# Stateless Datacenter Load-balancing with Beamer

Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu,
*University Politehnica of Bucharest*

https://www.usenix.org/conference/nsdi18/presentation/olteanu

**This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).**

**April 9–11, 2018 • Renton, WA, USA**

ISBN 978-1-931971-43-0

# Stateless Datacenter Load-balancing with Beamer

Vladimir Olteanu, Alexandru Agache, Andrei Voinescu and Costin Raiciu
*University Politehnica of Bucharest*

## Abstract

Datacenter load balancers (or muxes) steer traffic destined to a given service across a dynamic set of backend machines. To ensure consistent load balancing decisions when backends come or leave, existing solutions make a load balancing decision per connection and then store it as per-connection state to be used for future packets. While simple to implement, per-connection state is brittle: SYN-flood attacks easily fill state memory, preventing muxes from keeping state for good connections.

We present Beamer, a datacenter load-balancer that is designed to ensure stateless mux operation. The key idea is to leverage the connection state already stored in backend servers to ensure that connections are never dropped under churn: when a server receives a mid-connection packet for which it doesn't have state, it forwards it to another server that should have state for the packet.

Stateless load balancing brings many benefits: our software implementation of Beamer is twice faster than Google's Maglev, the state of the art software load balancer, and can process 40Gbps of HTTP uplink traffic on 7 cores. Beamer is simple to deploy both in software and in hardware as our P4 implementation shows. Finally, Beamer allows arbitrary scale-out and scale-in events without dropping any connections.

## 1 Introduction

Load balancing is an indispensable tool in modern datacenters: Internet traffic must be evenly spread across the servers that deal with client requests, and even internal datacenter traffic between different services is load balanced to ensure independent scaling and management of the different services in the datacenter.

Existing load balancer solutions can load balance TCP and UDP traffic at datacenter scale at different price points [26, 13, 9, 22, 15, 31, 12, 18]. However, they all keep per-flow state: after a load balancer decides which server should handle a connection, that decision is "remembered" locally and used to handle future packets of the same connection. Keeping per-flow state should ensure that ongoing connections do not break when servers and muxes come or go, but has fundamental limits:

- Standard scaling events that include both muxes and servers break many ongoing connections.
- SYN flood attacks prevent muxes from keeping "good" connection state, negating its benefits.
- Running stateful load-balancers in software with many flows reduces throughput by 40% (§6.1).

In this paper we design, implement and test Beamer, a **stateless and scalable datacenter load balancer** that supports not only TCP, but also Multipath TCP [27]. The key idea behind Beamer is *daisy chaining* that uses the per-connection state already held by servers to forward occasional stray connections to their respective owners.

Our prototype implementation can forward 33 million minimum-sized packets per second on a ten core server, twice as fast as Maglev [9], the state of the art load balancer for TCP traffic. Our stateless design allows us to cheaply run Beamer in hardware too, as shown by our P4 implementation (§5). Beamer can scale almost arbitrarily because each load balancer acts completely independently and holds no per-connection state. Our experiments show that Beamer is not only fast, but also extremely robust to mux and server addition, removal or failures as well as heavy SYN flood attacks.

## 2 Background

Services in datacenters are assigned public IP addresses called VIPs (virtual IP). For each VIP, the administrator configures a list of private addresses called DIPs (direct IPs) of the destination servers. The job of the load balancer is to load balance connections destined to the VIPs across all the DIPs. Hardware load balancing appliances have long been around and are still in use in many locations; however they are difficult to upgrade or modify and rather expensive. Traditional app-level proxies such as HAProxy or Squid that terminate the client's TCP connection and open a new one to the server are also not desirable because their performance is quite low.

A raft of load balancers based on commodity hardware have been proposed that seek to address the shortcomings of existing solutions [26, 9, 13, 12, 18, 22, 15, 31]. Their goal is to process packets as ***cheaply*** as possible, while ***balancing load evenly*** across a dynamically changing population of backend servers and ensuring ***connection affinity***: all packets of a connection should reach the same server.

Almost all existing load balancers follow the same architecture introduced by Ananta [26] and we provide a brief description in Figure 1. In Ananta, load balancing is performed using a combination of routing (Equal Cost Multipath) and software muxes running on commodity x86 boxes. All muxes speak BGP to the border datacenter router and announce the VIPs they are in charge of as accessible in one hop. The border router then uses equal-cost multipath routing (ECMP) to split the traffic equally

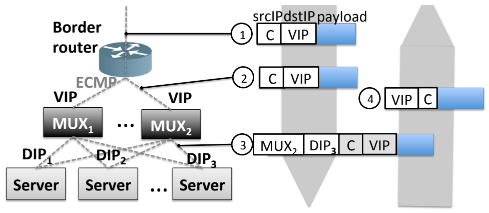


Figure 1: Load balancing: traffic to the VIP address is load-balanced across a pool of servers, each with a DIP address. Return traffic bypasses the muxes.

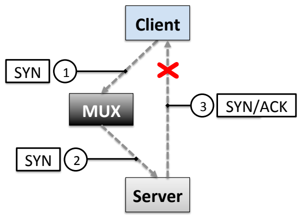


Figure 2: Mux and server disagree over the status of a connection.

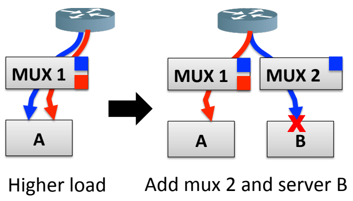


Figure 3: Scale out: stateful load balancers break TCP connections.

to these muxes. When a connection starts (i.e. a SYN packet arrives at a mux), a hash function is applied on the five-tuple and a server is chosen based on this hash. If a single server is added to the DIP pool, the assignment of some existing connections to servers will change; at the very least, the new server must receive an equal fraction of all ongoing connections. That is why, once a mapping of connection to DIP is chosen, it is stored locally by the mux to ensure all future packets will go to the same DIP.

Upon leaving the mux, the original packet is encapsulated and sent to the DIP. The receiving server first decapsulates the packet, changes the destination address from VIP to DIP, and then processes it in the regular TCP stack. When the reply packet is generated, the source address is changed from DIP to VIP and the packet is sent directly to the client, bypassing the mux to reduce its load (this is Direct Source Return, or DSR).

## 3 Limits of stateful load balancing

A key design decision of all existing load balancers is to keep a small amount of per-connection state to ensure *connection affinity*: once a connection is assigned to a backend, the mux will remember this decision until the connection finishes or a timer fires.

While per-connection state works well in the average case, it has a number of fundamental limitations which reduce its effectiveness in practice. First, because the mux only sees one direction of traffic, state kept by the mux can differ from the server's state for the same connection; the worst case here is the muxes' inability to cope with SYN flood attacks. Secondly, even in the absence of SYN floods, connections will be broken in scale-out or scale-in (or failure) events where both the mux set and the DIPs change simultaneously; such events happen naturally. Finally, software muxes' forwarding performance decreases with many active connections (see §6). We discuss these issues next.

**State mismatch between mux and server.** Consider the simple example in Figure 2: a client starts a TCP connection by sending a SYN packet, which is seen by a mux and then redirected to a server, and the mux saves the chosen mapping locally. The server replies with a SYN/ACK packet which never reaches its destination because the client is now disconnected. The server will send this packet a few times until it terminates the connection; the mux however is not aware of the reverse path unreachability and will maintain the state for minutes.

SYN flood attacks, where attackers send many SYN packets with spoofed IP source addresses [8], cause similar problems. During a SYN flood, the SYN/ACKs sent by the server never reach their destination, but both the server and the mux install connection state. SYN-cookies [8] are the standard protection against SYN flood attacks: when the number of half-open connections reaches a threshold, the server stops keeping state upon receiving a SYN, encoding the state in the SYN/ACK packet it sends to the client. When legitimate customers reply with the third ACK to finalize the connection handshake, the server uses the information from the ACK number (a reflection of its initial sequence number) and the timestamp (the echo reply field) in conjunction with local information to check if this is a valid connection; if so, it creates an established connection directly.

SYN cookies help the server shed unwanted state, but have no positive effect at the mux: the mux is forced to allocate state for every SYN it sees. Under a SYN flood attack, the servers will function normally but the muxes' connection memory will be overloaded to the point where they will behave as if they have no connection state, and thus DIP churn will break connections.

Ensuring state synchronization and defending against SYN floods at muxes is far from trivial: at the very least it requires muxes to keep more state (i.e. is the server sending SYN cookies or not?) and examine both directions of traffic; another cleaner solution is for the mux to terminate TCP. All solutions limit scalability.

**Connection failures during scaling events.** Even without SYN floods, keeping mux state does not guarantee connection affinity. Figure 3 shows a datacenter service that is running with one mux and one server. As load increases, one more mux and server are added. The border router now routes half the connections it sent to mux 1 to mux 2, as the blue flow in our example. Mux 2 does not have state for the blue flow, and it simply hashes it, assigns it to `B` and remembers the mapping for future packets. `B` receives packets from an unknown connection so it resets it. Such failures happen even when the border router uses resilient hashing and when all muxes use the same hash function. The necessary condition, though, is that both the mux set and the server set changes in quick succession, but such sequences of events occur naturally in datacenters during scale out and scale in events.

## 4 Beamer: stateless load-balancing

Using per-flow state at muxes fails to provide connection affinity in many cases. Can we do better without keeping flow state in the muxes? This is our goal here.

To achieve it, we leverage the per-flow state servers already maintain for their active connections. As an example, consider server `B` in Fig. 3: it receives a packet belonging to the blue connection, for which it does not have an entry in the open connections table; the default behaviour is to reset the blue connection. If `B` knew that server `A` might have state for this connection, it could simply forward all packets it doesn't have state for, including the blue connection, to `A`, where they could be processed normally. We call such forwarding between servers *daisy chaining* and it is the core of Beamer.

The architecture of Beamer mirrors that in Figure 1: our muxes run BGP (Quagga) and announce the same VIP to border routers. ECMP at the routers spreads packets across the muxes, which direct traffic to servers; finally the servers respond directly to clients. To build a scalable distributed system around daisy chaining, Beamer uses three key ingredients:

- *Stable hashing* (§4.1), a novel hashing algorithm that reduces the amount of churn in DIP pool changes to the bare minimum, while ensuring near-perfect load balancing and ease of deployment.
- A *fault-tolerant control plane*(§4.5) that scalably disseminates data plane configurations to all muxes.
- An in-band signaling mechanism that gives servers enough information for daisy chaining, without requiring synchronization (§4.2).

### 4.1 Stable hashing

Beamer muxes hash packets independently to decide the server that should process them. A good hashing algorithm must satisfy the following properties: it should load balance traffic well, it should ensure connection affinity under DIPs churn, and it should be fast.

A strawman hashing algorithm is to chose the target server by computing `hash(5tuple)%N`, where `N` is the number of DIPs; this is what routers use for ECMP. This mechanism spreads load fairly evenly and as long as the set of DIPs doesn't change, and mux failures or additions do not impact the flow-to-DIP allocations. Unfortunately, when a single server fails (or is added), most connections will break because the modulus `N` changes.

Consistent hashing [19], rendezvous hashing [30] and Maglev hashing [9] all offer both good load balancing and minimize or at least reduce disruption under churn. On the downside, in all these algorithms each server is in charge of *many* discontiguous parts of the hash space; this means the mux must match five-tuple hashes against many rules, reducing performance (for software deployments) or increasing hardware cost (for hardware ones). These algorithms target wide-area distributed systems and thus strive to reduce (mux) coordination. In datacenters, however, we can easily add lightweight coordination which enables a simple and near-optimal hash algorithm.

Beamer implements *stable hashing*, an extensible hashing approach that can be used to implement all the algorithms above. *Stable hashing* adds a level of indirection: connections are hashed against a fixed number of buckets, and each bucket can be mapped by the operator to any server. Before the load balancing service starts for a certain VIP, the operator chooses a fixed number of buckets `B` that is strictly larger than `N`, the maximum number of DIPs that will serve that VIP (e.g. `B=100N`). Each bucket is assigned to a single server at any time, and each server may have multiple buckets assigned to it. The number of buckets `B` and the bucket to server assignments are known by all muxes, and they are disseminated via a separate control plane mechanism (see §4.5). When a packet arrives, muxes hash it to a bucket by computing `b=hash(5tuple)%B`, and then forward the packet to the server currently assigned bucket *b*. As `B` is constant by construction, server churn does not affect the hashing result: a connection always hashes to the same bucket, regardless of the number of active DIPs.

Bucket-to-server mappings are changed on server failure or explicitly by the administrator for load-balancing and maintenance purposes. These mappings are stored in reliable distributed storage (Apache ZooKeeper [16] in our implementation); muxes retrieve the latest version before they start handling traffic. As changes to the bucket-to-DIP mappings are rare, this mechanism has low communication overhead and scales to datacenter-sizes (§6.4).

We show an example of stable hashing in Figure 4. The administrator has configured four buckets; muxes first hash flows into these buckets to find the destination

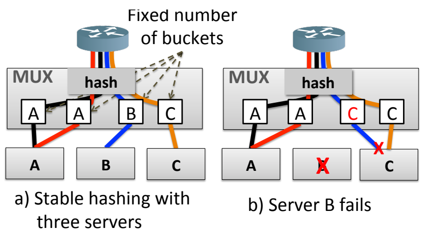


Figure 4: Stable hashing is resilient to server failures.

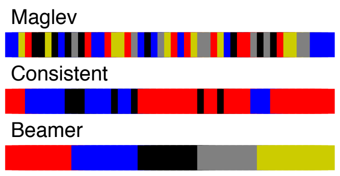
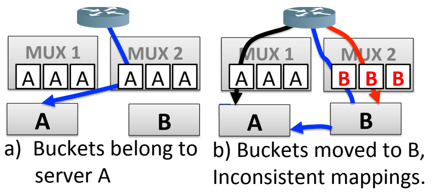


Figure 5: Hashing algorithms comparison


MUX 1
A A A
MUX 2
A A A
A
B
a) Buckets belong to server A
MUX 1
A A A
MUX 2
B B B
A
B
b) Buckets moved to B, Inconsistent mappings.
Figure 6: Daisy chaining allows server addition or removal without disrupting ongoing connections.

server. When server B fails, the controller will move the third bucket from B to C; the mapping is then stored in ZooKeeper and disseminated to all muxes. After the mapping is updated, flows initially going to A or C are unaffected, and flows destined for B are now sent to C. Only the blue connection, handled by B, is affected.

Our bucket-to-server mappings are managed centrally by the controller. The controller has the freedom to implement any bucket-to-server mapping strategy to mimick consistent hashing, rendezvous hashing or Maglev. In our implementation we chose a greedy assignment algorithm that aims to maximize the contiguous bucket ranges assigned to muxes; this is very useful especially when Beamer is deployed in hardware, because it can use fewer TCAM rules to implement its dataplane functionality. To provide intuition about why this is the case, in Fig. 5 we show how 47 buckets are assigned to 5 servers using the three algorithms, where each server is shown in a different colour: the bigger the fragmentation, the higher the cost to match packets against packets in the dataplane. Beamer is the least fragmented, followed by Consistent and Maglev. When servers come and go, bucket assignments to servers will also become fragmented even with Beamer; Beamer runs a periodic defragmentation process to avoid this issue (see §4.5).

Our stateless design ensures that mux churn has no impact on connections: as soon as BGP reconverges to the new configuration, load will be spread equally across all muxes and no connections will be broken.

## 4.2 Daisy chaining

There is a natural amount of churn of servers behind a VIP, be it for load balancing purposes or for planned maintenance. To implement such a handover, all our controller has to do is to map to the new server the buckets belonging to the old server and store the new mappings in ZooKeeper. The muxes will then learn the new mapping and start sending the bucket traffic to the new server. For a truly smooth migration, however, there are two complications that need to be taken in account: existing connections will be broken and there might be inconsistencies when some muxes use the new mappings while others are using the old ones.

To solve both issues we use *daisy chaining*, a transitory period where both the new server and the old one are active and servicing flows that hit the migrated bucket(s). We aim to move all new connections to the new server, and process ongoing connections by forwarding them to the old server even if they arrive at the new server.

We give an example in Fig. 6 where we migrate three buckets between servers A and B. Initially, both muxes have the same mappings for all buckets. Daisy chaining starts when the controller migrates the buckets from A to B by storing the new mapping in ZooKeeper and marking A as the previous DIP, along with the timestamp of the update. Note that the muxes save the previous DIP for each bucket, as well as the time the reallocation took place. To see how daisy chaining comes into play, let us consider the way packets are processed upon reception by the server. If the incoming packet is a SYN (TCP or MPTCP), a valid SYN-cookie ACK, or it belongs to a local connection, then we can process it locally. Otherwise, the packet could belong to a connection that has been previously established on another server. In this case, we want to daisy-chain packets back to the appropriate server, but only for a limited time.

To enable this, packets destined to ports lower than 1024 (higher numbers are used for MPTCP load balancing, see §4.4) also carry the previous DIP and the timestamp of the change (or 0 when there is none). We always save locally the highest timestamp seen for the bucket the packet is hashed to, and enable daisy chaining to the previous DIP when current time is smaller then the timestamp plus the daisy chaining interval. Thus, packets are redirected to the appropriate destination as long as daisy chaining is active. Otherwise, they are dropped and a RST is sent back to the source.

Daisy chaining adds robustness to our whole design. Consider what happens if the two muxes in Fig. 6 temporarily disagree on the server now in charge of the three buckets. Flows that hit mux 1 are load balanced according to the old mapping and will be directed to A, who will simply process them locally (the black connection). Meanwhile, B will locally service the red connection, but will daisy chain the blue connection to A (the previous DIP) because it doesn't have state for it. When mux 1 finally updates its state, the black connection will be sent to B, and daisy chained back to A (assuming the rule is still active). After all the muxes have updated their state, A will only receive packets related to ongoing connections, which will quickly drop in number. While in principle daisy chaining can be left running forever, we try to avoid migrating buckets that are being daisy chained. That is why our Linux kernel implementation uses a hard timeout of four minutes for daisy chaining.

There is one subtle corner case where daisy chaining still doesn't protect against broken connections, and we exemplify in figure 6. Consider the red connection that is being serviced by B after mux 2 updates its configuration; if this connection is sent to mux 1 (e.g. via ECMP churn in BGP), mux 1 will send it to server A which will reset it. To avoid this problem, packets also carry the generation number for the dataplane information, and all servers remember the highest generation number they have seen. In this example, server A will receive packets from B (and possibly from other muxes) with generation 2 and will remember 2 as the latest generation. When A receives a mid-connection packet that can not be daisy chained and for which it has no state, it will check if the generation number from the mux equals the highest generation number seen; if yes, the connection will be reset. If not, the server silently discards the packet. This will force the client to retransmit the packet, and in the meantime the stale mux mappings will be updated to the latest generation, solving the issue. Note that, if the border router uses resilient hashing, the mechanism above becomes nearly superfluous.

## 4.3 Mux data plane algorithm

The mux data plane algorithm pseudocode is shown in Fig.7. Lines 3-9 handle regular TCP traffic: first the bucket b is found together with the current and previous DIPs for bucket b. After that, the packet is encapsulated and sent to the current DIP. The algorithm is very simple, requiring a hash and one memory lookup in the buckets matrix (all three columns easily fit in one cache line). The remaining code in the mux performs Multipath TCP (MPTCP) [27] traffic load balancing equally cheaply: a single lookup is needed and the packet is encapsulated and sent to the appropriate DIP (see §4.4).

The simplicity of the mux is key to good performance: on one core our prototype can handle around 5-6Mpps, and around 33Mpps on an ten core Xeon box.

```
 1 packet* mux(packet* p){
 2   if (p->dst_port<1024){
 3     gen = buckets.version
 4     b = hash(5-tuple)%B;
 5     dip = buckets[b][0];
 6     pdip = buckets[b][1];
 7     ts = buckets[b][2];
 8
 9     return encapsulate(mux,dip,pdip,ts,gen,p);
10   }
11   else {
12     dip = id[p->dst_port];
13     return encapsulate(mux,dip,p);
14   }
15 }
```

Figure 7: Mux data plane pseudocode

## 4.4 Handling Multipath TCP

MPTCP deployment on mobiles is spreading: all IOS-based phones have it, as do top-end Android devices such as Galaxy S7 / S8. None of the existing datacenter load balancers support MPTCP, unfortunately, and this is a barrier to server-side deployment. This is because load balancing MPTCP is more difficult than regular TCP.

An MPTCP connection contains one or more subflows, and it starts when its first subflow is created. Each subflow looks very much like an independent TCP connection to the network, with the exception that its segments carry MPTCP-specific options. When load balancing MPTCP, all subflows of the same connection must be sent to the same DIP. Existing datacenter load balancers (e.g. Ananta [26], Maglev [9], SilkRoad [22], Duet [13]) treat MPTCP subflows as independent TCP connections, thus the DIP for each subflow will be decided independently, sending them to different servers most times, and breaking secondary subflows.

In MPTCP, after the initial subflow is setup, each endpoint computes the token—a unique identifier its peer has assigned to this connection. This token is embedded in the handshake of additional subflows within the same MPTCP connection and helps the remote end find the appropriate connection to bind the subflow to.

If one follows the mux state design approach, implementing MPTCP support requires storing the server-token $T_B$ to DIP mapping in some shared memory all muxes can access, but this poses two problems: first, since only the DIP knows the token, it should update the shared memory when a new connection is created; secondly, having a shared memory access for each additional subflow would be prohibitive from a performance point of view.

We propose a stateless solution that leverages the mobility support available in MPTCP to ensure that secondary subflows can be forwarded to the correct server. We use the destination port in SYN JOIN packets to encode the server identifier. This is shown in Fig. 8: when

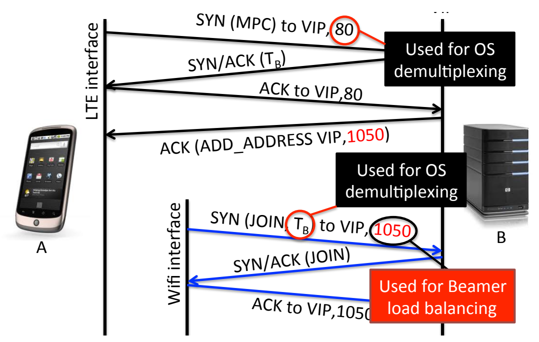

LTE interface
SYN (MPC) to VIP,80
Used for OS demultiplexing
SYN/ACK ($T_B$)
ACK to VIP,80
ACK (ADD_ADDRESS VIP,1050)
Used for OS demultiplexing
Wifi interface
SYN (JOIN $T_B$) to VIP,1050
SYN/ACK (JOIN)
Used for Beamer load balancing
ACK to VIP,1050
A
B

Figure 8: Load balancing MPTCP statelessly. Beamer uses address advertisement to embed the server identifier in the destination port of secondary subflows.

receiving TCP SYN or MPTCP initial subflow SYN packets, the port number is used to find the listening socket. However, SYN JOIN packets (handshake of secondary subflow) contain a token ($T_B$) that servers use to find the existing MPTCP connection [11]; the destination port is not used by the stack and we use it for Beamer.

Before deployment, Beamer assigns each server a unique identifier in the 1025-65535 range. We reserve port numbers (1-1024) for actual services, and utilize the remaining port numbers to encode server identifiers for secondary subflows. MPTCP allows endpoints to send `add address` options that specify another address/port combination of the endpoint to be used in future subflows. We use this functionality as shown in Fig. 8: whenever a new MPTCP connection is established (i.e. the third ACK of the first subflow is received), servers send an ACK with `add address` option to the client with the VIP address and the server identifier as port number. The client remembers this new address/port combination and will send subsequent subflows to it.

To handle MPTCP secondary subflows correctly, our mux (Fig. 7) treats traffic differently depending on the packet's destination port: traffic to ports greater than 1024 are treated as secondary subflows and directed to the appropriate servers. As each server has exactly one port associated to it, our solution can support at most 64K servers for each VIP. The muxes use another indirection table called `id`, that simply maps port numbers to DIP addresses (identified with $D_i$ here, see Fig. 9).

Note that we only need daisy chaining to redirect initial subflows of MPTCP connections or plain TCP connections. Secondary subflows are sent directly to the appropriate server (uniquely identified by the port).

## 4.5 Beamer control plane

We have designed our control plane to be scalable and reliable and built it on top of ZooKeeper. ZooKeeper ensures reliability by maintaining multiple copies of the data and using a version of two-phase commit to keep the copies in sync. Users can create hierarchies of nodes, where each node has a unique name and can have data associated to it, as well as a number of child nodes.

We show the operation of our control plane by detailing how the most important operations are implemented. The controller is the only machine that writes information into ZooKeeper and muxes only read ZooKeeper information. Servers do not interact with ZooKeeper at all. The node hierarchy used by Beamer is shown in Fig. 11.

When a new Beamer instance is created, the controller creates a high level node (called in this example "beamer") and a "config" child node holding the basic configuration information including the VIP and the total number of buckets. Next, the operator can add DIPs to the load balancer instance by specifying the DIP address, an identifier (unique within an instance) and a weight.

The bucket-to-server assignments are stored in the "mux_ring", while the "dips" node contains DIP-related metadata, which is not read by the muxes.

**Creating a DIP.** To add a DIP, the controller will add an entry for the DIP in the "dips" node, and then in the "id" node. ZooKeeper guarantees that all individual operations are atomic. If the controller or its connection to ZooKeeper crashes at any point, it checks the "dips" node for any in-progress DIP additions. If a DIP is not represented in the "id" node, it is added there as well.

**Load balancing** is run after one or more DIPs are added, before they are removed or after their weight is changed. The assignment algorithm runs in a loop, aiming to balance load properly while reducing daisy chaining:

- Select the most overloaded server `A` and underloaded server `B`, where load is the ratio between assigned buckets and weight.
- Find the maximum number of buckets `n` such that, if transferred from `A` to `B`, `A`'s load would not fall under the average, and `B`'s load would not rise above.
- Select the `n` buckets that have been in `A`'s pool for the longest time, and move them from `A` to `B`.

To move buckets between two servers, the controller simply updates the mux ring (see below). Our greedy bucket-to-DIP assignment algorithm will cause fragmentation when the DIP set is altered and buckets are reassigned to ensure good balancing. Beamer includes a defragmentation algorithm (see Appendix) that runs when fragmentation exceeds a threshold.

**Removing a DIP** begins by setting its weight to zero. After running the load balancing algorithm, the "dips" entry is removed.

**Updating mux dataplane configuration safely.** The muxes load the dataplane configuration from the

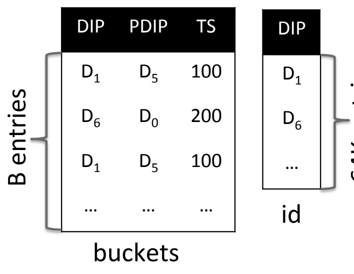


Figure 9: Mux data structures

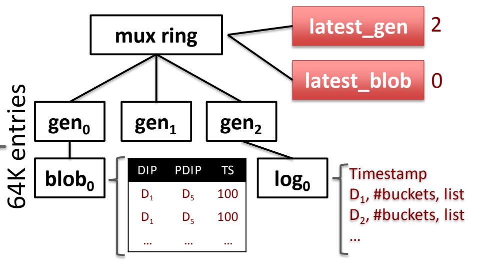


Figure 10: Mux configuration information stored in ZooKeeper

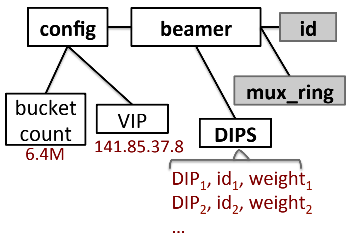


Figure 11: Controller information stored in ZooKeeper

"mux_ring" node in ZooKeeper, as shown in Fig. 10. The dataplane information is stored in several generation nodes, that have *logs*, which capture incremental updates to bucket ownership. Optionally, a generation node may also have a *blob*, which is an entire snapshot of the dataplane. The logs and blobs are compressed using zlib [1], and may span multiple nodes[1].

The blob contains the same data structure used by the muxes to forward packets. When it starts up, the mux first reads the values of "latest_blob" (the newest generation that contains a blob, in this case "gen0"). The mux reads the blob from "gen0" and obtains a functional forwarding table. If the "latest_gen" node has a value greater than the latest blob, the mux reads all the generations in ascending generation number order and applies the deltas contained therein. The mux now has an up-to-date forwarding table and can process packets.

ZooKeeper allows clients to register watches on nodes and it delivers notifications when the nodes' data is updated. We leverage this functionality to inform muxes that the forwarding information has changed: all muxes register watches for the "latest_gen" node; when it changes, the muxes will fetch and apply the new deltas.

Finally, the controller updates the mux ring information with the following algorithm: a) create a new generation node and store the updates to the bucket-to-server assignments, and b) update the "latest_gen" node to inform the muxes of the new version. The controller also creates blobs by applying the deltas in the same way the muxes do, creating the blob nodes under the current generation and then updating the "latest_blob" entry.

**Safety.** The controller algorithm above does not require any synchronization between muxes or the controller beyond ZooKeeper interactions. To ensure correctness, it maintains the following invariants: a) Muxes only read ZooKeeper information; they never update it. Configuration information is only written by the fault-tolerant controller; b) State updates are atomic from the muxes' point of view: they occur when the "latest_gen" node is changed

[1]ZooKeeper nodes have a maximum size of 1MB.

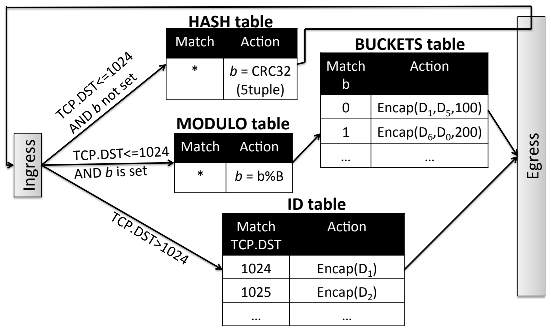


Figure 12: P4 implementation of a Beamer mux.

(an atomic ZooKeeper operation), which only occurs after the controller has finished writing the data pertaining to the newest generation; and c) Generations with an identifier smaller than "latest_blob" can be safely deleted by the controller since muxes do not need them to have an up-to-date version of the dataplane.

## 5 Implementation

Beamer servers run a kernel module (1300LOC) that handles decapsulation, address mangling and daisy-chaining. We have also patched the MPTCP Linux kernel implementation (version 0.90) to advertise the server ID for subsequent subflows (a few tens of lines of code). Our controller is implemented in 2100 lines of Java.

We have implemented Beamer muxes both in software and hardware (P4). The software mux runs a Click configuration atop the FastClick suite [4]. FastClick enables scaling to multiple cores, sets thread to core affinities and directly assigns NIC queue interrupts to cores.

**Software mux.** The core of the software mux is a Click element we have developed that implements our mux algorithm and acts as a ZooKeeper client to receive state updates. To improve performance, our design is completely lock free, which we achieve by carefully ordering the way we update the buckets matrix during updates.

**Our hardware mux implementation** is based on P4 [5] and is shown in Fig. 12. It contains two match-action

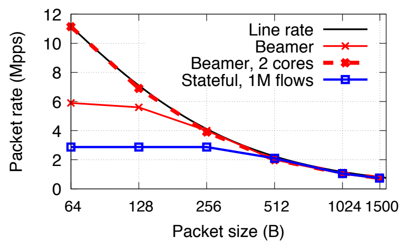


Figure 13: Forwarding performance vs. packet size (one core). Beamer outperforms stateful design 2-3x.

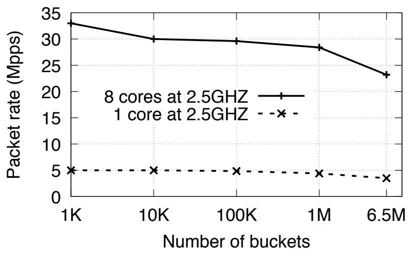


Figure 14: Forwarding performance (ten-core Xeon, 4 x 10Gbps). Beamer forwards 40Gbps with 128B packets.

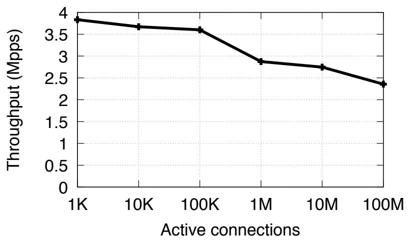


Figure 15: Software mux performance decreases with more active connections.

tables, one for the bucket-to-server mappings and one for the id-to-server mappings. The control part of the mux simply directs packets to one of these tables based on their destination port. The tables are populated by a software control plane that speaks to ZooKeeper.

The biggest challenge is computing the hash of the 5-tuple that is needed for lookup in the buckets table in the ingress stage of the pipeline: we can use the stock CRC32 function to compute it, but the checksum is calculated only in the egress stage. Since we cannot compute the CRC manually, we "recirculate" the packet instead: when the packet first enters the pipeline, its hash is calculated and stored as metadata "b" and the packet is resubmitted to the ingress port. To compute the modulus we add one more table with a single default entry where the modulus in computed in the action. Finally, the packet hits the buckets table and is encapsulated.

## 6 Evaluation

The purpose of our evaluation is to test the performance, correctness, fault tolerance and deployability of our prototype implementation. We used our local testbed containing 20 Xeon-class servers connected directly to an 48-port 10Gbps BGP router to test dataplane performance and perform microbenchmarks of our control plane. We ran experiments on Amazon EC2 to show that our controller can scale to a large Beamer instance with one hundred muxes, 64K DIPs and 6.4 million buckets. In the appendix we also evaluate stable hashing.

Our results show that Beamer is simultaneously fast and robust: no connections are ever dropped, in contrast to stateful approaches, Beamer's dataplane performance is twice that of the best existing software solution, and our mux introduces negligible latency when underloaded ($100\mu s$). The control plane experiments highlight the robustness and scalability of our design.

### 6.1 Micro-benchmarks

We first tested our software mux in isolation handling 1000 buckets. The server used for testing has a ten core Intel Xeon processor running at 2.7GHz, 16GB of RAM and a ten gigabit NIC using the 82599 Intel chipset. Our traffic generator is based on the *pkt-gen* utility from the netmap [29] suite. The generator can saturate a 10Gbps link with minimum sized packets. In each experiment we generate packets of a single size and we measure performance at the receiver using pkt-gen.

The source code of previous datacenter load balancers including Ananta and Maglev is not publicly available. To compare against such solutions, we implemented *Stateful*, a version of our mux that uses a hash table to store per flow load balancing decisions. We use *Stateful* to understand the performance of stateful load balancers.

The results are shown in Fig. 13. First, we note that the stateful design, running with 1 million active flows (a typical load seen in production [22]), is significantly slower than Beamer, because flow table lookups and insertions are comparatively expensive and result in cache-thrashing. Fig. 15 shows performance as a function of the number of active flows. Throughput drops from 3.9Mpps with one thousand active flows to 2.3Mpps with 100 million active flows. The performance results presented in the Maglev paper ([9], Fig. 8) are comparable to those of *Stateful*: 2.8Mpps per core and 12Mpps for six cores. Beamer forwards 6Mpps per core, twice faster.

To see how Beamer scales, we also increased the number of cores it uses to service the single NIC while spreading the NIC queues across the cores. With at least two cores, the Beamer software mux achieves line rate for all packet sizes. Note that the maximum throughput with 64B packets is lower than the expected 14.88Mpps because of the overhead of the encapsulation we use: our mux adds an IP-in-IP encapsulation header (20B) to all packets, and an IP option (16B) to packets to ports smaller than 1024.

Finally, we installed four ten gigabit NICs into a Xeon server with ten cores at 2.5GHZ per socket. The per-core forwarding performance on this machine is 10% slower than in the above experiments because the CPU is 10% slower. This setup allows us to test just how much traffic a software mux can handle if it uses all its resources.

We used four clients and four servers each with one 10Gbps NIC to saturate our MUX with 64B packets. We also varied the number of buckets to see how our design copes with larger server populations. Per core throughput with 64B packets drops to 5.6Mpps when the mux handles 100K buckets, and to 5.1Mpps when there are 1M buckets. The results are due to decreased cache locality when the memory needed to store the bucket information increases. A mux implementation could coalesce neighbouring buckets that point to the same server to reduce the number of effective buckets, thus increasing performance.

The total throughput per mux is shown in Fig.14: our mux can forward 23 to 33Mpps per server, or 20 to 30Gbps depending on the number of buckets. With 128B packets the mux saturates all interfaces (40Gbps).

**Performance with real traffic.** We used MAWI [21] traces to estimate the throughput of our mux in realistic traffic conditions, and to estimate how many web servers could be handled by a single mux. We built a replay tool that takes packet sizes from MAWI HTTP uplink traffic and generates such packets as quickly as possible.

We measured the performance of a mux with four 10Gbps NICs installed: our mux can forward 36Gbps of HTTP uplink traffic, saturating all links (considering our encapsulation overheads at the mux) while using 7 of the 10 cores of the machine.

In the MAWI traces, server-to-client traffic is 15 times larger than client to server traffic, so one mux can load balance a pool of servers that together serve 540Gbps of downlink traffic. HTTP servers running custom made stacks can serve static content at 60Gbps [20]; however most servers will serve much less than that because content is dynamic. We expect one server to source around 1-10Gbps of traffic, and expect that a single software mux could cater for 50-500 servers.

**Implementation overheads.** We measure the server overhead introduced by our kernel module that decapsulates packets and implements daisy chaining. To this end we ran a 10Gbps iperf connection between a client and a Beamer server and measured its CPU usage with and without our kernel module. The vanilla server has an average CPU utilization of 7%, and of 9% with our module installed; this overhead is negligible in practice.

**Latency.** Our software mux achieves high throughput, but have we sacrificed packet latency in our pursuit of speed? We setup an experiment where our mux is running on a single core and processing 64B packets sent at different rates. In parallel, we run a ping with high frequency between two idle machines. The echo request packet passes through the mux, and the reply is sent directly to the source. We show a CDF of ping latency measurements for different packet rates in Figure 16. As long as the CPU is not fully utilized, both median and worst-case packet latencies stay below 0.2ms. When we overload the mux with 6.6Mpps (600Kpps more than its achievable throughput), the ping latency jumps to 1.5ms and 14% of packets are dropped. This latency is a worst case and is explained by the time it takes one core to process all the packets stored in the 10 receive queues used by netmap (one queue per core, 256 packets per queue).

**P4 dataplane.** We do not have access to a Tofino switch yet, so we resort to both software deployment and NetFPGA deployment to test our P4 prototype.

We first ran our P4 mux in the behavioural model switch on one of our Linux machines and measured its performance: the switch can only sustain 55Mbps of iperf throughput with 1500B packets, and around 4.5Kpps with minimum-sized packets. Any performance measurements with this switch are therefore irrelevant; we do, however, use it to check the correctness of our implementation and interoperability with our controller and the Click-based software mux.

Our NetFPGA implementation of the P4 switch uses P4-NetFPGA [2]. To enable our prototype to compile we had to make a number of modifications. First, we upgraded our code to P4-16 which simplified our code because actions can compute checksums, so we don't need to recirculate packets anymore. Next, running on hardware imposes constraints on table actions, limiting the bitsize of action parameters. To avoid these problems we broke up bigger tables into cascading smaller tables which satisfy the constraints. The decomposition is done such that we maintain consistency even if concurrent tables are not modified simultaneously.

We tested our implementation with Vivado's xsim 2016.4 simulator, injecting a batch of packets, verifying they are processed correctly, and measuring the time needed by the switch. The simulator reports that it takes 154 $\mu$s to process 10000 packets with 100B packets; this means the P4 mux can handle around 60Mpps. Deploying this prototype on actual hardware is ongoing work.

## 6.2 Scalability and robustness

**Handling mux churn.** One of the major benefits of software load balancing is the ability to add capacity when demand increases. This means setting up a new mux and sending a BGP announcement for the VIPs it serves. As soon as the announcement propagates to the border routers, they start hashing traffic to the new mux.

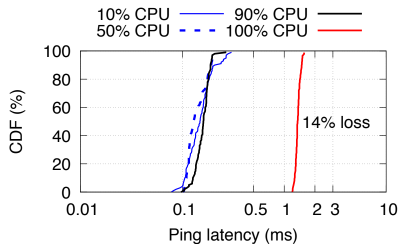


Figure 16: Mux latency is less than 0.3ms: when not fully loaded.

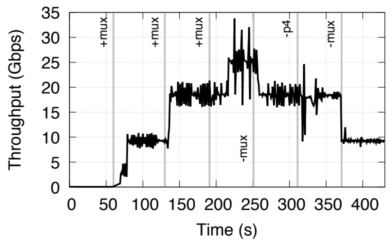


Figure 17: Beamer handles failures and mux churn smoothly.

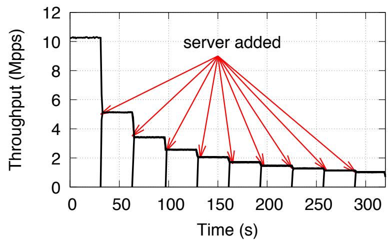


Figure 18: Beamer spreads traffic evenly across all active servers.

To emulate a real datacenter setup, we use an IBM 8264 RackSwitch as the border router and setup five clients and six servers, each with one 10Gbps interface. Clients open several `iperf` connections each to the VIP of the server, and we plot their added throughput in Figure 17. We begin the experiment with a P4 switch running alone and load balancing all traffic; the performance is terrible, just 55Mbps. Next, we add three software muxes a minute apart: the graph shows the few seconds it takes the muxes to setup a BGP session, announce the VIP, and to the router to install the route and start using it. Traffic scales organically as we add more muxes. Connections change muxes, yet they are not affected since our muxes are stateless. The P4 and Click muxes behave the same way, and are interchangeable.

We start simulating mux failures: first we kill a Click mux at 240s by bringing its network interface down, and throughput drops to 20Gbps. Next we kill the P4 mux: total throughput suffers until BGP discovers the failure and reconverges, then it recovers to 20Gbps. During the experiment not a single iperf connection is broken.

We conclude that Beamer handles mux churn smoothly, and that it is trivial to create a heterogeneous deployment with both software and hardware muxes.

**Handling server churn.** We want to see how Beamer balances server traffic when servers are added or removed. We generate 64B packets from a single machine and send them directly to one mux (using two cores). The experiments start with a single server receiving all traffic, and then we keep adding servers every 30 seconds using the controller's command-line interface. The controller moves buckets between servers to evenly balance the traffic across all servers by updating ZooKeeper data, as detailed in §4.5. When all the changes are ready, the controller "commits" the change by updating the current generation node, and the mux updates its state. Figure 18 shows the throughput received by each server as a function of time: changes are almost instantaneous, and traffic is evenly balanced across all active servers.

**Connection affinity.** We now use TCP clients to estimate the ability of Beamer and Stateful to provide connection affinity in different scenarios. In all experiments, we have 7 clients each open 100 persistent HTTP connections and continuously downloading 1MB files over each of them, in a loop, from servers in a Beamer cluster.

In our first experiment, we begin with two muxes and 8 servers and then perform a scale-down event: we first remove some servers, wait for 30s and then remove one mux. The number of broken connections is shown below.

| DIPs removed | 0 | 1 | 2 | 4 |
|---|---|---|---|---|
| Stateful | 0 | $54 \pm 7$ | $103 \pm 14$ | $214 \pm 48$ |
| Beamer | 0 | 0 | 0 | 0 |

As expected *Stateful* behaves poorly: it drops 7%-30% of the active connections when 1 to 4 servers are removed. After the DIP is removed, traffic is still sent correctly because there is state in both muxes. However, when one mux is removed, its state (for half of the connections hitting the removed DIP) is lost, and these will be hashed by the remaining mux to the other servers, which will reply with RST messages. In contrast, Beamer does not drop any connection because daisy chaining keeps forwarding packets to the removed DIP.

**SYN flood.** In our second experiment we use a similar setup, but only remove servers, keeping the mux set constant. In the absence of a SYN flood attack both Stateful and Beamer provide perfect connection affinity, as expected. We then started a SYN flood attack (1Mpps) running in parallel to our 7 clients; we show the number of dropped connections below:

| DIPs removed | 0 | 1 | 2 | 4 |
|---|---|---|---|---|
| Stateful | 0 | $87 \pm 2$ | $184 \pm 8$ | $351 \pm 21$ |
| Beamer | 0 | 0 | 0 | 0 |

Stateful performs rather poorly in this SYN flood attack, because the state its muxes keep for its real clients is flushed out by the aggressive attack. In contrast, Beamer's performance is not affected. Finally, we measured the effect of the SYN flood on the servers themselves, finding there was little impact: the average server utilization increased by 1% during the attack, and the flow completion times increased from 1ms to 1.3ms in the median and from 1.4ms to 1.7ms at the 99%.

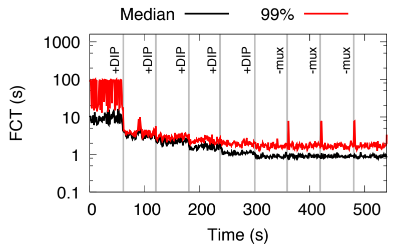


Figure 19: Flow completion times for MPTCP clients using Beamer

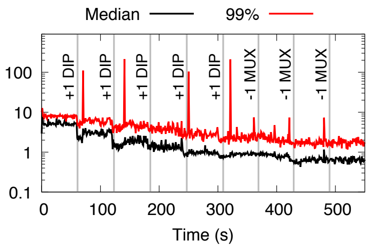


Figure 20: Flow completion times for MPTCP clients using Stateful.

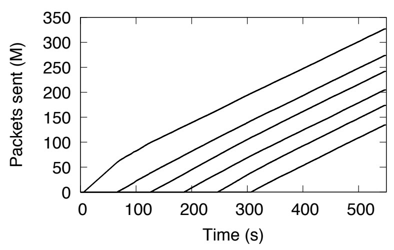


Figure 21: Outgoing traffic per server with Beamer

## 6.3 Load balancing HTTP over MPTCP

In our next experiment we emulate over-the-air mobile app updates. We setup four clients repeatedly downloading the same 100MB file using the `siege` tool from the VIP handled by Beamer. Each client opens 10 parallel downloads, and runs in a closed loop: as soon as one transfer finishes, it starts a new one. The clients run MPTCP and have two virtual interfaces on the same 10Gbps physical interface.

We start the experiment with four muxes and one server processing all the traffic. After every minute a new server is added until we reach six servers. Next, we start killing one mux every minute until there is a single mux running.

We plot the median and 99th percentile of client-measured flow completion as a function of time in figure 19. The graph shows that adding servers drastically reduces both the median flow completion time and especially the tail. The median drops from 10s when a single server is used, to 1s when six servers are used. The 99th percentile also drops from 50-100s with one server to a couple of seconds with six servers. Notice the ten second spikes in 99% FCT when muxes are killed: these are subflows that were handled by the failed mux, and they stall the entire MPTCP connection until they timeout and their packets are reinjected on good subflows.

Figure 21 shows the cumulative number of packets sent by each server and we can clearly see how initially only one server is active, another quickly follows and then more servers join one minute apart. The graph also shows that Beamer does a good job spreading connections equally across all active servers.

Finally, we wanted to check that daisy chaining works as described. We plot the number of daisy chained packets by each server in Figure 22. Note the different unit on the y axis (thousands of packets instead of millions): daisy chaining not only works, but it is also quite cheap. Daisy chaining forwards a total of 30 thousand packets, most of which are ACKs (total size is around 200KB).

Siege did not report any failed connections, but this could be masked by MPTCP's robustness to failures. We looked at individual subflow statistics and found that no subflows were reset; we did see numerous subflow timeouts triggered by mux failures, however these are well masked by MPTCP: when one subflow crossed a failed mux, its packets get resent on other working subflows.

We ran the same experiment with *Stateful* to test its behaviour when handling MPTCP connections and under mux failures. Packet traces show that, overall, less than 20% of MPTCP secondary subflows are created; this is expected, since *Stateful* is oblivious to MPTCP and randomly sends subflows to servers. With *Stateful*, MPTCP connections have a single subflow most times and behave like regular TCP. Without failures, FCTs should be similar to MPTCP/Beamer since the total achievable network capacity is the same. The FCT results in Figure 20 confirm our expectations: median FCTs are similar, however the 99th percentile is much higher. This is because MPTCP does a much better job of pooling the available network capacity than TCP does, thus reducing the outlier FCTs. Also note the huge spikes when DIPs are added or muxes are removed: this is Siege timing out on a connection after many retransmission attempts.

## 6.4 Controller scalability

Stable hashing works great as long as the centralized allocation of buckets to servers scales to large deployments. In this section we evaluate whether our controller can scale to many muxes and by extension to a large datacenter. To stress the controller we generate the maximum number of DIPs for a single VIP supported by our solution (64K), and create 100 buckets for each DIP, resulting in a total of 6.4 million buckets. We deploy our system in Amazon EC2 as follows: ZooKeeper runs on three VMs, one VM runs our controller, and one hundred VMs run our mux. According to our benchmarks in §6.1, one hundred muxes should easily be able to load balance traffic for 64K typical HTTP servers.

First, we want to see how long it takes to perform control plane operations on the maximum load balancer instance we can support. We have stress-tested the controller and run each operation multiple times with different numbers of pre-existing DIPs, randomized bucket-to-

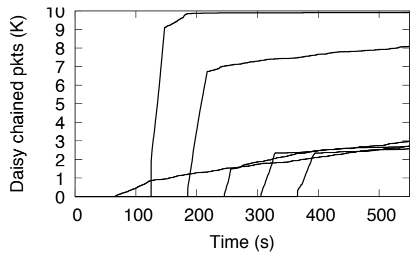


Figure 22: Packets daisy chained per server.

| # DIPs | Add (sec) | Rm (sec) |
|---|---|---|
| 1 | 0.63 | 0.58 |
| 10 | 0.57 | 0.57 |
| 100 | 0.69 | 0.67 |
| 640 | 0.87 | 1.58 |
| 6400 | 6.9 | 2.25 |
| 16000 | 8.1 | 3.2 |
| 32000 | 9.8 | 9.7 |

Figure 23: Duration of control plane ops on the largest Beamer instance

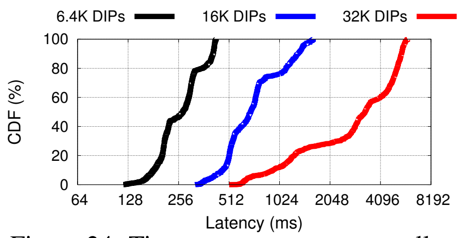


Figure 24: Time to propagate a controller update from ZooKeeper to all the muxes.

DIP assignments [2], and recorded the maximum completion time. The results are provided in table 23, showing that the Beamer controller performs large config changes in a few seconds.

Next, we measured the time it takes since the controller commits a new generation until all muxes download and start using it. Fig. 24 shows a CDF of the propagation time as measured at the muxes for three large config changes (adding 6.4K, 16K or 32K servers); the results show that even for 32K servers, all muxes use the new dataplane rules in a few seconds. Smaller updates are installed in tens to hundreds of milliseconds.

Finally, we note that all operations generated negligible amounts of control traffic; the largest ones (the addition or removal of 32K DIPs) incurred less than 1GB of traffic (10MB per mux).

## 7 Related work

Almost all existing solutions for datacenter load balancing keep per-flow state at muxes. Software solutions include Ananta [26], Maglev[9], IPVS[31] and GLB[15] while hardware ones include Duet [13] and SilkRoad [22]. Resilient hashing [6] takes a similar approach on switches and routers to avoid the pitfalls of ECMP. To allow scale in/out without affecting client traffic, stateful designs could use flow state migration, which is very expensive: OpenNF [14] or Split Merge [28] offer migration guarantees and strong consistency but at a steep performance cost (Kpps speeds).

A parallel effort to ours is Faild[3], a commercial stateless load balancer that works within a single L2 domain using ARP rewriting; this reduces its applicability to small clusters. Kablan et al.[17] propose to store per-flow state in a distributed key-value storage solution such as RAMCloud [25] instead of keeping it in memory; however its performance is limited to 4.6Mpps per box, eight times slower than Beamer.

[2]Updates to the dataplane are stored in a compressed format and having randomized bucket assignments yields near-worst-case compression ratios.

Load balancing within a single OpenFlow switch has been examined in [24, 32]. Orthogonal to existing load balancing solutions, Rubik[12] uses locality to reduce bandwidth overhead of load balancing while Niagara [18] offers an SDN-based solution to improve network-wide traffic splitting using few OpenFlow rules.

Paasch et. al [7] discuss the problems posed by MPTCP traffic to datacenter load balancers. Their analysis focuses on ensuring SYN(MPC) and SYN(JOIN) packets reach the same server, and it assumes muxes keep per flow state after the initial decision has been made. Duchene et. al [10] propose to load balance secondary MPTCP subflows by using IPv6 addresses; Beamer could easily implement this solution if both the client and the datacenter have IPv6 enabled and a working IPv6 path between them. Finally, Olteanu et al. propose [23] to load balance MPTCP traffic by encoding the server identifier in the TCP timestamp option; unfortunately this solution does not work if the client does not support or enable timestamps, and supports a smaller number of servers (8192) per VIP.

## 8 Conclusions

We have presented Beamer, the first stateless datacenter-scale load balancer solution that can handle both TCP and Multipath TCP traffic. Beamer muxes treat TCP and MPTCP traffic uniformly, allowing them to reach speeds of 6Mpps per core and 33Mpps per box, twice faster than the fastest existing TCP load balancer, Maglev [9]. A Beamer mux can saturate four 10Gbps NICs with real HTTP uplink traffic using just 7 cores.

Daisy chaining enables Beamer to provide connection affinity despite DIP and mux failure, removal and addition. In contrast to stateful designs, Beamer handles SYN flood attacks seamlessly.

Beamer is available as open-source software at `https://github.com/Beamer-LB`.

## Acknowledgements

This work was partly funded by the SSICLOPS and SUPERFLUIDITY H2020 projects.

## References

[1] zlib. `https://www.zlib.net/`.

[2] P4 to NetFPGA project. `https://github.com/NetFPGA/P4-NetFPGA-public/wiki`, February 2018.

[3] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the edge: Transport affinity without network state. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[4] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

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

[6] Brad Matthews and Puneet Agarwal. Resilient Hashing for Load Balancing of Traffic Flows. US Patent Application: US20130003549 A1, Jan 2013.

[7] Christoph Paasch, Christoph and Greenway, G. and Ford, Alan. Multipath TCP behind Layer-4 loadbalancers (internet draft). https://tools.ietf.org/html/draft-paasch-mptcp-loadbalancer-00, Sep 2015.

[8] W. M. Eddy. Defenses Against TCP SYN Flooding Attacks. *The Internet Protocol Journal*, 9(4), Dec 2006.

[9] 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)*, pages 523–535, Santa Clara, CA, Mar 2016. USENIX Association.

[10] Fabien Duchene and Vladimir Olteanu and Olivier Bonaventure and Costin Raiciu and Alan Ford. Multipath TCP Load Balancing. https://tools.ietf.org/html/draft-duchene-mptcp-load-balancing-01, July 2017.

[11] Ford, Alan and Raiciu, Costin and Handley, Mark and Bonaventure, Olivier. RFC6824: TCP Extensions for Multipath Operation with Multiple Addresses. `https://tools.ietf.org/html/rfc6824`.

[12] R. Gandhi, Y. C. Hu, C.-K. Koh, H. Liu, and M. Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *Usenix Annual Technical Conference*, 2015.

[13] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM*, 2014.

[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[15] GitHub Engineering. Introducing the GitHub Load Balancer. `https://githubengineering.com/introducing-glb/`, September 2016.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[17] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[18] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *CONEXT*, 2015.

[19] 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*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[20] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.

[21] MAWI Working Group Traffic Archive. `http://mawi.wide.ad.jp/mawi/`.

[22] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[23] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 20–25, New York, NY, USA, 2016. ACM.

[24] V. A. Olteanu, F. Huici, and C. Raiciu. Lost in network address translation: Lessons from scaling the world's simplest middlebox. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 19–24, New York, NY, USA, 2015. ACM.

[25] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[26] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[27] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *NSDI*, 2012.

[28] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[29] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.

[30] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.*, 6(1):1–14, Feb 1998.

[31] The Linux Foundation. The IP Virtual Server Netfilter module for kernel 2.6. `http://www.linuxvirtualserver.org/software/ipvs.html`, February 2011.

[32] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *HotICE*, 2011.

## Appendix

### A1. Defragmentation

There is a three-way trade-off between load balancing, fragmentation and churn. Beamer prioritizes load balancing, ensuring near-perfect balancing at all times. This means that we can either end up with fragmented bucket ranges assigned to servers (which will increase dataplane matching costs, especially for hardware dataplanes such as P4) or move buckets to reduce fragmentation but create daisy chaining traffic in the meantime.

Defragmentation is therefore necessary to ensure servers get a contiguous range of buckets, as this will reduce the number of rules needed in the mux dataplane. Beamer implements an algorithm that reduces fragmentation progressively, while keeping daisy-chaining costs small. The algorithm has two parameters: a target fragmentation rate ($fr \geq 1$, target average number of rules per DIP), and $bmax$, the maximum number of buckets that can be moved per server, per iteration.

The defragmentation algorithm has two phases: it first selects a *target mapping* and then it iteratively moves towards this target. The target mapping is computed whenever the DIP set changes, and this triggers a second implementation phase that includes a number of iterations where at most $bmax$ buckets per server are moved in each iteration; iterations are spread in time (one iteration every 4 minutes). The second phase stops whenever the target defragmentation rate is reached. While heuristic, the algorithm performs really well in practice.

The target mapping is computed greedily: starting at bucket offset 0, the controller selects the server which can take over a contiguous range while causing the least amount of churn. After every step, the offset is then incremented past the newly-allocated range.

```
start = 0
while (start < #buckets) {
  select DIP A s.t. churn of assigning A
         at position start is minimized.
  target(A) = {start, start+#buckets(A)}
  start += #buckets(A)
}
```

During each iteration of the second stage (pseudocode below), the controller performs a subset of the reallocations prescribed by the target mapping. It performs no

| Hashing algo. | Imbalance | Min. data plane rules | Server churn 1% | 5% |
|---|---|---|---|---|
| Consistent[19] | 2.27 | 9K | 0 | 0 |
| Maglev[9] | 1.01 | 65K | 2.3% | 3.3% |
| Stable Hashing | 1.01 | 1K | 0 | 0 |

Table 1: Hashing comparison, N=1000 and B=65537

more than *bmax* reallocations per DIP, while keeping the number of buckets constant for each DIP. (I.e. the number of buckets allocated to each DIP is the same as the number of buckets allocated away from it.)

```
let G = (V, E) be a directed multigraph, where
       vertices are DIPs and edges are bucket
       reassignments
for each DIP in V
  DIP.budget = bmax
while (G.has_cycles) {
  select cycle C
  for each realloc in C.edges {
    perform(realloc)
    E.remove(realloc)
  }
  for each DIP in C.vertices {
    DIP.budget -= 1
    if (DIP.budget == 0)
      V.remove(DIP)
  }
}
```

## A.2 Stable hashing evaluation

Table 1 shows the performance of Stable hashing against our implementations of the classical consistent hashing algorithm [19] and Maglev [9]. We used 1000 servers and 65537 buckets (the params are from the Maglev paper, §5.3, for fair comparison). We first measured the *imbalance*—the ratio between the maximum and average load—finding that Maglev and Stable have near-perfect load balancing, while Consistent hashing places twice more load on one unlucky server.

We next computed the minimum number of range rules we need to use in a hardware data plane to perform matching: algorithms that assign consecutive buckets to the same server will utilize fewer rules and are better. This is the case for Stable, where we only need one rule per server; Maglev, in comparison, needs as many rules as buckets, two orders of magnitude more than Stable. Consistent falls somewhere in the middle. Finally, we looked at the number of "innocent" connections that are disrupted when we remove a number of nodes; both Consistent and Stable have no collateral damage, while Maglev breaks 2.3%-3.3% of ongoing connections.

## A.3 Defragmentation evaluation

Stable hashing can enable very cheap hardware implementation with one rule per server, but this is only the case when all the buckets assigned to one DIP are contiguous. As servers come and go, even a perfect distribution can end up fragmenting the buckets, with each server in charge of many small ranges; this would require proportionally more rules to implement in hardware.

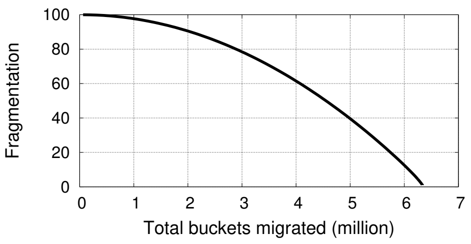


Figure 25: Defragmentating bucket-to-DIP assignments to reduce the number of data plane rules.

To avoid this effect, especially for hardware deployments, we can use the defragmentation described above: when the average number of rules per server increases beyond a given threshold, the defragmentation algorithm is invoked, reassigning buckets to remove fragmentation as described in §4.5.

We show a run of our defragmentation algorithm in Figure 25 starting from a worst case scenario where all buckets assigned to a DIP are scattered, and each DIP needs 100 rules to match its buckets. The figure shows how the fragmentation (number of rules needed per DIP) decreases as the algorithm migrates more buckets to reduce fragmentation.

The cost of defragmentation is daisy chaining, which is proportional to the number of buckets "moved" between servers. In the worst case when we move all buckets, Beamer will duplicate the incoming traffic for a brief period of time. To avoid creating congestion, the defragmentation algorithm moves slowly, migrating a few buckets at a time and then waiting for daisy chaining to end; this ensures that overall load increases only marginally.

We also note that defragmentation is only needed infrequently. We start with a fresh cluster containing 10K DIPs (1M buckets) and perform a number of control plane operations and show the resulting fragmentation in the table below. Even after large control plane operations, Fragmentation only increases slightly, and in some cases (like doubling or halving the cluster) it stays perfect. So in normal operation, we expect a cluster to slowly become more fragmented, reducing the need for defragmentation.

| # servers | Added | Removed |
|---|---|---|
| 0.1% | 1.01 | 1.01 |
| 10% | 2 | 2.1 |
| 33% | 1.5 | 2 |
| 50% | 1.33 | 1 |
| 100% | 1 | N/A |