SRLB: The Power of Choices in Load Balancing with Segment Routing

Yoann Desmouceaux*†, Pierre Pfister†, Jérôme Tollet†, Mark Townsley*†, Thomas Clausen*
*Ecole Polytechnique, 91128 Palaiseau, France
{yoann.desmouceaux,mark.townsley,thomas.clausen}@polytechnique.edu
†Cisco Systems Paris Research and Innovation Laboratory (PIRL), 92782 Issy-les-Moulineaux, France
{ydesmouc,ppfister,jtollet,townsley}@cisco.com

***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 on its local state. This way, applications themselves naturally decide on how to share incoming connections, while incurring minimal network overhead, and no out-of-band signaling.**

Tests on different workloads – including realistic workloads such as replaying actual Wikipedia access traffic towards a set of replica Wikipedia instances – show significant performance benefits, in terms of shorter response times, when compared to a traditional random load-balancer.

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 such load-balancers:

1) Network-level load-balancers, which operate at Layer-4 – a simple approach being to rely on Equal Cost Multi-Path (ECMP) [1] to homogeneously distribute network flows between the application instances. These approaches generally do not take application state into account, which can lead to suboptimal server utilization.
2) Application-aware 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. These approaches generally incur a cost from monitoring the state of each application instance, and sometimes also terminate network connections (such as an HTTP proxy).

A desirable load-balancer would combine the best of these two categories: be application or application-layer protocol agnostic (*i.e.* operate at Layer-4) and incur no monitoring overhead – yet be able to make informed dispatching decisions depending on the state of the applications.

A. Statement of Purpose

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

A key philosophical 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 – and, thus, SRLB discards the traditional design by which queries are unconditionally assigned to an application instance by the load-balancer. Rather, SRLB offers a received query to several application instance candidates, only one of which accepts and processes the query.

What enables this is IPv6 Segment Routing (SR) [2] – which allows specifying to the network that it should do more than just forward a data packet towards its destination: SR permits directing data packets through an (ordered) set of intermediaries, and instructing these intermediaries what to do with a received data packet. For example, one instruction could be *"process the contained query, if you can"*. All this within the IP forwarding plane, *i.e.* in an application-layer protocol agnostic fashion.

The role of the load balancer, then, simply becomes to monitor TCP flows, to ensure that data packets belonging to the same flow are delivered to the same application instance as the one which accepted the first packet of the flow.

In this way, SRLB enables that query acceptance decisions are made strictly locally, based on real-time information on the state of the application instance.

B. Related Work

Among existing Layer-4 load-balancing approaches, [3] (Maglev) and [4] (Ananta) aim at being able to scale the number of load-balancer instances at will, and make use of ECMP to distribute flows between those instances. They also make use of consistent hashing, 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.

Conversely, [5], [6] use Software Defined Networking (SDN) on a controller, to monitor the application instance load and network load – and then install network rules to direct flows to these application instances.

[7] lists three standard load-balancing techniques 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 includes [8], [9], [10], which assign queries as a function of their estimated size so that each application instance becomes equally loaded. In [11], a feedback approach is used to estimate the parameters of a queuing model representing the system, before making a load-balancing decision.

Layer-7 (application-layer protocol aware) load-balancers, *e.g.* [12] (HAProxy), also propose application-awareness by estimating the load on each application instance and assigning new queries accordingly. Load estimates are obtained by tracking open connections through the load-balancer to the backend servers, thus do not take other generated loads into account – *e.g.* internal traffic or traffic coming from other load-balancers.

C. Segment Routing

In IPv6 Segment Routing (SR) [2], each data packet indicates not only the destination to which the network is expected to carry the packet, but also an ordered sequence of instructions or operations (called *segments*), that the network is expected to execute on that packet. When a segment is "completely processed", that segment is (conceptually) discarded and the next segment (if any) is processed by the network, before the data packet is delivered to the final destination.

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* – but SR enables also traffic engineering, service chaining, etc. The SR information is expressed as an IPv6 Extension Header, comprising a list of segments and a counter `SegmentsLeft` – indicating the number of remaining segments to be processed.

D. Paper Outline

The remainder of this paper is organized as follows. Section II describes how Segment Routing can be used to perform *Service Hunting*, that is, in-network service selection. Section III describes two simple example connection acceptance policies. SRLB is then evaluated: section IV describes the experimental platform that has been used, section V provides an evaluation with a synthetic workload, and a realistic workload consisting of a Wikipedia replica is analyzed in section VI. Finally, section VII concludes this paper.

II. Service Hunting

This section introduces a new general concept, *Service Hunting*, which uses SR to direct network packets from a new flow through a set of *candidate servers* until one accepts the connection.

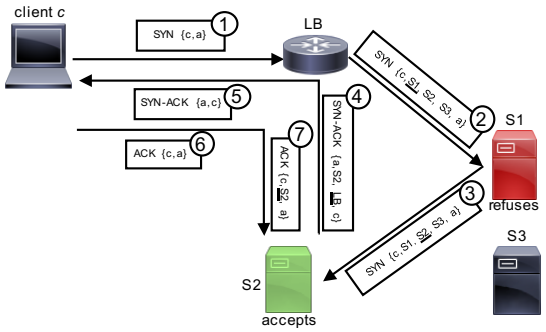

Figure 1. Service Hunting from client c to application a with 3 servers s_1, s_2, s_3. The path (source, segments, destination) is indicated between curly braces. The active segment is underlined.

A. Overview

Service Hunting assumes an IPv6 data center, in which applications are identified by *virtual IP addresses* (VIPs), and can be replicated among several servers, identified by their *physical addresses*. Servers run a *virtual router* (for instance, VPP [13]), which dispatches packets between physical NICs and application-bound virtual interfaces. Located at the edge of the data center, the load-balancer advertises routes for the VIPs.

When a query (typically, a TCP SYN packet as part of a connection request) for a VIP arrives at the load-balancer, the load-balancer will select a set of *candidate servers* which host an instance of the sought-after application, and insert an SR header into the IPv6 data packet accordingly. The SR header will contain a list of segments, each indicating that the query can be processed by either of these instances, and with the VIP as the last segment.

When the query reaches a candidate server, 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 is a purely local decision to accept or not the query, and 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 must not refuse a query.

In order to ensure that packets part of the same flow are treated by the same application instance, upon having accepted a query, the server hosting the application instance must signal this to the load balancer. This is done by inserting an SR header containing its own IP address, and the IP address of the load-balancer, in the connection acceptance packet (typically, a TCP SYN-ACK). An example of the whole procedure with 3 servers is given in figure 1.

B. Server Selection Policy

When the load-balancer receives a query, different policies can be used to select the list of candidate servers to include in the SR header. Parameters of importance for this selection include the number of candidate servers to include, and the

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

```
for each packet with SegmentsLeft = 2 do
    b ← number of busy threads
    if b < c then
        SegmentsLeft ← 0
        forward packet to application
    else
        SegmentsLeft ← 1
        forward packet to second server in SR list
    end if
end for
for each packet with SegmentsLeft = 1 do
    SegmentsLeft ← 0
    forward packet to application
end for
```

scheme according to which they are selected. Possibilities for such schemes include random selection and consistent hashing.

A simple and lightweight approach consists of selecting server addresses at random. While any number of random server addresses can be inserted in the SR segment list, [14] demonstrates a decreased marginal benefit from more than two servers, when the goal is load balancing. Thus, for the purpose of the experimental verification in this paper, two servers will be chosen at random from among all servers hosting a given application instance.

C. Connection Acceptance Policy

To perform Service Hunting, SRLB assumes an application agent, locally available to the virtual router in each server, which in real time informs the virtual router as to if the application instance wishes to accept queries. The application agent may make this decision based on coarse-grained information (*e.g.* CPU load, memory footprint) available from the operating system; or on more fine-grained information, if the application exposes real-time metrics about its load state (idle threads, *etc.*). Done through shared memory, this incurs no system calls or synchronization, thus imposes a negligible run-time cost.

III. Example Connection Acceptance Policies

This section describes two simple policies that can be used to decide whether or not to accept new connections. These policies assume a standard master-slave threading model for the application. Section IV will then show how these policies can be applied in the case of an HTTP server such as Apache.

A. Static policy

Let n be the number of worker threads of the application, and c a threshold parameter between 0 and $n+1$. In Algorithm 1, we introduce a simple policy, $\mathbf{SR_c}$, whereby the first server accepts the connection if and only if less than c worker threads are busy (recall that the second server always accepts the connection). When $c = 0$, all requests are satisfied by the second servers in the SR lists; when $c = n + 1$, all requests are satisfied by the first ones. These two cases reduce to a standard random load-balancing scheme. All choices of c between these two extremes yield an improvement over random load-balancing. Indeed, a server with c or more busy threads will be assigned a connection only if both itself and the first server in the SR list have more than c busy threads. The chance for this to happen is the square of the probability that one server has more than c busy threads, thus allowing for a better repartition of the load between all servers.

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

```
c ← 1 {or other initial value}
accepted ← 0
attempt ← 0
windowSize ← 50 {or other window size}
for each 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 < 0.4 and c < n then
            c ← c + 1
        else if accepted/windowSize > 0.6 and c > 0 then
            c ← c − 1
        end if
        attempt ← 0
        accepted ← 0
    end if
    {use SR_c policy with current value of c}
    b ← number of busy threads
    if b < c then
        accepted ← accepted + 1
        SegmentsLeft ← 0
        forward packet to application
    else
        SegmentsLeft ← 1
        forward packet to second server in SR list
    end if
end for
for each packet with SegmentsLeft = 1 do
    SegmentsLeft ← 0
    forward packet to application
end for
```

The choice of the parameter c has a direct influence on the behavior of the global system. Small values of c will yield better results under light loads, and high ones will yield better results under heavy loads. Indeed, if the chosen value is too small as compared to the load, the second server will receive almost all connections, and vice-versa. If the load pattern is known by the operator, the parameter c can be manually selected so as to maximize the load-balancing efficiency. If this is not the case, a dynamic policy can be used in order to automatically tune the value of the parameter.

B. Dynamic policy

This section introduces a dynamic policy, $\mathbf{SR_{dyn}}$, that can be used when the typical request load is unknown. The underlying intuition is the following: if the rejection ratio of the connection acceptance function is 0 (or 1), only the first (or second) candidates in SR lists serve requests, falling back to standard randomized load-balancing. To maximize utility, this policy aims to maintain a rejection ratio of $\frac{1}{2}$, by dynamically adapting the value of c so that this ratio stays close to $\frac{1}{2}$. The detailed procedure is described 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).

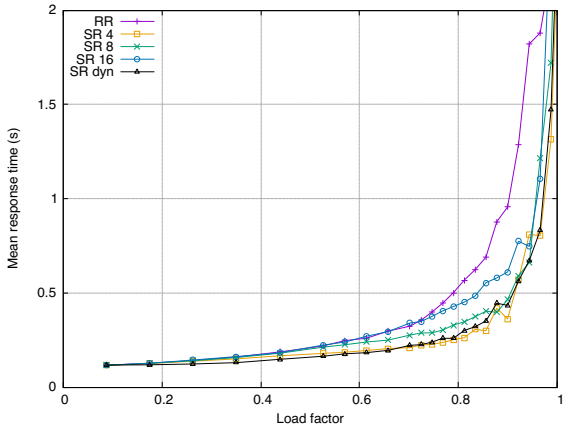

Figure 2. Average page load time for the Poisson workload as a function of the normalized request rate ρ: $\mathbf{RR}$ vs different $\mathbf{SR_c}$ policies (4, 8, 16, and dynamic).

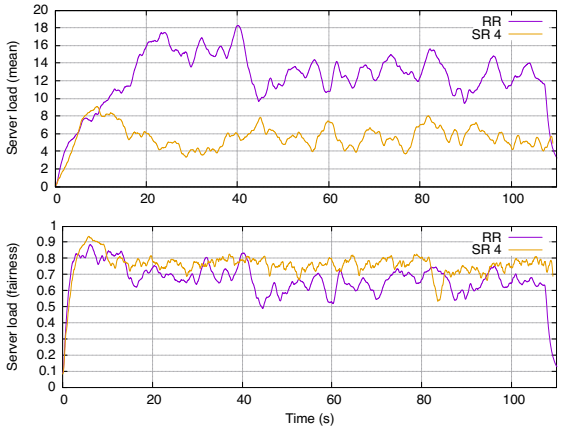

Figure 4. Instantaneous server load for a run of 20000 queries of the Poisson workload (mean and fairness over the 12 servers): $\mathbf{RR}$ vs $\mathbf{SR_4}$ policy, $\rho = 0.88$

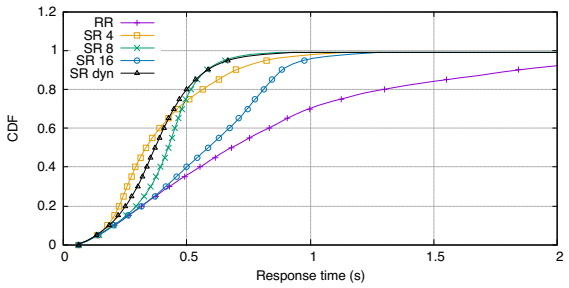

Figure 3. CDF of page load time over 20000 queries for the Poisson workload: $\mathbf{RR}$ vs different $\mathbf{SR_c}$ policies, $\rho = 0.88$

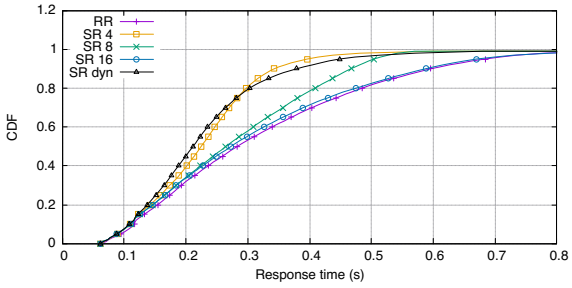

Figure 5. CDF of page load time over 20000 queries for the Poisson workload: $\mathbf{RR}$ vs different $\mathbf{SR_c}$ policies, $\rho = 0.61$

IV. Experimental Platform

The experimental platform used for evaluating SRLB is composed of a load-balancer, a server agent for the Apache HTTP server, and an overall system.

A. Load-Balancer

The load-balancer performing SR header insertion and flow steering is implemented as a VPP plugin [13]. While this choice is not significant to the performance results presented in this paper, it is convenient as VPP embeds an IPv6 Segment Routing stack and is kernel-bypass virtual routing capable.

B. Apache HTTP Server Agent

A server agent for the Apache HTTP server [15] has been implemented as a VPP plugin, accessing Apache's *scoreboard* shared memory[1] 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 III.

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

C. System Set-up

The experiments, described in sections V and VI, are conducted on a common platform. A traffic generator and a load balancer reside in one physical machine. Twelve application instances of an Apache HTTP server reside each in a 2-core VM, all of which are hosted in another physical machine, running a 24-core Intel Xeon E5-2690 CPU. VPP instances running in the load-balancer and on the 12 application servers were bridged on the same link, with routing tables statically configured.

Finally, 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. This allows that, under heavy load, the application response delays are measured, and not possible TCP SYN retransmit delays.

V. Poisson Traffic

A. Traffic and Workload Patterns

To evaluate the efficiency of the connection acceptance policies from section III under different loads, SRLB was tested

against a simple CPU-intensive web application, consisting of a PHP script running a `for` loop, and whose duration follows an exponential distribution of mean 100 ms. The 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 12-servers swarm, *i.e.* the smallest value of λ for which some TCP connections were dropped.

B. Connection Acceptance Policies Tested

With $\rho = \lambda/\lambda_0$ as the normalized request rate, for 24 values of ρ in the range $(0, 1)$, a Poisson stream of 20000 queries with rate ρ was injected in the load-balancer, using the policies $\mathbf{SR_4}$, $\mathbf{SR_8}$, $\mathbf{SR_{16}}$, $\mathbf{SR_{dyn}}$. As baseline, the same tests were run with a policy $\mathbf{RR}$ where queries are randomly assigned to one server, without Service Hunting.

C. Experimental Results

Figure 2 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 $\mathbf{RR}$ for $\rho = 0.88$. $\mathbf{SR_8}$ and $\mathbf{SR_{16}}$ likewise perform better than $\mathbf{RR}$ for all loads, but with a lesser impact. $\mathbf{SR_{dyn}}$ offers results close to the best static policy, $\mathbf{SR_4}$, showing that a manual policy tuning is not needed to obtain good performance.

Figure 3 shows the CDF of the page response time for the 20000 queries batch with $\rho = 0.88$, for each policy. $\mathbf{RR}$ 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), as well as the corresponding fairness index: $\frac{(\sum_{i=1}^{12} x_i(t))^2}{12\sum_{i=1}^{12} x_i(t)^2}$ (where $x_i(t)$ is the load of server i at time t) of each server, depicted in figure 4[2]. As $\mathbf{SR_4}$ better spreads queries between all servers (the fairness index is closer to 1), and servers are individually less loaded, better response times result.

For lighter loads, the same kind of behavior can be observed, except that high $\mathbf{SR_c}$ policies exhibit no benefits as compared to $\mathbf{RR}$. Figure 5 shows the CDF of the page load time for an experiment where $\rho = 0.61$: $\mathbf{SR_{16}}$ yields no improvement over $\mathbf{RR}$, 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 static policy.

VI. Wikipedia Replay

A. Traffic and Workload Patterns

To evaluate the performance of SRLB when exposed to a realistic workload, an experiment was constructed to reproduce a typical Web-service. Thus an instance of MediaWiki[3] (version 1.28), as well as a MySQL server and the `memcached` cache daemon, were installed on each of the 12 servers. The *wikiloader* tool from [16], and a dump of the database of the English version of Wikipedia from [17], were used to populate the MySQL databases, resulting in each server containing an individual replica of the English Wikipedia.

A traffic generator, able to replay a MediaWiki access trace with millisecond granularity and to record response times, was developed, and experiments were run using 24 hours of traces from [17]. 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 identify the maximum achievable rate for the testbed. Using $\mathbf{RR}$, the testbed could sustain 50% of the peak load while exhibiting reasonable response times (smaller than one second).

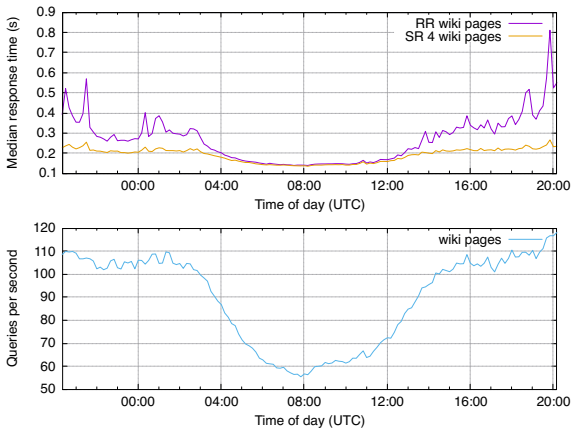

Figure 6. Wikipedia replay: query rate and median load time for wiki pages over 24 hours (10 mins bins). $\mathbf{RR}$ vs $\mathbf{SR_4}$ policy.

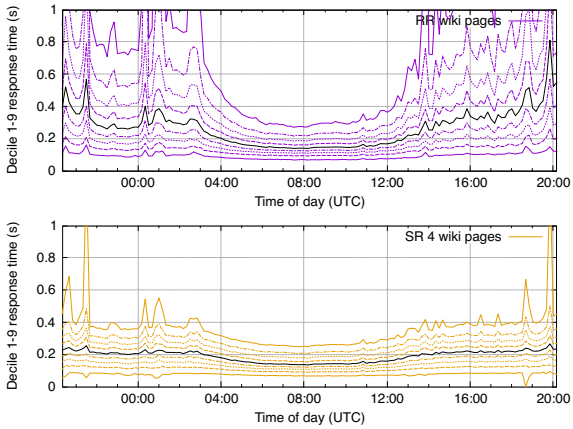

Figure 7. Wikipedia replay: decile $1, \ldots, 9$ of load time for wiki pages over 24 hours (10 mins bins). $\mathbf{RR}$ vs $\mathbf{SR_4}$ policy.

B. Connection Acceptance Policies Tested

Given the superior performance of $\mathbf{SR_4}$ in the experiments from section V-C, 50% of the 24-hour trace was replayed

[2]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.

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

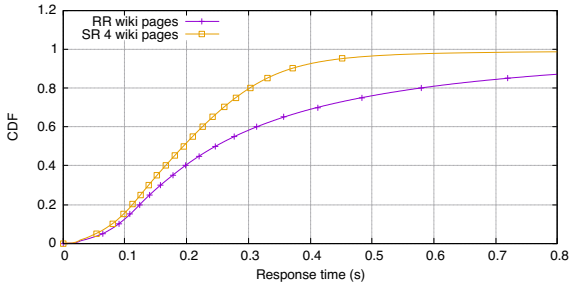

Figure 8. Wikipedia replay: CDF of wiki page load time over 24 hours. $\mathbf{RR}$ vs $\mathbf{SR_4}$ policy.

against both $\mathbf{SR_4}$ and $\mathbf{RR}$, and client-side response times were collected.

C. 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.

Static page response times were found to be equivalent, regardless of if $\mathbf{SR_4}$ or $\mathbf{RR}$ were used. However, the load times of wiki pages, identifiable by the string `/wiki/index.php` in their URL, exhibited interesting differences.

Figure 6 depicts the wiki page request rate and the median wiki page load time for both $\mathbf{RR}$ and $\mathbf{SR_4}$ during the 24h replay[4]. 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 60 pages per second, $\mathbf{RR}$ and $\mathbf{SR_4}$ yielded very similar performance. However as the request rate increased, when using the application-unaware $\mathbf{RR}$ policy observed page load times increased notably – whereas when using $\mathbf{SR_4}$, a comparably much smaller increase in page load times incurred.

To understand the variability of the response times along the 24 hours, figure 6 depicts deciles 1-9 of the wiki page load time distribution, for each 10 minutes bin. Again, $\mathbf{SR_4}$ shows less variability under higher loads than does $\mathbf{RR}$.

Finally, as an indicator of "global good behavior", figure 8 depicts the CDF of the wiki page load times over the whole day. Overall, the median response time went from 0.25s with $\mathbf{RR}$ to 0.20s with $\mathbf{SR_4}$. Furthermore, the tail of the distribution is steeper with the application-aware $\mathbf{SR_4}$ scheme, with the third quartile going from 0.48s to 0.28s.

VII. Conclusion

This paper has introduced SRLB, a distributed load balancing system which, while remaining application and application-layer protocol agnostic, is able to perform application-instance state aware query assignment by way of Service Hunting within the IP forwarding plane. This allows SRLB to offer — not impose — queries to application instances, leaving the decision to accept (or not) a query to those. SRLB thus does not employ any out-of-band signaling, nor requires any central monitoring, nor imposes any load balancing policy. As a flexible framework, SRLB is able to accommodate a broad spectrum of policies. As a baseline, this paper has tested a naive random query dispatch policy – and has compared with a static and a dynamic query acceptance policy, enabled by the Service Hunting features of SRLB developed in this paper. Evaluation of those policies, conducted using a simulated Poisson workload, as well as on a Wikipedia replica, shows that SRLB is able to better spread the load between all servers than a random load-balancer.

[4]Data has been binned in 10 minutes slots.

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] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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, http://www.globule.org/publi/WWADH_comnet2009.html.