dispersed, response times. This can be explained by inspecting the evolution of the mean instantaneous load (the number of busy worker threads) over all servers, as well as the corresponding fairness index: $\frac{(\sum_{i=1}^{48} x_i(t))^2}{48 \sum_{i=1}^{48} x_i(t)^2}$ (where $x_i(t)$ is the load of server i at time t), depicted in figure 10b[6]. As $\mathbf{SR_4}$ better spreads queries between all servers (the fairness

[5]A fit is performed on $\mathbf{SC}$ to rescale the units. The obtained scaling coefficients are then used for all policies.

[6]These values have been smoothed through an Exponential Window Moving Average filter, of parameter $\alpha = 1 - \exp(-\delta t)$ where δt is the interval of time in seconds between two successive data points.

index is closer to 1), and servers are individually less loaded, better response times result.

For lighter loads, a similar behavior can be observed, except that high $\mathbf{SR_c}$ policies exhibit no benefits as compared to $\mathbf{SC}$. Figure 10c shows the CDF of the page load time for an experiment where $\rho = 0.71$: $\mathbf{SR_{16}}$ yields no improvement over $\mathbf{SC}$, and $\mathbf{SR_8}$ yields a relatively small improvement, however the $\mathbf{SR_4}$ policy provides a substantial improvement in response times – and $\mathbf{SR_{dyn}}$ remains able to successfully match $\mathbf{SR_4}$, the best tested static policy.

3) Reducing the Number of Servers: Previous experiments have shown how 6LB is able to yield a reduced page response time, for a given request rate. Conversely, if an SLA on the target response time is to be satisfied, 6LB can be used to decrease the number of servers needed to reach that SLA. In order to quantify this, a simple experiment has been conducted to find out how many VMs can be shut off while achieving a pre-defined SLA. Assume that the 48 VMs were deployed with $\mathbf{SC}$ to attain an average response time of 0.58 s, *i.e.* that the application faces a total request rate of $\rho = 0.71$ (values taken from figure 10a). Using the same request rate, a batch of 80000 requests was ran against less and less VMs with the $\mathbf{SR_4}$ policy, until the same average response time was reached. Figure 11a shows the average response time as a function of the number of VMs: with 6LB 40 VMs are needed to meet the same SLA as compared to 48 VMs with $\mathbf{SC}$ – a reduction of 17%.

4) Influence of the Consistent Hashing Table Size: Using a smaller hash table can be beneficial in environments with tight resources, but at the cost of evenness in the distribution of the application instances within first segments of the SR lists (as explained in section III). In order to quantify this, a Poisson stream of 80000 requests with request rate $\rho = 0.71$ was sent to the load-balancers against the $\mathbf{SR_4}$ policy, using different hash table sizes. Figure 11b shows the average response time as a function of the table size used (tables have sizes 2^k for performance reasons). The response times are almost identical for high table sizes, with a noticeable influence when $M \leq 1024$. Also, except when $M \leq 128$, the average response time stays lower than when using the $\mathbf{SC}$ policy with the same rate.

5) Consistent Hashing Resiliency: The resiliency of the consistent hashing mechanism introduced in Algorithm 3 in real conditions is evaluated through a simple experiment, where a simultaneous change in the application instances pool and in the load-balancer pool is introduced. With $M = 4096$ buckets in the consistent hashing tables, 1000 long-lived flows are injected in the system and handled by the first 6LB instance. Then, x application instances are removed, while at the same time the ECMP router is reconfigured to use the second 6LB instance. The number of connection resets is recorded for $\mathbf{SR_8}$ and $\mathbf{SC}$, and depicted in figure 11d for several values of x (averaged over 10 experiments). 6LB increases the resiliency over $\mathbf{SC}$ (as described in section III-B1): apart from "unavoidable" resets corresponding to connections that were pinned to a removed instance, no more than 2% of extra connections were reset by 6LB, as compared to 4% with $\mathbf{SC}$.

6) SYN → SYN-ACK Latency: In order to quantify the additional forwarding latency induced by 6LB, figure 11c depicts the SYN → SYN-ACK latency as seen by the client for $\mathbf{SR_4}$ and $\mathbf{SC}$, for the experiment where $\rho = 0.71$. As compared to $\mathbf{SC}$, with 6LB, the SYN packet *can* be forwarded to an extra server, and the SYN-ACK packet *must* be forwarded through the load-balancer. Overall, this increases the median latency by 69 μs. Restricted to those connections that are accepted by the second server in the SR list (corresponding to 38% of the 80000 queries in this experiment), the median latency is increased by 57 μs. For connections accepted in first instance, the median latency is increased by 32 μs.

7) Comparison against Centralized Policies: Centralized load-balancing policies do not offer the resiliency of consistent hashing approaches, but in exchange provide more fairness. In order to position 6LB as compared to this class of load-balancers, two centralized policies are evaluated: (i) Round-Robin and (ii) weighted Round-Robin with feedback. With the latter policy, feedback is obtained by polling the load of each application instance every 200 ms (over an out-of-band TCP channel), before adjusting the weight of the instance in the Round-Robin algorithm accordingly[7]. Figure 11e depicts the average page load time as a function of the request rate ρ. Results show that Round-Robin provides more fairness than single-choice consistent hash, but is outperformed by $\mathbf{SR_{dyn}}$ (with equivalent results for light loads $\rho \leq 0.7$). For heavier loads, the feedback policy slightly improves performance over Round-Robin, but remains outperformed by $\mathbf{SR_{dyn}}$: this shows the benefit of using instantaneous information rather than relying on periodic feedback.

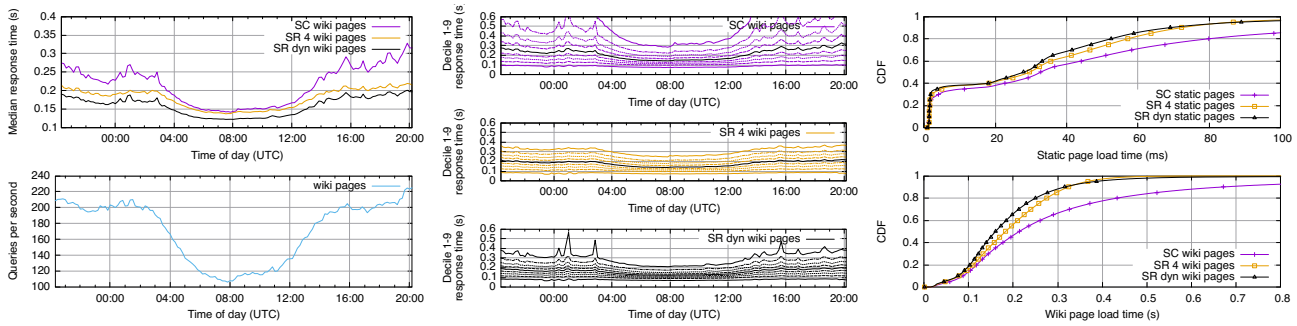
8) Influence of the Variance of Service Times: To understand the influence of the variability of job service times, an experiment is conducted, where job CPU times distributions have different variances. To that purpose, the previously used exponential distribution is replaced with several log-normal distributions (a simple class of positive-valued distributions with a parameter influencing the variance). The response times are set to have the same median as previously, but different variance parameters, allowing to evaluate from constant to very skewed response times. Figure 11f depicts the mean response time as a function of the standard deviation of service times, for a Poisson stream of 80000 queries at rate $\rho = 0.71$, against $\mathbf{SR_{dyn}}$, $\mathbf{SC}$ and Round-Robin.

In the extreme case where response times are constant, Round-Robin performs the best (as instances will have totally processed a query before being assigned a new one) and 6LB performs better than $\mathbf{SC}$. Indeed, with single-choice consistent-hashing queries can be placed on a server that is already busy (if "unlucky once"), whereas 6LB needs to be "unlucky twice" for this to happen. When the skewness of job service times increases, 6LB's use of local information becomes a greater and greater advantage, and it eventually shows the best performance among all approaches.

C. Wikipedia Replay

To evaluate the efficiency of 6LB when exposed to a realistic workload, an experiment has been constructed to reproduce

[7]The weight w is adjusted with $w = 0.1 + 0.9\exp(-8 \times (b/32)^2)$, where b is the current number of busy worker threads.



(a) Query rate and median wikipage load time. (b) Decile $1, \ldots, 9$ of wikipage load time. (c) CDF of page load time over the 24 hours.

Figure 12. Wikipedia replay: $\mathbf{SC}$ vs $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$ policies.

a typical (and, popular) Web-service. Thus an instance of MediaWiki[8] (version 1.28), as well as a MySQL server and the `memcached` cache daemon, were installed on each of the 48 servers. The *wikiloader* tool from [39], and a dump of the database of the English version of Wikipedia from [40], were used to populate the MySQL databases, resulting in each server containing an individual replica of the English Wikipedia.

1) Traffic and Workload Patterns: A traffic generator, able to replay a MediaWiki access trace and to record response times was developed, and experiments were run using 24 hours of traces from [40]. These traces correspond to 10% of all queries received by Wikipedia during this timeframe, from among which only traffic to the English Wikipedia was extracted and used for the experiment.

A first experiment was to size the server farm, *i.e.* to identify the smallest number of VMs necessary to be able to serve queries while exhibiting reasonable response times. With 28 VMs, the median response time during peak hours is smaller than 400 ms: the remainder of this section will assume this size for the server farm.

2) Connection Acceptance Policies Tested: Given the superior performance of $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$ in the experiments from section VI-B, the 24-hour trace was replayed against both $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$, and client-side response times were collected. As a baseline, the trace was also replayed against the reference $\mathbf{SC}$ policy.

3) Experimental Results: The experiment allowed classifying queries into two groups: (i) requests for static pages, which are not CPU-intensive, and for which response times were of the order of a millisecond, and (ii) requests for wiki pages, that trigger `memcached` or MySQL and thus are more CPU-intensive. 6LB was found to offer only a small improvement over $\mathbf{SC}$ for static page response time (figure 12c, top). However, the load times of wiki pages, identifiable by the string `/wiki/index.php/` in their URL, exhibited interesting differences.

Figure 12a depicts the wiki page request rate and the median wiki page load time for the three tested policies during the 24h replay (data has been binned in 10 minutes slots). It can be observed that at the off-peak period around 8:00 UTC, when the system was lightly loaded and subject to a request rate of around 110 pages per second, $\mathbf{SC}$ and $\mathbf{SR_4}$ yielded similar performance, and $\mathbf{SR_{dyn}}$ exhibited even lower response times. As the request rate increases, using the application-unaware $\mathbf{SC}$ policy yielded notably increased page load times – whereas when using $\mathbf{SR_4}$ or $\mathbf{SR_{dyn}}$, a comparably much smaller increase in page load times incurred.

To understand the response time variability over 24 hours, figure 12b depicts deciles 1-9 of the wiki page load time distribution, for each 10 minutes bin. Again, $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$ show less variability under higher loads than does $\mathbf{SC}$. Among $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$, the latter has the lower variability under lighter loads, but is outperformed under higher loads.

Finally, as an indicator of "global good behavior", figure 12c (bottom) depicts the CDF of the wiki page load times over the whole day. Overall, the median response time went from 0.22 s with $\mathbf{SC}$ to 0.18 s with $\mathbf{SR_4}$ and 0.16 s with $\mathbf{SR_{dyn}}$. Furthermore, the tail of the distribution is steeper when using 6LB, with the 90-th percentile going from 0.67 s with $\mathbf{SC}$ to 0.32 s with $\mathbf{SR_4}$ and 0.31 s with $\mathbf{SR_{dyn}}$.

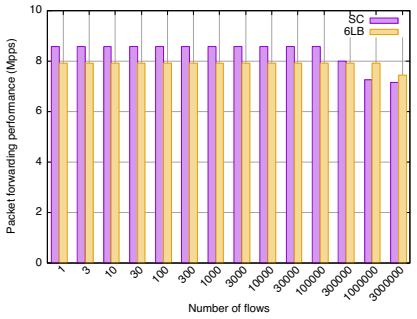


Figure 13. Upstream packet forwarding rate evaluation: 6LB vs single-choice consistent-hashing, using a single CPU core.

D. Throughput Evaluation

The advantages provided 6LB in load-balancing fairness come at the cost of some overhead as compared to single-choice load-balancing approaches, notably due to maintaining flow state and performing IPv6 SR header insertion. To understand the impact of 6LB in terms of CPU overhead, the packet-forwarding performance of the VPP implementation introduced in this paper is evaluated. Maglev [2] with GRE encapsulation has also been implemented as a VPP plugin,

[8]https://www.mediawiki.org/wiki/Download

and serves as a reference point. Evaluation was conducted on a single core of a machine running an Intel E5-2667 CPU at 3.2 GHz, with an Intel X710 10 Gbps NIC. The load-balancer was manually initialized to install a pre-determined number of flow entries, and a packet generator (sitting on another machine on the same Ethernet link) was set to send TCP ACK packets corresponding to these flows, at line rate. Packets were set to return to the packet generator, and the number of packets effectively forwarded by 6LB was recorded – allowing to determine the maximum forwarding capability of the implementation, for upstream traffic. ACK packets were used rather than SYN, as they are expected to represent the majority of the upstream traffic.

Figure 13 depicts the achievable forwarding rate (in millions of packets per second, Mpps), as a function of the number of flow entries installed. Two main results are to be noted. First, the kernel bypass and vectorization capabilities of VPP make it very efficient for load-balancing (be it single-choice or 6LB), with a raw forwarding capability of around 8 Mpps with one CPU core – with [2] reporting 2.7 Mpps with kernel bypass, and 0.5 Mpps without. When the number of flows reaches approximatively 10^5, the performance of both implementations degrades, as the flow table cannot reside entirely in the CPU cache. Second, it can be seen that 6LB incurs only 8% CPU overhead as compared to the load-balancer reference implementation. This overhead can be explained by the greater complexity of the per-packet operations, and the fact that the hash-table for flow state needs to handle collisions – whereas the one from the reference load-balancing plugin does not. Yet, this CPU overhead remains relatively negligible, and the additional 8% resources that might need to be deployed to use 6LB should be largely compensated by the fact that less application instances need be deployed, due to the greater fairness induced (as shown in section VI-B3).

VII. Conclusion

This paper has introduced 6LB, an innovative network service offering flexible, scalable, reliable, distributed, application-aware, but at the same time application-agnostic and application-protocol-agnostic, load balancing.

This is accomplished by an architecture in which (i) load-balancers using an extended consistent hashing algorithm to map incoming flows onto a set of candidate application instances, (ii) to offer – not impose – these network flows to the candidate application instances, leaving them the decision to accept (or not) a flow. Once an application instance has accepted a flow (iii) data packets of no interest to the load-balancer are sent directly from the application instance to the client. When a network flow is reassigned to another load balancer (*e.g.* if a load balancer is added to, or removed from, the system), this will be detected, and in-band signaling will reestablish the necessary state in this new load-balancer for continued operation, ensuring (iv) that a traffic flow between a client and an application instance becomes pinned to that application instance, regardless of changes to the load balancing infrastructure. The use of Segment Routing, specifically SR Functions, allow defining and implementing this as a network service, *i.e.* entirely below the application layer.

This paper has also introduced a simple two-server random assignment policy (motivated by the concept of *power of two choices*), combined with a static or dynamic query acceptance policy. These policies were compared to a naive one-server random query dispatch policy, by means of an analytical model, as well as an evaluation on a 48-servers deployment. Evaluation of those policies, conducted using a simulated Poisson workload as well as on a Wikipedia replica, shows that 6LB is able to better spread the load between all servers than single-choice consistent-hashing load-balancers. Evaluation of the packet-forwarding performance of the implementation shows that these benefits are attained at a negligible cost in terms of CPU overhead.

Appendix

Proof of Theorem 1. Let $c \geq 1$ a threshold parameter, and $\lambda \in [0, 1)$. First, it will be shown that $s_n \leq \lambda^n$ for all $n \geq 0$. For $1 \leq n \leq c$, $s_n = \lambda[s_{n-1} - s_c(1 - s_{n-1})] \leq \lambda s_{n-1}$; for $n > c$, $s_n = \lambda s_c s_{n-1} \leq \lambda s_{n-1}$. Thus $s_n \leq \lambda s_{n-1}$ for all $n \geq 1$, and since $s_0 = 1$, it follows by induction that $s_n \leq \lambda^n$.

It remains to show that $\mathbb{E}(\hat{T}) \leq \frac{1}{1-\lambda}$. Let $\delta \in [0, 1]$, then: $\mathbb{E}(\hat{T}) = \frac{1}{\lambda}\sum_{n=1}^{+\infty} s_n + \delta s_c = \frac{1}{\lambda}\sum_{n=1}^{c-1} s_n + \frac{1}{\lambda}\frac{s_c}{1-\lambda s_c} + \delta s_c$. Using $s_n \leq \lambda^n, s_c \leq \lambda^c$ and $\delta \leq 1$ gives: $\mathbb{E}(\hat{T}) \leq \frac{1}{\lambda}\sum_{n=1}^{c-1}\lambda^n + \frac{1}{\lambda}\frac{\lambda^c}{1-\lambda^{c+1}} + 1 \cdot \lambda^c = \frac{1-\lambda^{c-1}}{1-\lambda} + \frac{\lambda^{c-1}}{1-\lambda^{c+1}} + \lambda^c = \frac{1}{1-\lambda} + \lambda^{c-1}(\frac{1}{1-\lambda^{c+1}} - \frac{1}{1-\lambda} + \lambda)$. Since $c \geq 1$, $\lambda^{c+1} \leq \lambda^2$, which yields: $\mathbb{E}(\hat{T}) \leq \frac{1}{1-\lambda} + \lambda^{c-1}(\frac{1}{1-\lambda^2} - \frac{1}{1-\lambda} + \lambda) = \frac{1}{1-\lambda} - \frac{\lambda^{c+2}}{1-\lambda^2} \leq \frac{1}{1-\lambda}$, which completes the proof. □

State	Incoming SR function / SR functions added	Next state
`LB_LISTEN`	SYN from client / `s1.connectAvail(lb)` `s2.connectForce(lb)`	`HUNTING`
`LB_LISTEN`	data from client / `s1.recoverStickiness(lb)` `s2.recoverStickiness(lb)`	`HUNTING`
`HUNTING`	SYN from client / `s1.connectAvail(lb)` `s2.connectForce(lb)`	`HUNTING`
`HUNTING`	`createStickiness(s)` / remove SR header	`STEER(s)`
`STEER(s)`	data from client / `s.ackStickiness(lb)`	`STEER(s)`
`STEER(s)`	`removeStickiness(s)` / remove SR header	`LB_LISTEN` after 10 sec

Table II
HANDSHAKE PROTOCOL STATE MACHINE FOR A GIVEN FLOW, AT A LOAD-BALANCER *lb*

References

[1] D. Thaler and C. Hopps, "Multipath issues in unicast and multicast next-hop selection," in *Requests For Comments*. Internet Engineering Task Force, 2000, no. 2991.

[2] 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 *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[3] M. Rahman, S. Iqbal, and J. Gao, "Load balancer as a service in cloud computing," in *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*. IEEE, 2014, pp. 204–211.

[4] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

State	Incoming SR function / SR functions added	Next state
LISTEN	connectAvail(lb) (available) / remove SR header	WAIT(lb)
LISTEN	connectAvail(lb) (busy) / forward	LISTEN
LISTEN	recoverStickiness(lb) (not local) / forward	LISTEN
LISTEN	connectForce(lb) / remove SR header	WAIT(lb)
WAIT(lb)	connect[Avail|Force](lb) / remove SR header	WAIT(lb)
WAIT(lb)	data from app / lb.createStickiness(s)	WAIT(lb)
WAIT(lb)	ackStickiness(lb) / remove SR header	DIRECT(lb)
DIRECT(lb)	recoverStickiness(lb2) (local) / remove SR header	WAIT(lb2)
DIRECT(lb)	ackStickiness(lb) / remove SR header	DIRECT(lb)
DIRECT(lb)	data from app / direct return to client	DIRECT(lb)
DIRECT(lb)	FIN from app / lb.removeStickiness(s)	LISTEN after 10 sec

Table III
HANDSHAKE PROTOCOL STATE MACHINE FOR A GIVEN FLOW, AT A SERVER s

[5] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[6] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen, "SRLB: The Power of Choices in Load Balancing with Segment Routing," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 2011–2016.

[7] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: cloud scale load balancing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[8] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[9] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11, pp. 1203–1213, 1999.

[10] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Transactions on Networking (TON)*, vol. 6, no. 1, pp. 1–14, 1998.

[11] 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," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[12] R. Gandhi, Y. C. Hu, C.-K. Koh, H. 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*, 2015, pp. 473–485.

[13] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM Sigcomm Demo*, vol. 4, no. 5, p. 6, 2009.

[14] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild." *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[15] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 129–143.

[16] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE transactions on software engineering*, no. 5, pp. 662–675, 1986.

[17] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM journal on computing*, vol. 29, no. 1, pp. 180–200, 1999.

[18] H. Shen and C.-Z. Xu, "Locality-aware and churn-resilient load-balancing algorithms in structured peer-to-peer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 849–862, 2007.

[19] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[20] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *NSDI*, vol. 13, 2013, pp. 185–198.

[21] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure, "Traffic duplication through segmentable disjoint paths," in *IFIP Networking Conference (IFIP Networking), 2015*. IEEE, 2015, pp. 1–9.

[22] V. Cardellini, M. Colajanni, and S. Y. Philip, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, p. 28, 1999.

[23] Q. Zhang, L. Cherkasova, and E. Smirni, "Flexsplit: A workload-aware, adaptive load balancing strategy for media clusters," in *Electronic Imaging 2006*. International Society for Optics and Photonics, 2006, pp. 60 710I–60 710I.

[24] G. Ciardo, A. Riska, and E. Smirni, "Equiload: a load balancing policy for clustered web servers," *Performance Evaluation*, vol. 46, no. 2, pp. 101–124, 2001.

[25] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered web servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 3, pp. 219–233, 2005.

[26] S. Sharifian, S. A. Motamedi, and M. K. Akbari, "A content-based load balancing algorithm with admission control for cluster web servers," *Future Generation Computer Systems*, vol. 24, no. 8, pp. 775–787, 2008.

[27] "HAProxy: the reliable, high-performance TCP/HTTP load balancer." [Online]. Available: http://www.haproxy.org

[28] R. Gandhi, Y. C. Hu, and M. Zhang, "Yoda: A highly available layer-7 load balancer," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 21.

[29] C. Filsfils *et al.*, "SRv6 Network Programming," Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-srv6-network-programming-01, Jun. 2017, work in Progress.

[30] S. Previdi *et al.*, "IPv6 Segment Routing Header (SRH)," Internet Engineering Task Force, Internet-Draft draft-ietf-6man-segment-routing-header-06, 2017, work in Progress.

[31] The Fast Data Project (fd.io), "Vector Packet Processing (VPP)." [Online]. Available: https://wiki.fd.io/view/VPP

[32] F. William, "An introduction to probability theory and its applications," 1950.

[33] D. J. Newman, "The double dixie cup problem," *The American Mathematical Monthly*, vol. 67, no. 1, pp. 58–61, 1960.

[34] D. S. E. Deering and R. M. Hinden, "IP Version 6 Addressing Architecture," in *Request for Comments*. Internet Engineering Task Force, 2006, no. 4291.

[35] M. D. Mitzenmacher, "The power of two choices in randomized load balancing," Ph.D. dissertation, UNIVERSITY of CALIFORNIA at BERKELEY, 1996.

[36] J. D. Little, "A proof for the queuing formula: L=λw," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[37] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984, vol. 38.

[38] "The Apache HTTP server project." [Online]. Available: http://www.apache.org

[39] E.-J. van Baaren, "Wikibench: A distributed, wikipedia based web application benchmark," *Master's thesis, VU University Amsterdam*, 2009.

[40] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

Yoann Desmouceaux received the Diplôme d'Ingénieur from École Polytechnique (Palaiseau, France) in 2014, and the MSc degree in Advanced Computing from Imperial College (London, U.K.) in 2015. He is currently undertaking an industrial PhD under joint supervision of Mark Townsley (Cisco Systems) and Thomas Clausen (École Polytechnique). His research interests include high-performance networking, IPv6-centric protocols, load-balancing, reliable multicast and data-center optimization algorithms.

Pierre Pfister received an M.Sc from École Polytechnique (Palaiseau, France) in 2012 as well as an M.Sc in Communication Systems from École Polytechnique Fédérale de Lausanne (Switzerland) in 2013. He has been an active participant in the Home Networking and IPv6 IETF working groups, where he co-authored multiple Standard Track RFCs. He is currently a Research Engineer at Cisco's Paris Innovation and Research Lab, where his interests include high-scale content delivery networks, load-balancing and IPv6 multi-homed networks. He received the Cisco Pioneer Award in 2017 for his work on FD.io VPP open-source project, where he applies high performance computing principles to virtualization and networking technologies.

Jérôme Tollet is a Distinguished Engineer working for the CTO Office of Cisco. Jérôme has extensive experience of computer systems and network architectures, and strong technical expertise gained from more than 18 years designing and implementing networking solutions. He actively contributed to the Open Networking Foundation (ONF), ETSI Industry Specification Group on NFV, IETF Service Function Chaining (SFC) Working Group and various open source initiatives including OpenDayLight, OpenStack and FD.io. He holds multiple patents and obtained his Master's degree in Computer Science from a joint degree program at Pierre and Marie Curie - Paris 6 University and ENST Paris. Jérôme is a frequent speaker at international conferences.

Mark Townsley is a Cisco Fellow, Professor Chargé de Cours at École Polytechnique, and co-founder of the Paris Innovation and Research Laboratory (PIRL). Before Joining Cisco in 1997, he held positions at IBM, the Institute for Systems Research (ISR) and the Center for Satellite and Hybrid Communications Networks (CSHCN) at the University of Maryland. Mark served as IETF Internet Area Director from 2005-2009, IETF L2TP Working Group Chair from 1999-2005, IESG Liaison to the Internet Architecture Board (IAB), and IETF Pseudowire WG Technical Advisor. Mark was the lead developer of the original implementation of L2TP in Cisco IOS as well as lead author of IETF L2TP protocol specification (RFC 2661). One of the original architects of the World IPv6 Day and Launch, Mark contributed significantly to the deployment of IPv6 on the internet, including lead author of RFC 5969, IPv6 Rapid Deployment (6RD). In 2011, Mark co-founded the IETF Homenet Working Group, and served as chair until 2017. In addition to his Faculty appointment at École Polytechnique, Mark lectures on Future Internet Architectures at Telecom Paris Tech (TPT), and serves on the steering committee for the joint TPT-Polytechnique Advanced Computer Networking master's degree. Mark holds a Bachelor of Science (summa cum laude) degree in Electrical Engineering from Auburn University and a Masters degree in Computer Science (magna cum laude) from the Johns Hopkins University Applied Physics Laboratory. When not traveling, he lives with his family in Paris, France.

Thomas Clausen is a graduate of Aalborg University, Denmark (M.Sc., PhD – civilingeniør, cand.polyt), and a Senior Member of the IEEE. Thomas has, since 2004 been on faculty at École Polytechnique, France's leading technical and scientific university, where he holds the Cisco-endowed "Internet of Everything" academic chaire.
At École Polytechnique, Thomas leads the computer networking research group. He has developed, and coordinates, the computer networking curriculum, and co-coordinates the Masters program in "Advanced Communication Networks" (ACN). He has published more than 80 peer-reviewed academic publications (which have attracted more than 12000 citations) and has authored and edited 24 IETF Standards, has consulted for the development of IEEE 802.11s, and has contributed the routing portions of the recently ratified ITU-T G.9903 standard for G3-PLC networks – upon which, e.g., the current SmartGrid & ConnectedEnergy initiatives are built. He serves on the scientific council of ThinkSmartGrids (formerly: SmartGridsFrance).