Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Advanced Topics in
Communication Networks (Fall 2018)

Group Project Report

“Load-aware” L4 load balancing

Author:
Kamila Součková
Advisor: Alexander Dietmüller

Supervisor: Prof. Dr. Laurent Vanbever

Submitted: 23 December 2018
9 Pages

Abstract

As traffic to a service grows, the ability to scale horizontally across a pool of application servers presents substantial performance benefits. Traditional software-based load balancers are unable to cope with more than several million packets per second. This is unacceptably limiting in the face of networks able to deliver billions of packets per second. Furthermore, load balancers usually ensure a uniform distribution of requests, which does not necessarily correspond to a uniform distribution of server load. This causes a sub-optimal resource utilisation and leads to reduced performance. This report explores a novel load-balancing solution based on software-defined networking (SDN) with a dynamically adjustable distribution of traffic able to deliver line-rate performance with comparatively little implementation effort.

Contents

1 Introduction

L4 load-balancers enable horizontal scaling of traffic for a service across multiple application servers. They operate by mapping a *virtual IP address* (VIP) to a pool of *direct IP addresses* (DIPs), or the actual application servers: when a request arrives to the VIP, the load balancer forwards it to one of several servers in the DIP pool. This is critical to performance, especially for large services, because with load balancing what appears to be a single endpoint can process more requests at the same time. However, the load balancer quickly becomes a bottleneck: a high-end software load balancer can forward a few Mpps, while the underlying network throughput can easily reach several Gpps.[3]

Furthermore, load balancers typically select servers from the pool uniformly, which can pose problems in certain scenarios. In case of long-running requests, or if the servers are heterogeneous, (whether it is because of a different hardware configuration, or because they run multiple applications), uniformly distributed requests often do not correspond to uniformly distributed load. It would be very useful to take into account some server metrics, such as server load or mean request latency, when making load balancing decisions.

Therefore, this project proposes a load balancer based on software-defined networking (SDN), able to perform entirely in the data plane (i.e. at line rate), which can additionally forward the requests to the application servers with an arbitrary, dynamically adjustable distribution.

The distribution can then be derived at real-time from application server metrics such as server load or request latency.

2 Background and related work

L4 load balancing in general works as follows:

1. Match an incoming packet's destination IP address and destination port against the load balancer's Virtual IP addresses and map to a server pool.

2. Select a DIP from the pool. This can be done e.g. in a round-robin fashion, or by hashing the five-tuple. This normally results in a uniform distribution of requests to servers.

3. Rewrite the destination address and port.

4. Forward the packet to the selected server.

5. Handle the return path correctly: rewrite the source IP address and source port back to the Virtual IP address on replies for this request, so that it appears that the response comes from the load balancer.

The challenge when implementing a P4-based load balancer is **per-connection consistency**, i.e. making sure that all packets of a given connection are forwarded to the same application server. Unlike in software, a hardware-based load balancer cannot afford to keep much state (because that would reduce performance), and therefore it is not trivial to ensure that all packets of the same connection will be forwarded to the same server when the pool changes. Pool changes are quite frequent, as especially in large data centres servers come offline or online multiple times per second.[5]

SDN-based load balancing has been explored in [6]. The paper focused on performance, scalability, and the ability to handle frequent changes to the DIP pools. The approach in this paper is highly optimised and rather clever. While their way of ensuring per-connection consistency has a smaller memory footprint than ours, we found it valuable to create a simpler approach. We enjoyed seeing how SDN enables solving the same problem in very different ways, with different trade-offs.

To our knowledge, SDN-based load balancing with an adjustable distribution has not been explored before.

3 Implementation

3.1 Guiding principles

During the implementation, we dedicated substantial effort to keeping the high-level structure clean: the different components are in clearly separated, well decoupled modules. This is true not only, but especially for code dealing with different network layers: the L2 switching, “L2.5” ARP, and L3 routing are completely separate and each layer can be freely exchanged with another implementation without modifying the other layers (as it should be). We use this approach both for the controller code (which uses various modularity-friendly features of Python such as modules, classes and inheritance), and for the P4 code, to the extent possible by the current “flat” file structure needed by the P4 compiler.[1] Thanks to this disciplined way of working, the individual components can be thought about, implemented, debugged, and tested separately. This allowed us to progress quickly when unexpected challenges arose, as changing existing code was comparatively easy due to the clean separation of concerns.

3.2 L3 and below: a simple router

A load balancer’s function is ultimately to rewrite the destination IP and port and then send the traffic to that server. Therefore, up to L3 it performs standard routing. For this project we implemented simplified routing which has its ARP and MAC tables pre-filled by the controller instead of running the ARP protocol and L2 learning. The following describes our routing implementation.

3.2.1 L3: routing

The packet’s destination IP address is matched in the router’s *routing table*, using a longest-prefix match (LPM). The packet may either be addressed to a host the router is directly connected to (on some interface), or it may need to be sent to a different network, through a gateway (via some interface).

Therefore, the routing table maps a prefix to either of:

- a next hop through a gateway and an interface, or
- a direct connection through an interface.

```
routing_table : Prefix -> NextHop (GatewayIP, Interface) | Direct Interface
```

If there is no match (no route to host), we drop the packet.

Note that the next hop’s IP address exists only in the router’s memory: it does not appear in the packet at any time.

[1]Looking back, the author would today write the controller code with much less inheritance and much more explicit function composition: somewhat in the spirit of [4]. Nevertheless, even this imperfect approach was good enough to save a lot of time in the later stages of development. Anything that keeps things decoupled pays off eventually.

3.2.2 "L2.5": ARP

We now know the IP address and interface of the next hop. (Note that this is a host that is connected to us directly—it is on the same wire segment.) We need to translate this into an L2 MAC address in order to pass it to L2. This mapping is stored in the *ARP table*.

```
arp_table : (IPv4Address, Interface) -> MACAddress
```

Note: Interface conceptually belongs there, but the IP address should in fact be unique. Our code leaves out the interface for simplicity.

If there is no match, i.e. when we dont know the MAC address for the IP address, the control plane would normally send an *ARP request* (and drop the packet). In our case, we opted to simplify by pre-filling the ARP table from the (known) topology instead.

IPv6 uses NDP instead of ARP. While the protocol is different, the data-plane table is conceptually the same.

3.2.3 L2: switching

Here we get a packet with some destination MAC address, and we need to decide on which port we should put it. We use a MAC table to do it:

```
mac_table : MACAddress -> Port
```

Real switches are somewhat more complicated: for example, redundant links mean that a MAC address may exist on more than one port. We do not support such scenarios.

The MAC table is normally filled at runtime by learning source MACs from packets. In our case, we pre-fill it from the known topology for simplicity.

3.2.4 The control flow: putting it together

The router applies the tables described above as follows:

1. Apply routing. Find the next hop (either gateway or direct).

2. Apply ARP translation to the "next hop" host. Fill out the destination MAC address in the packet.

3. Apply the MAC table to find the right port for this destination MAC address. Send it out.

3.3 L4: Simple load balancer

As already hinted, a simple load balancer handles incoming packets as follows:

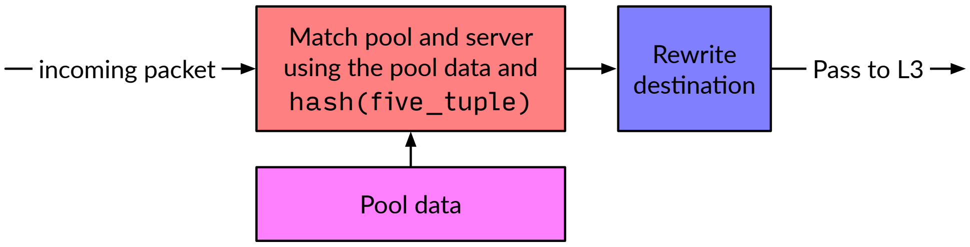

incoming packet
Match pool and server using the pool data and hash(five_tuple)
Rewrite destination
Pass to L3
Pool data

Figure 1: Flow diagram of the simple load balancer

1. Match the packet's destination IP address and port against the *VIP table*. If it matches, this is a packet we should load balance. If it does not, we skip the following steps. This table's match action sets not only the pool ID, but also the pool size.

2. Compute a hash of the five-tuple, modulo pool size. The five-tuple identifies a connection, therefore the hash will be the same for all packets of the same connection.

3. Match the pool ID and the hash to the *DIP table*: select a specific server. If the hash is sufficiently uniform, the server will be selected uniformly at random.

4. Rewrite the packet's destination IP address and port to the selected server's IP address and port.

5. Pass to L3 for routing to the server. Note that now the packet's destination is that server, so L3 can handle it without awareness of our rewriting.

Care must be taken to also rewrite packets on the return path, so that they appear to come from the load balancer. In our case, we add two more tables to handle this, and we apply these only if the VIP table did not match.

1. Match the packet's source address and port against the *inverse DIP table*. This table's match action sets the pool ID.

2. Match the pool ID against the *inverse VIP table*. This table's match action rewrites the source address and port to the appropriate VIP.

3.4 Changing the distribution

A simple way to change the distribution of requests with minimal code changes is to add an entry for a server multiple times (with different hashes):

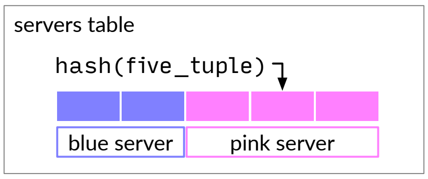


Figure 2: Multiple buckets per server

This allows to change the ratios of requests that land in different servers, at the cost of increased memory usage: the table size will now be $\sum$ weights. This is what we implemented in our project.

Memory usage could be improved by using something other than an exact match:

- The P4 standard specifies an LPM match kind, which could be leveraged to decrease the table size to $\ln \sum$ weights by splitting the bucket sizes to powers of two and adding one entry per power of two, as shown in Figure 3.

- the V1 model offers a range match kind: this would keep the table size independent of the weights.

We did not implement this in the project, but it would be a trivial change.

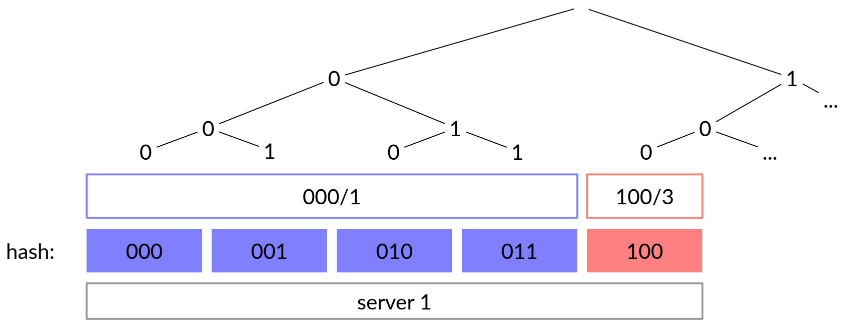


Figure 3: Decreasing table size by using LPM instead of exact match

3.5 Per-connection consistency

3.5.1 The problem

If we implement what has been described so far, we will get a functional load balancer with weights, but we cannot change the weights (or the servers in pools) at runtime without losing per-connection consistency. When the pool distribution changes, some buckets will be assigned to different servers and therefore connections in those buckets will be broken (see Figure 4).

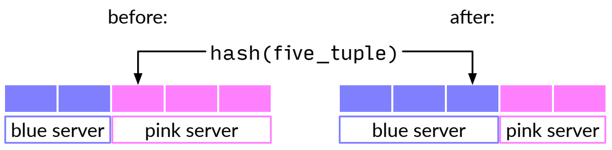
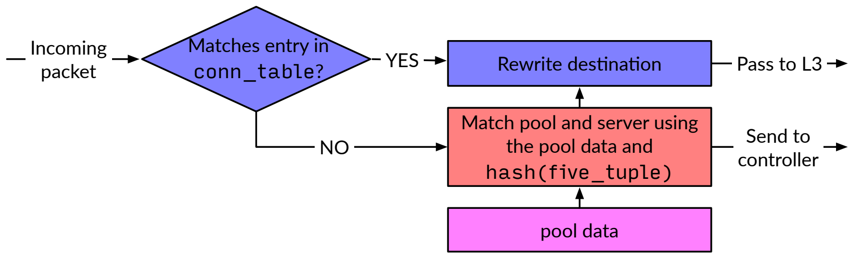


Figure 4: Changing the distribution will break connections which hashed into the third bucket.

3.5.2 Fix #1: saving server assignment in a connections table

The obvious fix is to remember the server for a connection instead of re-selecting it for every packet. The flow would then be as follows:



Incoming packet
Matches entry in conn_table?
YES
Rewrite destination
Pass to L3
NO
Match pool and server using the pool data and hash(five_tuple)
Send to controller
pool data

Figure 5: Flow diagram with the connections table

We need to use a table (with exact match on the 5-tuple), not SRAM, because we need an exact match to not break anything.

The problem with this approach is that we cannot write to the table instantly (as table writes are done by the control plane). Therefore, although this approach allows us to remember connections after some time, there is a “dangerous window” between the first packet of the connection and the completion of the table write: see Figure 6.

3.5.3 Fix #2: Closing the “dangerous window”

The dangerous window is dangerous because we have discarded the old data. Therefore, we can get around the problem by not discarding it until all pending connection writes using it have

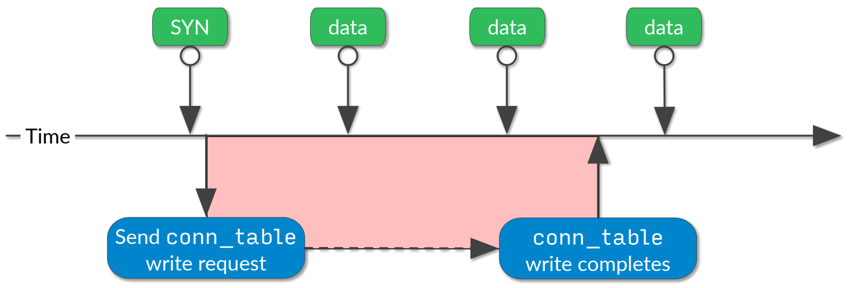


Figure 6: “Dangerous window” where we could still choose the wrong server

been written into the connections table.

Versioned tables To be able to keep both old and current data in the same P4 table, we have created a universal “versioned table” abstraction:

- the P4 code adds an extra `meta.version` key (exact match)
- the P4 code reads a versions register and stores the value in `meta.version`
- the controller writes to the register to tell the P4 switch which version should be currently active
- the controller may use our `VersionedP4Table` abstraction, which looks like a normal P4 table, but has an extra `commit()` method and handles versions (i.e. adds the extra key and writes to the register when needed)

Note that in addition to enabling us to store old data, this also enables transactions (even across tables, if multiple tables use the same version register): the controller can make changes in a “scratch” version and commit atomically by a single register write.

Determining which table version goes with a packet When a packet arrives (and it does not match the connections table), we need to find out whether we should use the current pools version or some old version (and which). Abstractly, we want to associate the five-tuple with a version. We do not have the usual data structures used for storing key-value pairs in P4, so we needed some creativity to solve this sub-problem. The key idea is that we do not need a lot of versions at the same time: the “dangerous window” is rather short ($\ll 500\,\text{ms}$), so a very small number of version is always enough. We chose to support four versions in our implementation (including the “scratch” one, so three versions are usable at any time: the current one and two old ones.)

Now, instead of storing key-value pairs, we can query the other way around: “for each version: did we use this version to select the server for this connection?” These are set membership queries (for each version, we have the set of the connection five-tuples using that version), and therefore they can be implemented using Bloom filters.

Therefore:

- the P4 switch needs four Bloom filters with connection five-tuples
- for an incoming packet:
  - by default, set version to the value in the versions register
  - for i in $\{0, 1, 2, 3\}$: if the five-tuple is in Bloom filter i: set version to i

After the Bloom filters, the version is set correctly.[2] Therefore, we can match against our VIP and DIP tables (which contain the versioned data) and the server will be selected correctly.

To handle pool table versioning correctly, the controller needs to keep track of outstanding connections table writes and wait for their completion before overwriting an old version. Therefore, the table writes must be done asynchronously. We used the Twisted Python framework[2] for writing event-driven code, and we created an asynchronous abstraction over P4 tables (which also enables easy introspection). We also had to re-write the rest of the controller to be asynchronous, which was not trivial, but it paid off, as described below.

Bloom filters implementation hurdles We encountered several entertaining problems when implementing the Bloom filters:

1. P4 does not have multi-dimensional arrays (even though constant-size arrays could be unrolled at compile time, so it could have them).
2. Not having multi-dimensional arrays, which are just syntactic sugar for offset calculations, we chose to create a single register array and calculate the offsets ourselves. However, as we later discovered, even though an API to clear a register array partially exists, it is in fact not fully implemented: an exception is thrown erom deep inside the `runtime_API` Python code when a pair of indices is passed. We tried clearing the register array cell by cell, but unsurprisingly this was very slow: it took about 8 seconds to clear the 8000 cells that we use for the Bloom filter.
3. Being unable to do this nicely due to the above, we settled on using the C preprocessor to generate four Bloom filter register arrays and four pairs of functions identical except for a few occurrences of the numbers 0-3. This worked, but not before encountering three different compiler bugs: one related to nested structs, and two cryptic enough to discourage any further investigation.
4. The Python control-plane API communicates with the switch in a blocking and thread-unsafe manner. Therefore, we had to run it in a separate thread to avoid blocking the main event loop, and synchronise the calls to avoid race conditions.

3.6 Putting it all together

With the connections table, versioned pool tables, and version-checking Bloom filters, the load balancer can preserve per-connection consistency. The final flow is depicted on Figure 7.

4 Evaluation

4.1 Integration tests

To make sure that our load balancer works correctly, we created a "mini test framework" for our P4 switch using `pytest`[1]. This mini-framework enables us to write full integration tests: it uses `p4run` to create a Mininet network and start the P4 switch, and it can perform end-to-end tests by telling the Mininet hosts to run anything from `ping` and `netcat` to custom servers and clients controlled via RPC.

Thanks to using the Twisted framework, which makes it trivial to combine multiple event loops, we were able to write very specific tests which show where exactly we have a problem. Though the initial time investment into creating the mini test framework and rewriting everything into Twisted was substantial, it clearly paid off during late-stage refactoring.

[2]With a very high probability, as Bloom filters are probabilistic.

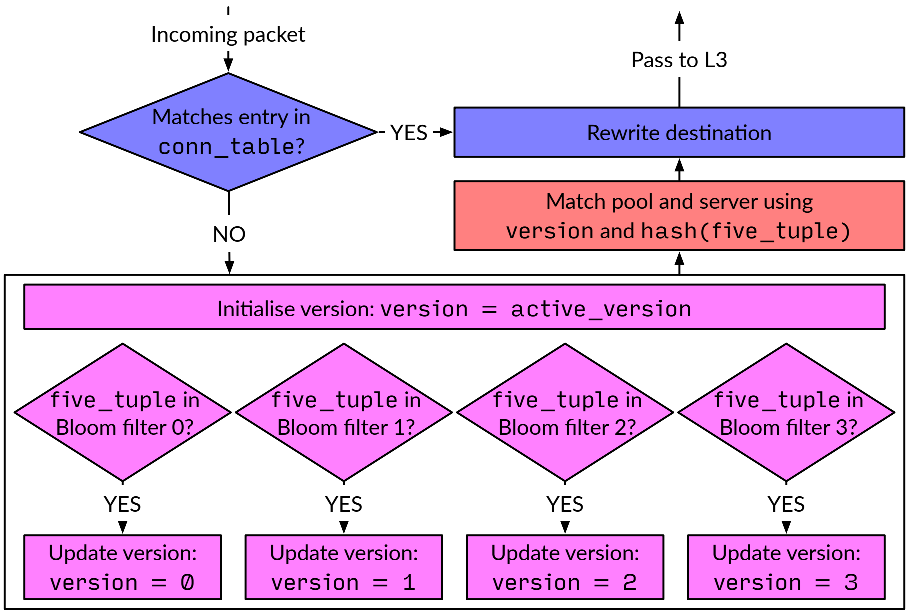

Incoming packet
Matches entry in conn_table?
YES
NO
Pass to L3
Rewrite destination
Match pool and server using version and hash(five_tuple)
Initialise version: version = active_version
five_tuple in Bloom filter 0?
five_tuple in Bloom filter 1?
five_tuple in Bloom filter 2?
five_tuple in Bloom filter 3?
YES
YES
YES
YES
Update version: version = 0
Update version: version = 1
Update version: version = 2
Update version: version = 3

Figure 7: Life of a packet

The tests check everything from L2 to not breaking connections, and therefore we can be sure that the load balancer does what it should. Listing 1 shows a sample tests run.

```
vagrant@p4:/project$ pytest
=========================== test session starts ============================
platform linux2 -- Python 2.7.12, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
rootdir: /project, inifile:
plugins: twisted-1.8, timeout-1.3.3
collected 23 items / 2 skipped

test/l2_lazy/blackbox_test.py ..                                     [  8%]
test/l3_router_lazy/blackbox_test.py ..                              [ 17%]
test/l4_loadbalancer/keep_connections_test.py ...                    [ 30%]
test/l4_loadbalancer/versioned_tables_test.py .....s                 [ 56%]
test/l4_loadbalancer-unversioned/connection_breaking_test.py .       [ 60%]
test/l4_loadbalancer-unversioned/l3_still_works_test.py ..           [ 69%]
test/l4_loadbalancer-unversioned/loadbalancing_test.py ...           [ 82%]
test/l4_loadbalancer-unversioned/twisted_loadbalancing_test.py ...s  [100%]

================== 21 passed, 4 skipped in 315.61 seconds =================
```

Listing 1: A sample test session

4.2 Live system simulation

To visualise our load balancer's impact, we wrote a server which simulates load (by reporting a load proportional to the number of concurrent connections). We also created a client for generating load.

We started up four of the servers, each set to simulate a system with a different number of CPUs (and therefore a different ability to handle the load).

We then programmed the load balancer to read this simulated value and set the requests distribution to load^{-1} for each server.[3] To see the effect, the load balancer was initially set to

[3]In the real world, load^{-1} does not work very well, because that way the system is a feedback loop with a delay, which means that oscillations will occur. Nevertheless, this allows us to see that the load balancer works.

balance uniformly (i.e. with all weights set to 1), and later it was switched to take the server load into account and re-distribute the requests accordingly. A sample run of the experiment is on Figure 8.

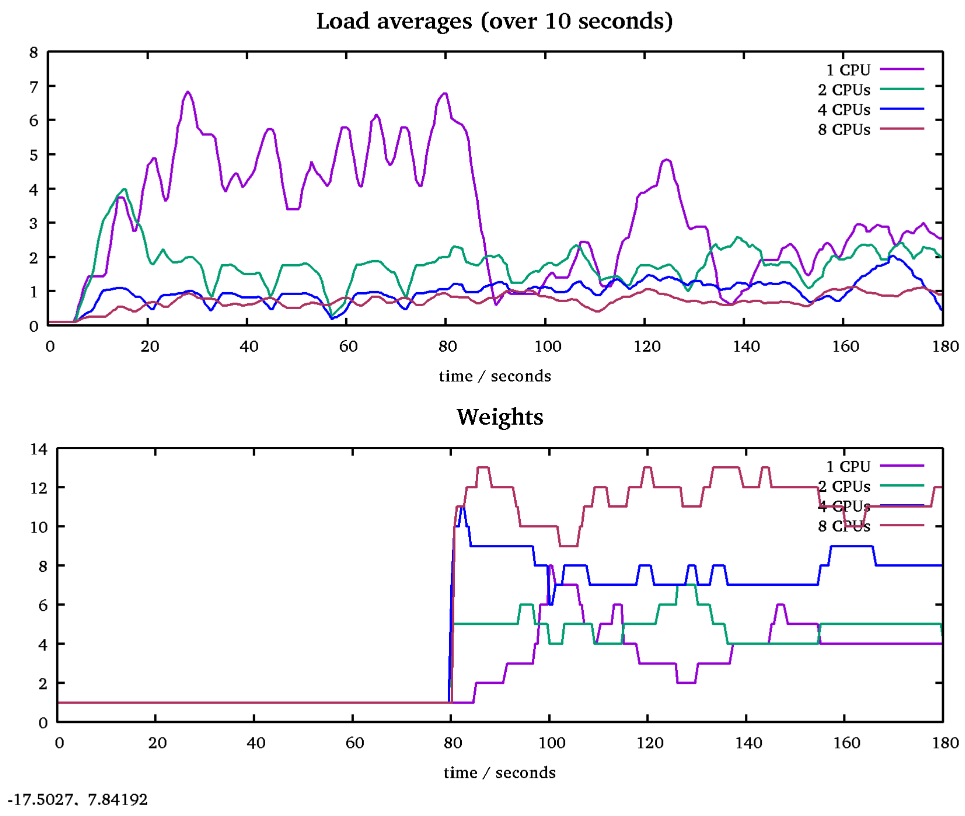


Figure 8: Loads and weights over time

5 Conclusion

We have implemented and tested an L4 load balancer which forwards traffic entirely in the data plane. The distribution of the requests can be freely adjusted at runtime (including adjusting it multiple times per second), and per-connection consistency is ensured. A simulation of a system responding to the server load confirms that we can balance based on server metrics in real time and noticeably decrease the disparity between the individual servers.

During the implementation, we explored various challenges: from the software engineering aspects such as cleanly structuring code, test-driven development, and asynchronous/event-driven programming, through simulating a live environment, to trying cutting-edge things with the P4 compiler.

Apart from the applications in the load balancer, we have contributed also generally applicable original ideas such as integration tests for P4 switches and controllers, versioned P4 tables with transactional semantics, and a probabilistic constant-time key-value store with a small value-space implemented using a number of Bloom filters.

A function usable in production is a control theory question, and is very likely application-specific, so we did not explore this subproblem.

References

[1] pytest test framework documentation. `https://docs.pytest.org/en/latest/`. Accessed: 2018-12-10.

[2] Twisted: an event-driven networking engine for Python. `https://twistedmatrix.com/`. Accessed: 2018-12-10.

[3] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 523–535.

[4] Fackler, A., and Manista, N. The end of object inheritance & the beginning of a new modularity. `https://www.youtube.com/watch?v=3MNVP9-hglc`. Accessed: 2018-12-10.

[5] Govindan, R., Minei, I., Kallahalla, M., Koley, B., and Vahdat, A. Evolve or die: High-availability design principles drawn from Google's network infrastructure.

[6] Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM 17* (2017).

A Group organization

The original group split after about two weeks.

Most of this project was done by **Kamila Součková**.

Others' contributions are:

Nico Schottelius Wrote the packet parsers (Ethernet, IPv4, IPv6, TCP).

Sarah Plocher Wrote the first version of the L2 switch. None of the code is present now.