Stateless Load-Aware Load Balancing in P4

Benoît Pit-Claudel*†, Yoann Desmouceaux*†, Pierre Pfister†, Mark Townsley*†, Thomas Clausen*
*École Polytechnique †Cisco Systems
{benoit.pit-claudel,yoann.desmouceaux,thomas.clausen}@polytechnique.edu, {ppfister,townsley}@cisco.com

***Abstract*—Leveraging the performance opportunities offered by programmable hardware, stateless load-balancing architectures allowing line-rate processing are appealing. Moreover, it has been demonstrated that significantly fairer load-balancing can be achieved by an architecture that considers the actual load of application instances when dispatching connection requests. Architectures which maintain per-connection state for resiliency and/or track application load state for fairness are, however, at odds with hardware-imposed memory constraints. Thus, a desirable load-balancer for programmable hardware would be both stateless and able to dispatch queries to application instances according to their current load.**

This paper presents SHELL, a stateless application-aware load-balancer combining (i) a power-of-choices scheme using IPv6 Segment Routing to dispatch new flows to a suitable application instance from among multiple candidates, and (ii) the use of a covert channel to record/report which flow was assigned to which candidate in a stateless fashion. In addition, consistent hashing versioning is used to ensure that connections are maintained to the correct application instance, using Segment Routing to "browse" through the history when needed. The stateless design of SHELL makes it suitable for hardware implementation, and this paper describes the implementation of a P4-NetFPGA prototype. A performance evaluation of this SHELL implementation demonstrates throughput and latency characteristics comparable to other stateless load-balancing implementations, while enabling application instance-load-aware dispatching and significantly increasing per-connection consistency resiliency.

I. Introduction

In data-center and cloud architectures, workload virtualisation has become the norm, with applications replicated among multiple application instances, each capable of independently serving incoming queries [1], [2], [3]. An important functional part in these architectures is the load-balancer (LB), dispatching incoming queries amongst application instances. To make the load-balancer "invisible", a Virtual IP Address (VIP), shared by all application instances providing the same service, is advertised to the Internet in place of the address of the load-balancer, requiring the load-balancer to provide per-connection consistency (PCC), *i.e.* ensuring that traffic from a connection (typically identified by its network 5-tuple: source & destination addresses, L4 protocol, source & destination port) is *always* directed to the same application instance. *Naive* load balancing use Equal Cost Multi-Path (ECMP) [4] mapping connections to application instances, using a hash function and a modulo operation.

A. Challenges

A drawback of using ECMP for load-balancing is the lack of resiliency to changes to the application instance set, which causes the modulus in the ECMP operation to change and most connections to be redistributed across application instances, thus breaking PCC and causing connection resets. *Consistent hashing* [5], [6], [7] attempts to address this, by providing a more resilient mapping of connections across application instances [8], [9] through maintaining an intermediate table which, with high probability, yields a persistent mapping of the 5-tuple space to the set of application instances, even when faced with changes in the application instance set. Maglev [8] uses consistent hashing with per-connection state to maximize the probability of PCC: a connection breaks only when **both** (i) per-connection state is removed (if memory is exhausted – *e.g.* due to a denial-of-service attack – or if traffic is rebalanced to a new LB instance) **and** (ii) consistent-hashing changes the mapping of the connection to a different application instance (if there is a change in the set of application instances, which should affect only a small number of connections).

The pseudo-random nature of consistent hashing assigns queries to application instances *regardless* of their actual load state [8]. While this does not pose any problem for non-CPU-intensive applications (*e.g.* serving static Web pages), performance may degrade for CPU-intensive applications (*e.g.* data processing), for which the number of concurrently served queries per application instance must be minimized. Assigning queries to the least loaded from among *two* randomly chosen application instances (rather than to *one* randomly chosen application instance) was shown in [10] to improve load-balancing fairness. Based on this, 6LB [11] combines a Maglev-like consistent hashing with assigning connections to set of two application instances, which decide amongst themselves which will accept the connection. This is achieved by forwarding connection request (SYN) packets using IPv6 Segment Routing (SR) [12] and then by maintaining state in the LB as to which server has accepted the connection.

The need to keep connection-state make Maglev and 6LB difficult to implement on programmable hardware devices, whereas it is known that hardware-based load-balancers offer a potential performance benefit [13], [14], [15]. Beamer [15] circumvents this state requirement, by calling on assistance from servers hosting application instances to maintain PCC. When the set of application instances changes, and consistent hashing maps a flow to a different application instance, Beamer directs packets from that flow to that *new* application instance, while also embedding the address of the previously used application instance in the packet header. This allows the *new* application instance, in case it does not have connection

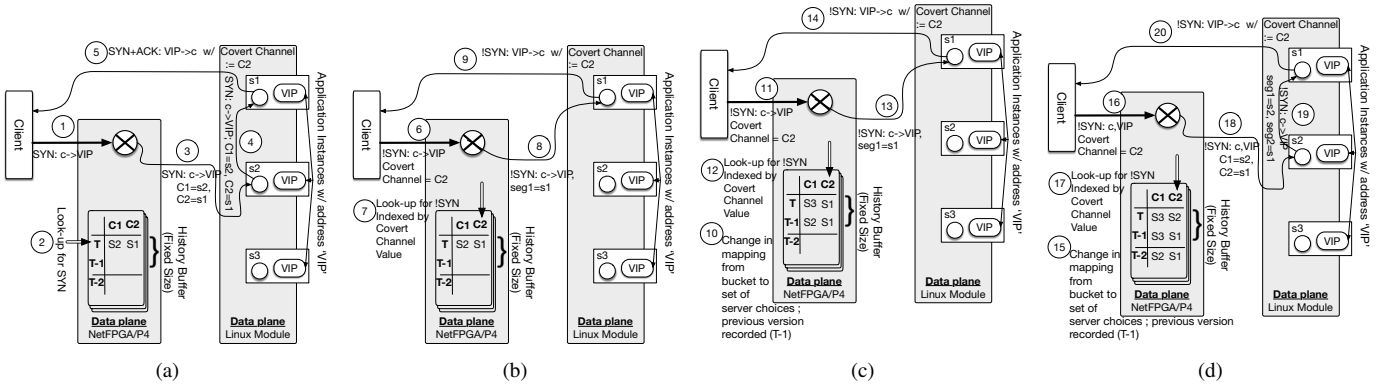

Figure 1. SHELL overview. (a) A SYN packet is steered through $c = 2$ candidate servers (c_1, c_2), in this example (s_2, s_1). Here, the second candidate $c_2 = s_1$ accepts the connection. This choice c_2 is reported in the covert channel of the SYN-ACK packet. (b) Subsequent packets from the client are steered by the LB to the correct server s_1, by looking up the covert channel value c_2 in a consistent hashing table. (c) Upon server reconfiguration, with high probability the assigned server is not modified due to consistent hashing. (d) A server reconfiguration modifies the candidate list to $(c_1^T, c_2^T) = (s_3, s_2)$. SHELL then uses the history matrix to go through previous servers that were c_2 for this bucket – here, $(c_2^T, c_2^{T-1}) = (s_2, s_1)$.

state for a received packet, to forward it to the previous instance.

A shared requirement of [11], [15] is that an application instance is able to direct packets to a second application instance, when needed. For 6LB, only connection establishment packets are proposed to two application instances, allowing load sharing – at the expense of state in the LB for forwarding subsequent packets directly to the correct application instance. Beamer offers packets to two application instances only when there is a change in consistent hashing – but, then, does so for all packets in a flow, to avoid keeping state in the LB.

Thus the question: *is it possible to provide a stateless load-balancer that dispatches queries according to the state of the applications?* This requires the LB to be able to (i) send connection requests through a chain of "candidate" application instances for local connection acceptance decisions, and (ii) statelessly direct packets in an established flow to the one application instance which accepted the connection request.

B. Statement of Purpose

This paper introduces SHELL, an application-agnostic, application-load-aware, stateless load-balancer, which (i) proposes new connections to a *set* of pseudo-randomly-chosen application instances, each making a local acceptance decision, and (ii) "marks" subsequent packets in a flow so as to allow the load-balancer to direct them to the appropriate application instance without requiring the load-balancer to maintain per-connection-state.

The statelessness of the load-balancer makes it a candidate for an implementation in a hardware platform. Thus, this paper proposes a prototype P4-NetFPGA [16] implementation of SHELL targeting the NetFPGA SUME [17] platform, as well as an extensive performance evaluation of the P4 implementation and of the stable hashing algorithm.

C. Paper Outline

The remainder of this paper is organized as follows: section II provides an overview of SHELL, with section III detailing key design aspects. The P4-NetFPGA implementation of the load-balancer is detailed in section IV, followed by a performance evaluation in section V. Resiliency of consistent hashing is evaluated in section VI, before section VII concludes this paper.

II. Overview

SHELL consists of 3 main components: a control plane, a P4-based load-balancing data-plane, and a server agent. An exemple execution is illustrated in figure 1.

The control-plane constructs two tables (see section III-B): (i) a *consistent hashing* table, used to direct new connection request packets (*e.g.* TCP SYNs) to a set of candidate application instances, which improves *fairness*, and (ii) a *choice history* table, for directing subsequent packets in a flow (*e.g.* TCP ACKs), which improves *resiliency*.

The P4 load-balancer uses 5-tuple hashing to map each new connection request to a set of candidate application instances from the *consistent hashing* table provided by the control-plane. Segment Routing (SR) [12] is then used to direct such packets through the selected set of application instances, until one accepts the connection (note that the last application instance in the set must always accept), as in [11]. The position in the list of the application instance which accepted the connection, c_i, is communicated back to the client (see section III-C), which in turn includes it in all further packets from the client to the load-balancer. This enables the P4 load-balancer to send these packets directly to the application instance handling the connection. When a change in the set of application instances causes modifications to some of the *consistent hashing* buckets, changes are saved in a *history*. The P4 load-balancer then directs (using SR) subsequent packets to the current and previous application instances associated with the value c_i received in the packets.

Finally, the server agent (i) accepts new connection requests or forwards them to the next candidate application instance, and (ii) forwards further packets until reaching the application instance that accepted the connection. This server agent is

Algorithm 1 Consistent hashing history table construction

▷ *update version numbers*
for $b \in \{0,\dots,B-1\}, i \in \{h-1,h-2,\dots,1\}, j \in \{0,\dots,c-1\}$ **do**
  $t[b][i][j] \leftarrow t[b][i-1][j]$
end for
▷ *build new consistent hashing table*
for $b \in \{0,\dots,B-1\}, j \in \{0,\dots,c-1\}$ **do**
  $t[b][0][j] \leftarrow consistentHashing(b,j)$
end for
▷ *remove duplicates*
for $b \in \{0,\dots,B-1\}, j \in \{0,\dots,c-1\}$ **do**
  if $t[b][0][j] = t[b][i][j]$ for some $i \neq 0$ **then**
    delete $t[b][i][j]$ and shift $t[b][i+1,\dots,h-1][j]$ upwards
  end if
end for

Table I
EXAMPLE ENTRY OF THE HISTORY MATRIX, FOR A GIVEN BUCKET

	Choice 1	…	Choice c
Epoch t	s_3	…	s_4
Epoch $t-1$	s_2	…	s_{13}
⋮	⋮	⋮	⋮
Epoch $t-h+1$	s_{13}	…	s_{10}

implemented as a Linux kernel module; the details hereof are out-of-scope for this paper.

III. DESCRIPTION

In this section, c is the number of candidate application instances, through which connection requests are directed, B the number of buckets used in consistent hashing, h the *"depth"* of the consistent hashing *"history matrix"* maintained for a bucket $t[b]$ (an example of $t[b]$ for a bucket b is shown in table I).

A. Data Plane

The behavior of the P4 load-balancer and of the application instance depends on which packet is received:

1) A connection request (TCP SYN) received at LB is hashed, on its 5-tuple, into a bucket with index b. The first row of the history matrix for b, $t[b][0][:]$, is then used for generating an IPv6 SR header [18] with a segment list of $(t[b][0][0],\dots,t[b][0][c–1])$ followed by VIP – in the example from table I: $(s_3,\dots,s_4,VIP)$.

2) A connection request (TCP SYN) received at application instance is processed by the server agent – which examines its local state (*e.g.* its CPU load), and determines to either accept the request, or to forward the packet to the next segment in the SR header. Note that the last candidate in the list must always accept the connection.

A server agent having accepted a connection will record, for the connection lifetime, its own index c_i in the segment list received with the connection request.

3) When an application instance accepts a connection, it will embed the recorded c_i for that connection in the TCP SYN-ACK – and in *all future outgoing packets for that connection*, using a covert channel – how this is accomplished is discussed in section III-C.

4) All subsequent (i.e. TCP non-SYN) packets sent by the client will also encode c_i, again using a covert channel.

5) All subsequent (i.e. TCP non-SYN) packets received at the LB are hashed, on their 5-tuple, identifying a bucket b. The LB also extracts the value c_i from the covert channel. The c_i-th colum of $t[b]$ is then used to generate an IPv6 SR header with segment list $(t[b][0][c_i],\dots,t[b][h–1][c_i])$ followed by the VIP – in the example from table I: $(s_3,s_2,\dots,s_{13},VIP)$.

6) All subsequent (i.e. TCP non-SYN) packets received at the application instance will be examined, and if corresponding connection-state is found, will be processed locally – otherwise, are forwarded to the next segment[1].

B. Control Plane - Consistent Hashing Table Computation

The algorithm described in [11, algorithm 3] is used to generate, for each bucket b, a *candidate list* of application instances $\ell = (s_1,\dots,s_c)$. This list is recorded as the first row of the history matrix for the bucket: $t[b][0][:] = \ell$.

When the set of application instances is modified, all entries in the history matrix are offset by one (*i.e.* $\forall b, t[b][i+1][:] \leftarrow t[b][i][:]$), and the new mapping of candidate application instances for each b is calculated, as per [11, algorithm 3].

Since consistent hashing will leave most bucket entries unmodified after server reconfigurations [8, figure 12] [11, figure 7], *i.e.* $t[b][0][:] = t[b][1][:]$ for most b, duplicate history entries are removed as per algorithm 1.

C. Possible covert channels

The server-state-aware and stateless load-balancing approach described in this paper relies on the application instance being able to inform clients to include c_i (the position of the application instance in the segment list at time of connection establishment) in all packets subsequent to the connection request. This c_i is used by the load balancer to direct these to the appropriate application instance.

If it is possible to modify the client networking stack so as to cooperate with the load-balancing architecture, a simple option would be to embed the identifier (rather than the value of c_i) of the application instance having accepted the connection in packets sent from the application instance, then reflect this identifier in the packets sent by the client. This can be achieved with transport protocols such as QUIC, which embed a connection identifier chosen by servers and reflected by clients.

However, for TCP connections, it is necessary to convey c_i from the application instance to the client, and make the client relay c_i in subsequent packets, without client-side modification. This requires using a *covert channel from the application instance, through the client, and to the application instance*, which the client neither inspects nor interferes with. Approaches accomplishing this include:

TCP sequence numbers, which are reflected by endpoits in the acknowledgement field. For connections shorter than $1/c$

[1] In the unlikely event that there is no next segment, the packet is dropped – this corresponds to the case where the history was not long enough to include the application instance that had accept the connection in the first place.

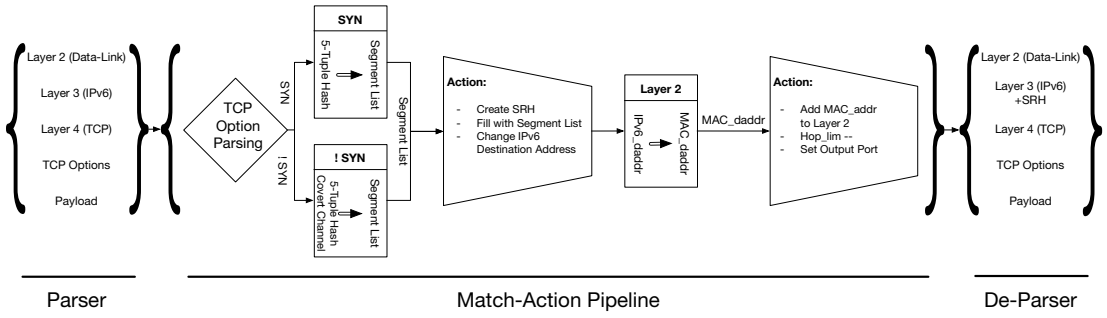

Figure 2. SHELL P4 data-plane overview.

```
header tcp_opts_24_t {
  bit<32> nopnoptypelength1; bit<63> value1; bit<1> potential_cov_channel1;
  bit<32> nopnoptypelength2; bit<63> value2; bit<1> potential_cov_channel2;
}
state parse_tcp_opts_24 {
  packet.extract(hdr.tcp_opts_24);
  transition select(hdr.tcp_opts_24.nopnoptypelength1) {
    32w0x0101080a : set_covert_channel_24_1; /* NOP NOP TS (type=08, length=0a) */
    32w0x0101050a : parse_tcp_opts_24_2; /* NOP NOP SACK10 (type=05, length=0a) */
    default: accept;
  }
}
state set_covert_channel_24_1 {
  user_metadata.covert_channel = hdr.tcp_opts_24.potential_cov_channel1;
  transition accept;
}
state parse_tcp_opts_24_2 {
  transition select(hdr.tcp_opts_24.nopnoptypelength2) {
    32w0x0101080a : set_covert_channel_24_2; /* NOP NOP TS (type=08, length=0a) */
    default: accept;
   }
}
state set_covert_channel_24_2 {
  user_metadata.covert_channel = hdr.tcp_opts_24.potential_cov_channel2;
  transition accept;
}
```

Figure 3. Example TCP TLV parsing in the P4 LB, for $d_{off} = 11$.

of the sequence number space (*i.e.* $2^{32}/c$ bytes), this allows implementing the covert channel through the high-order bits of the sequence number. For longer connections, a flow table will be however required.

TCP timestamps, which are also reflected by TCP endpoints. The covert channel can be carried in the low-order bits of the TCP timestamp, as is done in [19] – and according to [20], [21] is resilient to middlebox processing. Modifying the timestamp by at most $c-1$ units ($c = 2$ usually [10]) will have a negligible effect on RTT estimation.

To avoid using a flow table, SHELL uses **TCP timestamps** as covert channel for TCP connections: the server agent of the accepting application instance encodes its index c_i in the low-order bits of the TCP timestamp – and the LB inspects TCP timestamp sent by clients.

IV. P4 Load-Balancer Implementation

The workflow of the P4 implementation of the LB data-plane of SHELL is illustrated in figure 2. It uses three match-action tables, corresponding to (i) segment lists to be inserted into SYN packets, as given by the first row of the history matrix, (ii) segment lists to be inserted in non-SYN packets, as given by columns of the history matrix, and (iii) a Layer-2 lookup for output packets.

Processing a packet starts with parsing its headers. If they do not match the expected format (*e.g.* the packet uses UDP), the packet is dropped; otherwise, they are processed by the match-action pipeline. Both (i) the 5-tuple hash of the packet, computed by way of an external function from the P4-NetFPGA framework, and, (ii) in case of a non-SYN packet, the covert channel value as found in the TCP timestamp field (see below), are used as keys to the match tables, to access the corresponding segment list. The segment list is then fed to an action, which builds the SR header. To complete the match-action pipeline, a match is performed on the newly-added destination address of the packet (*i.e.* the first SR segment), which calls an action making the packet egress through the correct interface. Finally, the de-parsing stage emits the packet with the newly-built headers.

For want of a lookahead function and of variable-length header support in the compiler, only a set of "reasonable options" is parsed (specifically, SACKs and timestamps, *i.e.* those that can be found in non-SYN packets according to [20]). Depending on the length of the TCP header found in the "data offset" (d_{off}) field, an option header of suitable type is parsed, corresponding to lengths $d_{off} \in \{8, 11, 13, 15\}$. Figure 3 depicts an example of such a parsing, when $d_{off} = 11$. The bits of the covert channel are then stored into a meta-data field[2]. A parameter of interest to the performance of the data-plane is the maximum size of the parsed TCP header d_{off}^{max}: its influence on the performance is evaluated in section V.

V. P4-LB Implementation Performance

A key element to the performance of SHELL is the per-packet latency, incurred in the P4-dataplane of the LB. It depends primarily on two factors: (i) the latency incurring when receiving a packet over an ingress interface, inserting an SR header, and transmitting this (now larger) packet over an egress interface (section III-A), and (ii) the latency incurring from extracting c_i from within a TCP option (TCP timestamp, section III-C).

These factors are evaluated using the P4-NetFPGA framework and the Xilinx Vivado software suite, simulating packets going through one interface of a 10G NetFPGA-SUME, and are documented in this section.

[2]To ensure compatibility with clients not using TCP timestamps, if a SYN packet is missing the TCP timestamp option (detected by $d_{off} \leq 8$), an SR header with only one candidate will be inserted. Lack of a TCP timestamp in non-SYN packets is then interpreted as a covert channel value of $c_1 = 1$.

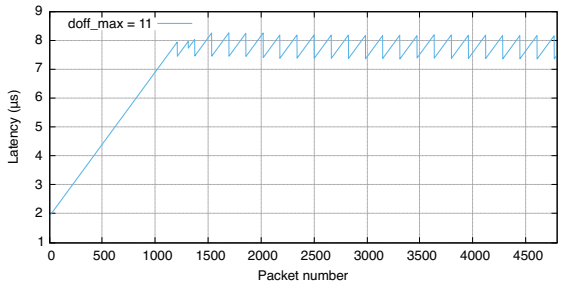

Figure 4. SHELL P4 dataplane evaluation: per-packet latency for a burst of 4800 packets ($c = 2, h = 2, d_{off}^{max} = 11$)

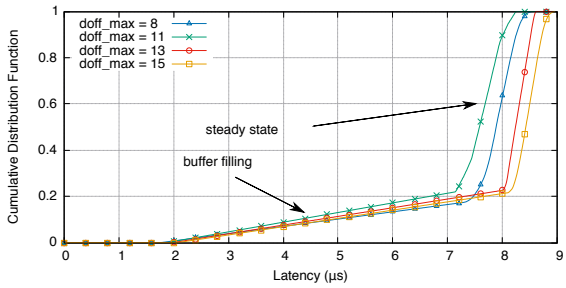

Figure 5. SHELL P4 dataplane evaluation: distribution of per-packet latency for a burst of 4800 packets ($c = 2, h = 2$, different d_{off}^{max})

Table II
P4-NETFPGA DATAPLANE PERFORMANCE

Throughput (Mpps)	59.8
Worst-case latency (μs)	8.96

Table III
P4-NETFPGA DATAPLANE RESOURCE USAGE

Max TCP size	LUT	LUTRAM	FF	BRAM
$d_{off}^{max} = 8$	36.9%	19.4%	33.3%	59.3%
$d_{off}^{max} = 11$	40.1%	22.0%	36.4%	63.2%
$d_{off}^{max} = 13$	43.8%	24.9%	40.2%	67.7%
$d_{off}^{max} = 15$	48.7%	28.6%	45.8%	74.1%

A. Effect of SR Header Insertion on Latency

For the purpose of the simulations, $c = 2$ choices were used for SYN packets, and a history of $h = 2$ was used for ACK packets, thus SR headers with 3 segments were inserted, increasing the packet size by 56 bytes between ingress and egress. Feeding ingress packets at maximum line-speed fills the egress interface queue over time, thus progressively increasing packet forwarding latency – as depicted in figure 4, showing stable results from above 1500 packets. When the egress buffer is empty, the latency is 2.1 μs, and when the buffer is full it oscillates between 8.2 μs and 9.0 μs. Thus, for subsequent simulations, batches of 4800 packets are injected at line-rate on the ingress interface.

Table II reports the throughput and worst-case latency obtained: SHELL, comparable to Beamer [15], can sustain 60 Mpps, *i.e.* 22× as much as what is reported for the single-core software implementations of Maglev [8], while also providing application instance load-awareness.

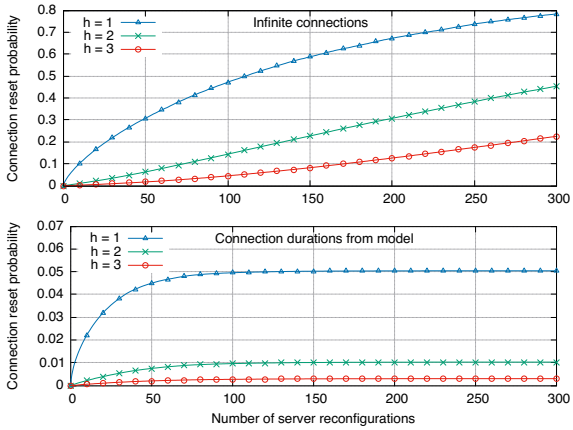

Figure 6. SHELL consistent hashing evaluation: probability for a connection started at time $t = 0$ to be reset after n server reconfigurations. $B = 35591$ buckets, 500 servers, $c = 2$ choices, different history depths h. Top: infinite connections. Bottom: connection duration and server reconfiguration rate model from [13], [22].

B. Effect of TCP Parsing on Latency and FPGA Resources

As explained in section IV, the TCP timestamp option is parsed by matching a predefined set of option headers, thus the greater d_{off}^{max} (maximum admissible size of parsed TCP headers), the more program branches – and the greater the latency. This is evaluated by testing $d_{off}^{max} \in \{8, 11, 13, 15\}$, and depicted in figure 5. The latency varies from 1.8 μs ($d_{off}^{max} = 8$) to 2.1 μs ($d_{off}^{max} = 15$) for an empty egress queue, and its average after the egress queue has filled up goes from 7.7 μs ($d_{off}^{max} = 11$) to 8.5 μs ($d_{off}^{max} = 15$).

This allows noting that, in a controlled environment (*e.g.* a data-center, where SHELL typically would be deployed), where a strict set of TCP options can be enforced, it is possible to trade off parsing safety against reduced latency. However, parsing more potential options increases the amount of logic consumed on the FPGA. As reported in table III, which depicts the resource usage on the FGPA for different values of d_{off}^{max}, this can increase LUT usage by up to one third.

VI. CONSISTENT HASHING RESILIENCY

Long-lived connections are particularly vulnerable to application instance reconfigurations – thus, as a *worst-case* scenario, simulations using *infinite-length connections*, and subject to successive application instance removal/insertions, are performed; each simulation is repeated 5 times.

For a random connection, assuming that it was uniformly drawn from among the $B \times c$ buckets and choices possible, the probability that it is reset[3] after n application instance reconfigurations, is depicted in figure 6 (top), which shows the impact of the history depth h, and figure 7 (top), which shows the impact of the number of buckets B.

[3]A connection assigned to a bucket/choice pair is deemed reset after n reconfigurations if the application instance to which it was initially assigned no longer appears in the corresponding history table.

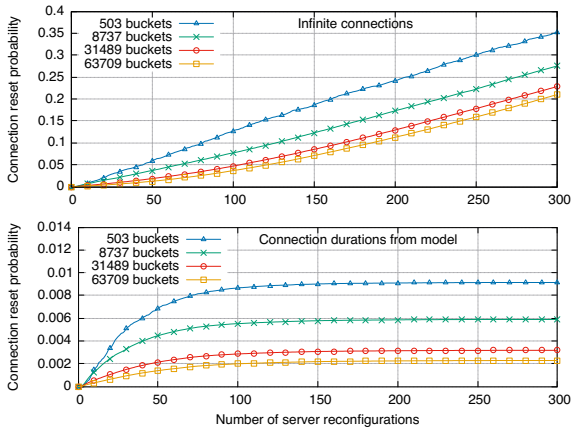

Figure 7. SHELL consistent hashing evaluation: probability for a connection started at time $t = 0$ to be reset after n server reconfigurations. 500 servers, $c = 2$ choices, $h = 3$, different number of buckets B. Top: infinite connections. Bottom: connection duration and server reconfiguration rate model from [13], [22].

In realistic scenarios, application instance reconfigurations are expected to be rare, and connections are unlikely to last for more than a few application instance reconfigurations – at most. The bottom graphs of figures 6 and 7 depict the connection reset probability using the connection duration distribution model from [22, Figure 7.a], and with the application instance reconfiguration rate distribution taken as the *highest 1% rates over a month for 100 clusters* from [13, Figure 2] – with a median reconfiguration rate of $13.5\ \mathrm{min}^{-1}$.

In these conditions, less than 1% of the connections were lost when using SHELL with a non-void history ($h > 1$). Without history (*i.e.* $h = 1$, as would be the case in [8], [11]), more than 5% of the connections were lost.

VII. Conclusion

This paper has introduced SHELL, an application-agnostic load-balancing architecture which combines application load awareness (by using a *power-of-choices* scheme upon connection establishment), statelessness (by using a *covert channel* to indicate which of the candidates had accepted the connection), and resiliency (by using *consistent hashing* and *versioning*). Being stateless makes SHELL suitable for a hardware implementation, as demonstrated in this paper through prototype development using the P4-NetFPGA framework. An evaluation of throughput and latency of this prototype implementation shows that the attainable performance is equal to that of other hardware implementations, while providing application-awareness and therefore improving load-balancing fairness. Further, simulation of the consistent hashing resiliency shows that the number of long-lived connections dropped, even in worst-case scenarios, is negligible.

Acknowledgements

This work was supported, in part, by the Cisco-Polytechnique chair "Internet-of-Everything" (https://www.internet-of-everything.fr/).

References

[1] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[2] N. Dragoni *et al.*, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[3] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[4] 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.

[5] D. Karger *et al.*, "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.

[6] ——, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11, pp. 1203–1213, 1999.

[7] 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.

[8] D. E. Eisenbud *et al.*, "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.

[9] P. Patel *et al.*, "Ananta: cloud scale load balancing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[10] 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.

[11] Y. Desmouceaux *et al.*, "6lb: Scalable and application-aware load balancing with segment routing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 819–834, April 2018.

[12] C. Filsfils *et al.*, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[13] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 15–28.

[14] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[15] V. Olteanu *et al.*, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018, pp. 125–139.

[16] P4.org, "P4 → NetFPGA: A low-cost solution for testing P4 programs in hardware," 2017. [Online]. Available: https://p4.org/p4/p4-netfpga-a-low-cost-solution-for-testing-p4-programs-in-hardware.html

[17] N. Zilberman *et al.*, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[18] S. Previdi *et al.*, "IPv6 Segment Routing Header (SRH)," Internet Engineering Task Force, Internet-Draft draft-ietf-6man-segment-routing-header-13, May 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-6man-segment-routing-header-13

[19] F. Duchene and O. Bonaventure, "Making multipath tcp friendlier to load balancers and anycast," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 2017, pp. 1–10.

[20] A. Medina, M. Allman, and S. Floyd, "Measuring interactions between transport protocols and middleboxes," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 336–341.

[21] M. Honda *et al.*, "Is it still possible to extend tcp?" in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011, pp. 181–194.

[22] A. Roy *et al.*, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 123–137.