Datacenter Scale Load Balancing for Multipath Transport

Vladimir Olteanu
University Politehnica of Bucharest

Costin Raiciu
University Politehnica of Bucharest

Abstract

Multipath TCP traffic is on the rise with recent deployments by Apple and Samsung on mobile phones. Despite this, MPTCP adoption on servers is falling behind and the major problem is that Multipath TCP does not work with existing datacenter load balancers.

In this work we present MPLB, a distributed load balancer for Multipath TCP traffic. MPLB uses stateless software multiplexers to direct traffic to backend servers and is resilient to mux and network failures, as well as backend server churn. We have implemented and tested MPLB, finding it can handle 6Mpps per machine with minimum-sized packets, seamlessly scale-out or in and gracefully handle mux failures.

1. INTRODUCTION

Load balancing is an indispensable tool in modern datacenters: Internet traffic must be evenly spread across the frontend 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.

Multipath TCP (MPTCP) is a recent extension to TCP [5] that allows endpoints to utilize multiple paths through the network in the same transport connection and is a drop in replacement for TCP. Multipath TCP adoption is gaining pace. Apple IOS (and OSX) implements MPTCP, Samsung and LG offer Android versions with Multipath TCP implementations and several Korean operators use MPTCP to "bond" LTE and 802.11ac to achieve gigabit speeds on mobile phones. Current deployments use a proxy to terminate MPTCP and talk TCP to the servers, but this is only a stop-gap solution.

HotMIddlebox, August 22-26 2016, Florianopolis , Brazil

 ISBN 978-1-4503-4424-1/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2940147.2940154

The next logical step is for MPTCP to be deployed in datacenter servers, but load balancer support is missing.

The job of a load balancer is to send the traffic from a single connection to the same server and is typically achieved by hashing on the packet five-tuple to decide the destination server for both TCP and UDP traffic. There is currently no scalable load balancer solution for Multipath TCP traffic: an MPTCP connection can have an arbitrary number of subflows, each of which looks like an independent TCP connection, and only SYN packets contain connection-identifying information. Using a hash of the five-tuple to load balance MPTCP subflows would result in different subflows arriving at different servers and breaking the protocol.

In this paper we design, implement and test MPLB, a scalable load balancer for Multipath TCP traffic. In our solution, each load balancer acts completely independently and holds no per-connection state. This ensures great scalability: we can add or remove load balancers very quickly and without any impact on existing connections, and can seamlessly tolerate failures. To support stateless load balancing we developed a few novel techniques, including: a) load balancers choose connection keys instead of servers, b) servers can redirect packets to other servers to ensure smooth handovers for scale-out and scale-in and c) each server encodes its identifier in the least-significant bits of the timestamp option.

We have implemented a prototype of MPLB and ran it on our local testbed. Our preliminary results show that each mux can handle around 6Mpps MPTCP SYN packets or around 26Gbps with Internet MTUs; MPLB can quickly scale up and down to track demand and is robust to failures.

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 of the destination servers called DIPs (direct IPs). The job of the load balancer is to distribute 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 ex-

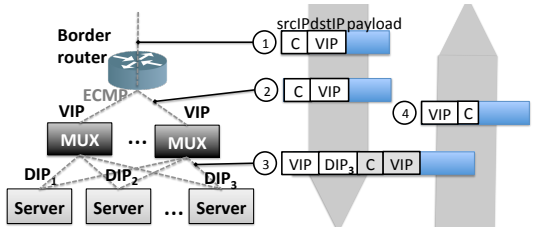

Figure 1: Load balancing: traffic to a VIP is spread across a pool of servers. Return traffic bypasses muxes.

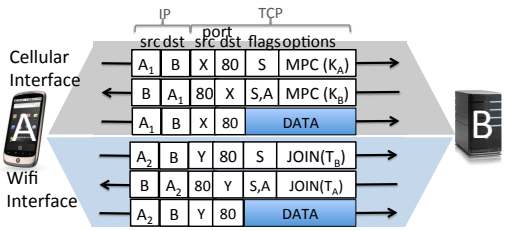

Figure 2: MPTCP Operation: only SYN packets identify the MPTCP connection.

pensive. That is why load balancers based on commodity hardware have been proposed [1–3, 6, 7, 9, 11]. There are two types of software load balancers: type 1, that terminate the client TCP (or MPTCP) connections and open new connections to the servers, and type 2, that do not terminate the connections.

Type 1 load balancers such as HAProxy [1] or Nginx [2] terminate TCP connections and could support MPTCP by a simple kernel upgrade; unfortunately they scale poorly since they must process both client-to-server and server-to-client traffic (ten times larger then client-to-server traffic) and can only handle a limited number of active connections (300K is an estimate for HAProxy).

Type 2 load balancers do not terminate TCP connections and can scale to large datacenters, and this is the focus of this work. At a high level, our solution uses the same architecture proposed by Ananta [11] and Maglev [3] and shown in Fig. 1. Load balancing is performed only for client-to-server traffic using a combination of routing 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 to these muxes. When a packet arrives at a mux, the mux decides by some mechanism which DIP it should be destined for.

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

Multipath TCP. Existing load balancer designs do not support Multipath TCP traffic. To understand the additional requirements posed by MPTCP, we provide a brief description of the relevant parts of the protocol.

Consider the example in Fig. 2. To start an MPTCP connection, mobile A could use its cellular interface to send to B a TCP SYN segment that has a multipath capable option (MPC). The option signals to B that A wishes to use MPTCP for this connection and also conveys K_A, A's key for this connection. To use MPTCP, B will reply with a SYN/ACK segment that also carries an MPC option with its own key, K_B. The keys are used by each endpoint to derive a token for the new connection; this token must be locally unique. The token is obtained by taking the 32 most significant bits of a SHA1 hash of the key, i.e. $T_A = MSB_{32}(SHA1(K_A))$.

After the initial subflow is setup, A (or B) can now add subflows to the existing connection. A could use its Wifi interface to send a SYN/JOIN message, which tells B that the new subflow is part of an existing connection. The JOIN option carries B's token, allowing B to associate the subflow to the correct connection.

When an MPTCP sender wishes to send data, it can use any of the available subflows. The sender will create a TCP segment with the appropriate subflow addresses and ports, together with an option that informs the remote end of the connection level sequence number of this data (not shown). No connection identifying information is carried in data packets.

When load balancing MPTCP traffic, all subflows of the same connection must be sent to the same DIP. Existing load balancers such as Ananta will treat MPTCP subflows as independent TCP connections: a hash on the five-tuple will decide the DIP for each subflow. In most cases the different subflows will be sent to different servers, breaking all subflows except the first one.

3. OVERVIEW OF MPLB

Ideally, muxes should not keep per flow state: the fate of each packet should be decided independently, without any mux-specific state. If this were the case, any mux could load balance any packet, and mux failures would have no adverse effects on ongoing connections. The changes needed to ensure fault-tolerance underpin our whole design, so we present them first.

The statring point of our solution is very simple: instead of hashing the SYN and then remembering the decision locally (as in Ananta or Maglev), muxes apply a hash function to each packet and choose the target server by computing $hash(5-tuple)\%N$, where N is the number of DIPs. As long as the set of DIPs doesn't change, the load balancing decision will be the same regardless of the mux, 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 used changes.

Stable hashing. To avoid this issue, we add a level of indirection by using "buckets" as follows. First, we choose a number of buckets B that is strictly larger than

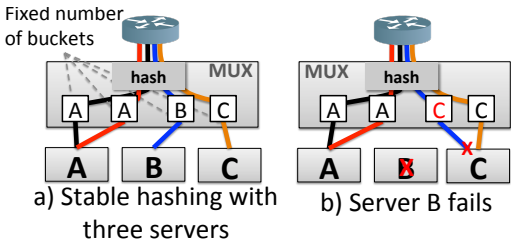

Figure 3: Stable hashing is resilient to server B failing: only B's flows are moved.

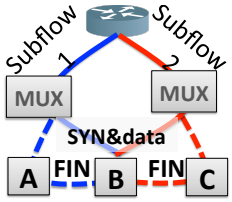

Figure 4: Load balancing MPTCP statelessly

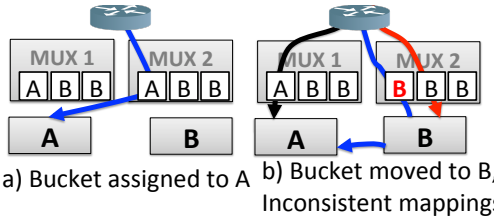

Figure 5: Daisy chaining enables scaling without affecting ongoing flows.

N, the number of servers. Each bucket is "assigned" to a single server, and one server may have multiple buckets assigned. The number of buckets and the buckets to server assignments are known by all muxes, and they are disseminated via a separate mechanism (see below). When a packet arrives, muxes hash the connection to a bucket by computing $b = hash(5-tuple)\%B$, and then forward the packet to the server currently assigned bucket b. As the set of buckets is constant, server churn does not affect the hashing result: if a server fails, only the flows it hosted will be affected.

A centralized controller manages the mappings between buckets and servers. These mappings change infrequently: on server failure or explicitly by the administrator for load-balancing and maintenance purposes. A discussion of how the mappings are managed is beyond the scope of this paper.

We show an example of stable hashing in Figure 3. When server B fails, the controller will map the bucket to server C; the mapping is then 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 connections that were already open on B before the failure are broken.

3.1 Load balancing Multipath TCP

MPTCP load balancing is hard because there is no discernible relationship between the initial connection SYN sent by the client and the following subflow SYN packets, as shown in Fig. 2: the initial SYN contains A's key, and the second subflow SYN contains B's token.

There are two possible types of solutions to solve the problem. When a new connection is setup, B could inform the associated mux of its token (either directly, or by using a distributed key-value store). Unfortunately, such solutions either force the mux to store per-flow state, reducing fault tolerance, or add latency to every packet because of distributed state lookups.

A better solution is for muxes to load balance JOIN packets using the token directly: treat the token as a hash and select a DIP by performing a lookup in the buckets array (i.e. $token\%B$). For this solution to work correctly, all connections destined to server B must have a token (chosen at connection startup) that points to B. This is not the case for MPTCP: tokens used by B depend on the randomly generated per-connection key.

A brute force solution is for B to randomly generate multiple keys instead of just one per connection and select the first key that generates an appropriate token. The time complexity of this method depends on the number of DIPs and is quite high: 2ms per connection when load balancing across 1000 DIPs.

Instead, we move key generation to the muxes: the mux will deterministically generate a key for every new connection, hash it to find the token and the appropriate DIP. The key is then communicated to the DIP in the encapsulated packet. The DIP does not generate a new key for the MPTCP connection, and simply uses the one provided by the mux.

To load balance data packets in a stateless way we use two separate mechanisms, shown in Figure 4.

1. Redirect traffic. When a new subflow is setup, the mux computes a hash of the five-tuple and selects the DIP (say C) that is responsible for the subflow under plain five-tuple hashing. It communicates this DIP to B, who in turn, instructs C to forward to B all packets it receives for that five-tuple. Redirection is enough to ensure correctness; however it is also inefficient: traffic will hit two servers instead of one.

2. Embed the token in regular segments. To reduce the costs of redirection, we change the server stack to embed the token in its outgoing segments and "trick" the standard MPTCP client to echo these tokens in return traffic. In particular, our servers embed the token in least significant bits of the timestamp option carried by most TCP segments except the FIN. Clients echo these values sent by the server in its timestamp options, and the muxes then use the least significant bits of the TSecr part of the timestamp to properly load balance the packets. Only packets that do not have the timestamp options are load balanced using the five-tuple and thus redirected (FIN packets). More details of the timestamp implementation, including a discussion on its correctness and safety, are given below.

The pseudocode of our mux algorithm is in Fig. 6. This algorithm has very small memory usage as for most packets in the fast path it does one packet memory access to retrieve the token and then indexes in the buckets array called srv, where each entry stores the DIP address of the server in charge of that bucket. The packet is then encapsulated and sent to the server. For SYN(JOIN) packets, the mux does more work: it hashes the five-tuple and informs the server of the redirection point. Finally, SYN(MPC) packets are the most expensive, since a key is generated and the token is computed.

```
if (MP_CAPABLE) {
    k = SHA1(fivetuple, client_key, secret);
    t = SHA1(k);h = hash(fivetuple);
    dip = srv[t % BUCKETS];
    redir_dip = srv[h % BUCKETS];
    encap(p, k, redir_dip, dip);
} else if (MP_JOIN) {
    t = retrieve from JOIN option;
    h = hash(fivetuple);
    dip = srv[t % BUCKETS];
    redir_dip = srv[h % BUCKETS];
    encap(p, redir_dip, dip);
} else {
    if (timestamp)
        t = LSB13(timestamp);
    else
        t = hash(fivetuple);
    dip = srv[t % BUCKETS];
    encap(p,dip);
}
```

Figure 6: Mux load balancing algorithm. No per-flow state is used to decide the fate of packets.

Daisy-chaining for smooth server handover. There is a natural amount of churn of servers behind a VIP due to failures, scaling-out or in or planned maintenance. To allow smooth handovers, we need a mechanism to migrate connections from a source server (A) to a target one (say B). Supporting migration is easy for the controller: it must simply re-map the A's buckets to B and then inform the muxes. The muxes will then start sending the bucket traffic to B.

For a truly smooth migration, however, there are two complications that need to be taken in account: existing connections at A will be broken during the migration and there may be temporary disagreements amongst muxes in the buckets-to-servers mappings. 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, as shown in Fig. 5. We aim to move all new connections to B, the new server, but allow old connections to continue by forwarding them to A.

To start daisy chaining, the controller informs both servers of the bucket migration. When daisy-chaining, B will: 1) Locally service packets if they are SYN(MPC) or belong to a local connection. 2) Redirect the packets to the appropriate server if they match an MPTCP-redirect rule, and 3) Daisy-chain: send all other packets to server A. Server A will process traffic as usual until all of its ongoing connections are closed or have timed-out. At this point A informs B there is no more need to do daisy chaining, completing the migration.

Daisy chaining adds robustness to our whole design. Consider what happens if the two muxes in Fig. 5 temporarily disagree on the server now in charge of the bucket being moved. Flows that hit mux 1 are load balanced according to the old mapping and will be directed to A, which processes them as usual (the black flow). Meanwhile, B will locally service the red connection and forward the blue connection to A. This robustness removes the need for strict synchronization of all muxes when buckets are remapped.

Embedding the token in timestamps. We need to encode the MPTCP token in every packet to allow muxes to forward MPTCP correctly without costly redirection. To ensure quick datacenter adoption, we seek a solution that is deployable at the servers and does not require client or protocol changes.

We rely on TCP timestamps for our purpose. Timestamps are present in all packets except the FIN packets and contain two parts, one set by the sender called $TSval$ and another called $TSecr$ that contains the value most recently received from the remote end. In particular, the server embeds X in the $TSval$ part of the timestamp on all its outgoing segments, where X is monotonically increasing in time, and the client will echo the most recent X it has seen in return packets.

In our solution, the server simply embeds the thirteen lowest order bits of the MPTCP token in the lowest order bits of the $TSval$ field as follows:

```
TSval = 0xFFFFFFFF & (TSval << 13)
TSval |= (token & 0x1FFFF)
```

The remaining question is whether our change is safe for the original uses of the timestamp option. The server's timestamps are used for two purposes: accurate server RTT measurement and protection against wrapped sequence numbers at the client.

Protection against wrapped sequence numbers. RFC 1323 [8] specifies bounds on how quickly or slowly the timestamp clock can evolve. With our change, the clock will move 2^{13} times faster than the host clock. Our implementation is based on Linux, where the timestamp clock ticks once every millisecond. Our clock will therefore appear to tick once every 122 nanoseconds, which is well within the bounds set by RFC1323.

For active connections, our solution works perfectly fine. However, problems appear for connections that are idle for long periods, such as those used by mobile phones for push notifications. In such cases, if the pause between packets is large enough, the new timestamp will be rejected by the client if it has advanced by more than 2^{31}. We solve this issue by having the server send keepalive messages at least once every four minutes.

RTT Measurement. On receipt of a packet, the server obtains the 19 least significant bits of the packet timestamp by shifting right the value it receives. Next, it fills out the most significant bits by copying them from the current value of its local clock. It then computes the RTT by subtracting the timestamp from the current clock. If the value is negative, the highest order bits must have changed (i.e. the 19-bit range has wrapped around). In this case, the server simply adds $1 << 19$ to the RTT. Our algorithm is guaranteed to work correctly as long as the packet RTT is smaller than 512s. This always holds in practice, since 512s is four times larger than the maximum segment lifetime.

Cores	SYN			Data (86B)		
	TCP	MPC	JOIN	TS	TS(Worst)	No TS
1	4.07	1.01	3.64	4.36	3.79	4.23
2	6.48	2.02	6.65	6.99	6.89	7.00
3	6.46	2.98	6.92	6.98	6.96	6.96
6	6.30	5.87	6.62	6.69	6.69	6.67

Table 1: Mux performance vs. packet type (Mpps)

4. PRELIMINARY EVALUATION

We have implemented the mux in the Click modular router suite [4] and we run it using the FastClick distribution that relies on netmap [12] to bypass the kernel for improved performance. Our servers run Linux kernel 3.18 with the MPTCP patch (version 0.90) modified such that the server uses the connection keys generated by the mux and tokens are embedded in all outgoing timestamps. We have also implemented a host agent kernel module that decapsulates and encapsulates packets, applies redirection and daisy chaining rules. These rules are managed by a userspace daemon.

We have tested the performance and correctness of our prototype in a small testbed.

Performance evaluation. We first tested a mux in isolation. The server we used has a six core Intel Xeon E5645 processor running at 3GHz, 8GB of RAM and two dual-port ten gigabit Intel 82599 NICs.

Our traffic generator is based on the *pktgen* utility from the netmap [12] suite: we modified it to generate multiple types of packets. The traffic generator can saturate a 10Gbps link with minimum sized packets. In each experiment we generate a single packet type and we measure performance at the receiver using pktgen.

Table 1 lists performance of our software multiplexer. As expected, the single core results show that processing SYN(MPC) packets is the most expensive: the mux does two SHA1 computations and packet encapsulation. In comparison, for all other packet types the encapsulation is the most expensive operation

Processing MPTCP SYN(JOIN)s is slightly more expensive than regular TCP SYN packets, as we have to perform one additional hash operation and encapsulate one more field. For smallest-size data packets (86B), direct hashing (shown as "no timestamp") is slightly slower than using timestamps for load balancing, because of the hash calculation. We also measure a worst-case scenario for the timestamp option, placing it at the end of the options field and preceding it with nops: throughput is now 3.79Mpps, 14% lower compared to the default placement used by Linux.

We then increased the number of cores used to service the single NIC, spreading the NIC queues across all the participating cores. The results for two cores show an increase in throughput for all packet types. From three cores and up, only the SYN(MPC) packets improve in performance, and performance for other types of packets hits a plateau at around 6Mpps. To understand the issue, we kept removing complexity until we were left running only the netmap bridge, but the problem still persisted; the netmap paper reports a similar pathology on the receive path: when packet sizes are between 65B and 127B receive throughput drops to 7Mpps (see [12] Fig. 6). The bottleneck is, most likely, due the high number of PCIe transactions.

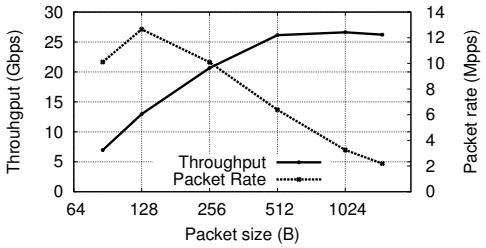

Figure 10: Mux forwarding performance vs. packet size. Our prototype can forward 26Gbps with 512B packets.

All our experiments so far have used minimum sized TCP packets and a single NIC. In our next experiment, we measured throughput for a variety of packet sizes in a mux with four ten gigabit NICs. The results are presented in figure 10 and show that the mux saturates three ten gigabit NICs when packet sizes reach 512B, and that performance for 86B TCP packets is around 11Mpps, confirming our PCIe link bottleneck theory.

How many servers can one of our muxes handle? Muxes are mainly used to handle HTTP requests from clients, so we can get an estimate by examining real-life web traffic characteristics. We examined recent traffic traces from the MAWI backbone link, finding that client to server traffic (to be serviced by the muxes) is overwhelmingly composed of small packets: more than 90% of packets are 128B or less. Of these, only 5% are SYN or FIN packets; the wide majority are TCP acks and small TCP data segments. These findings, corroborated with the performance results above imply that the mux should handle close to 10Gbps of client-to-server traffic. The MAWI traces also show that server-to-client traffic is 15 times larger than client to server traffic: a single mux can, in principle, load balance for a pool of servers that together serve 150Gbps of downlink traffic. We expect one server to source around 1-10Gbps of traffic, so a single mux should cater for 15-150 servers.

Ring size. In Fig. 7 we vary the ring size and measure the throughput achieved when forwarding SYN(MPC) packets. Ring size directly affects the performance of batching heavily used in FastClick and netmap: smaller rings will generate more interrupts and PCIe transactions, reducing performance; ring sizes above 256 slots don't help throughput but increase latency.

Latency. 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 SYN(MPC) packets sent at different rates. In parallel, we run a ping with high frequency between two idle machines, and the echo request passes unmodified through the mux. We show a CDF of ping latency for different packet rates in Figure 8. As long the CPU is not fully utilized, me-

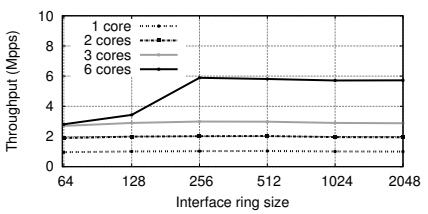

Figure 7: Effect of NIC ring size on performance.

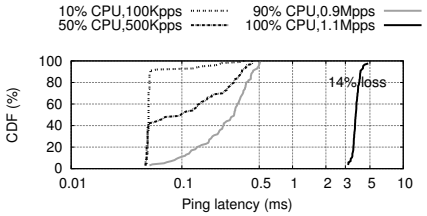

Figure 8: Latency induced by our mux

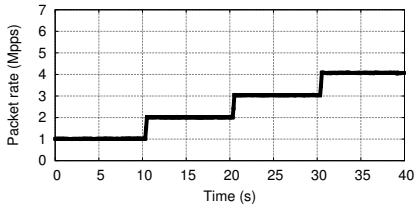

Figure 9: Scaling out muxes instantly increases available capacity.

dian and worst-case packet latencies stay below 0.5ms. When we overload the mux, the latency jumps to 3-4ms and 14% of packets are dropped. The 4ms latency is a worst case and is mostly due the time it takes one core to process all the packets stored in the 6 receive queues used by netmap (256 packets per queue). With appropriate provisioning, overloading the mux CPU can be avoided, and the latency inflation is at most 0.5ms.

Timestamp evaluation. To ensure our timestamp modifications are safe, we ran experiments varying the RTT between 1-500ms and loss rates between 0.01% to 5%, comparing the throughput of single-path MPTCP using regular timestamps or the modified timestamps. Our timestamp changes are benign: in all experiments the differences in throughput are under 10%-20%, and explained by typical TCP performance variability.

Scaling out. In a large datacenter, scaling out entails starting a new mux and sending a BGP announcement for the VIPs it serves. The border routers will then start hashing traffic to the new mux. We have a local cluster where we run experiments on machines connected by an Openflow-enabled IBM G8264 Rackswitch. In our context, using BGP is overkill; instead, we insert static rules in the switch to load balance traffic across our muxes using the source address prefix for redirection.

We generate and spread 10Gbps of SYN(MPC) packets across our muxes, which forward them to a single DIP measuring throughput. In Fig. 9 we start with a single mux running on one core and processing 1Mpps; after that, every 10s we start one more mux and split the traffic across muxes. With every additional mux, there is an increase of 1Mpps of processed traffic, as expected. In a large scale deployment we expect to see more latency before the capacity increases, perhaps on the order of seconds, due to BGP propagation delays.

5. RELATED WORK

Paasch et. al [10] discuss the problems posed by Multipath TCP 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 placement decision. They provide two possible solutions to ensure the SYN(JOIN) token can be used to select the correct server: change the MPTCP handshake mechanism such that the token is announced explicitly (and not derived from the session key) or change the way the token is derived from the key, using a block cipher instead of a hash function. Our solution sidesteps these changes to the MPTCP protocol by having muxes choose per connection keys, and also allows muxes to be stateless for data packets.

6. CONCLUSIONS

Load balancing Multipath TCP is a necessity, yet designing a scalable load balancer is far from trivial. We have used three main ideas to tackle this challenge: stable hashing via buckets, generating connection keys at muxes and embedding the MPTCP token in TCP timestamps on every packet. We have designed and implemented MPLB, a stateless software load balancer that can scale seamlessly and tolerates mux faults without dropping any connections. Our prototype can forward 10Mpps for an expected HTTP client-to-server load on a single modest Xeon machine with two NICs.

Acknowledgements

The authors would like to thank Mark Handley for his suggestion to encode the server identifier in the timestamp. This work was partly funded by SSICLOPS H2020 (644866) and UEFISCDI project Mobil4 (11/2012).

7. REFERENCES

[1] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/, 2016.
[2] NGINX - HTTP Server. http://nginx.org/en/, 2016.
[3] Daniel E. Eisenbud et al. Maglev: A fast and reliable software network load balancer. In *NSDI*, Santa Clara, CA, 2016.
[4] E. Kohler et al. The Click modular router. *ACM Trans. Computer Systems*, 18(1), 2000.
[5] Ford, Alan et al. RFC6824:TCP Extensions for Multipath Operation ... https://tools.ietf.org/html/rfc6824.
[6] Gandhi Rohan et al. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM*, 2014.
[7] Gandhi Rohan et al. Rubik: Unlocking the power of locality ... In *ATC*, 2015.
[8] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP Extensions for High Performance, May 1992.
[9] Nanxi Kang et al. Efficient traffic splitting on commodity switches. In *CONEXT*, 2015.
[10] Paasch, Christoph et.al. MPTCP behind Layer-4 loadbalancers (ID). draft-paasch-mptcp-loadbalancer-00, Sep 2015.
[11] Patel, Parveen et. al. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.
[12] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.