All Roads Lead to Rome: An MPTCP-Aware Layer-4 Load Balancer

Yijing Zeng , Milind Buddhikot*, Suman Banerjee
UW-Madison, *Nokia Bell Labs
{yijingzeng, suman}@cs.wisc.edu, *milind.buddhikot@nokia.com

***Abstract*—Multipath TCP (MPTCP) is a promising protocol that aggregates the bandwidth of mobile client's multiple interfaces. However, currently it is still not widely adopted. A key reason for this slow adoption is that the layer-4 load balancers (LBs) used to scale TCP based services in data centers are not MPTCP-aware and forward the multiple TCP subflows of the same MPTCP connection independently to different backends (BEs). In this paper, we present *RomanRoads (RR)*, an MPTCP-aware layer-4 LB which eliminates this hurdle to widespread MPTCP adoption. Compared with prior proposals, *RR* is easily deployable because it does not change the service provider's network configuration, makes no modification to ordinary TCP protocol and only minimal modification to the MPTCP connection setup process, and supports the case of multiple LBs. We implement *RR* in the form of a software LB and validate its correctness and high performance through extensive experiments. *RR* achieves 100% correctness at steady state, no connection disruption during LB churns, and line-rate throughput for packets from 5-tuple seen before. Moreover, we shed light on the desired properties of an LB-friendly multipath layer-4 protocol to provide guidance for future multipath protocol design.**

***Index Terms*—MPTCP, load balancing.**

I. Introduction

Today's mobile devices commonly support both cellular and WiFi interfaces and will evolve to support additional mmWave and shared spectrum (e.g. 3.5GHz CBRS) interfaces as 5G networks are deployed. Therefore, in order to support the exponentially growing mobile data traffic, extreme multi-connectivity where mobile devices concurrently use a combination of licensed, unlicensed and shared spectrum bands will be a norm.

Given this, techniques that exploit such multi-connectivity will be key to providing multi-Gbps throughput to future mobile applications. Multipath TCP (MPTCP) is one such promising technique. Although it is originally proposed in the data center environment, it is very suitable to boost the aggregated bandwidth of mobile devices with multiple radios as well as to improve other performance metrics e.g. latency, connection reliability. Previous work has shown that it can aggregate the bandwidth of various interfaces operating in different spectrum bands and provide a massive capacity to applications on mobile clients, improve the latency of multiple wireless interfaces by selecting primary flow wisely, and preserve the energy of multiple wireless interfaces by carefully scheduling the traffic between multipaths.

Yijing and Suman were supported in part by US NSF grants CNS-1838733, CNS-1719336, CNS-1647152, CNS-1629833, and CNS-2003129.

ISBN 978-3-903176-39-3 © 2021 IFIP

In addition to its good performance, another reason that makes MPTCP promising is that its design has the potential of wide adoption. In the TCP/IP stack, MPTCP sits on top of ordinary TCP and below the socket interface to the application layer. Multipath related signaling is realized using a TCP option field, and each subflow of an MPTCP connection is just an ordinary TCP flow. When the middleboxes between the client and the server do not support MPTCP, it can gracefully fall back to ordinary TCP.

Despite this elegant and backward-compatible design, currently MPTCP is still not widely deployed. A key reason for this slow adoption is that the layer-4 (L4) load balancers (LBs) used to scale TCP based services in data centers are not MPTCP-aware. The state-of-the-art (MPTCP-unaware) LB designs treat each subflow in an MPTCP connection as independent TCP flow, and based on the 5-tuple of the flows, they forward multiple TCP subflows of the same MPTCP connection to different backends (BEs), which eliminates the benefits of MPTCP.

This problem was noticed previously and several proposals have been submitted to address this problem. However, they all seem to lack the ability of wide adoption. They either require significant changes of the service provider's network configurations [1]–[4], or significant modifications of ordinary TCP or MPTCP protocol [2], [3], [5], [6], or do not support multiple LBs [2], [3], [6], [7]. Thus, a widely deployable MPTCP-aware L4 LB is still an open problem.

RomanRoads(*RR*): In this paper, we present our design of an MPTCP-aware L4 LB, *RomanRoads (RR)*, whose name comes from the idiom "All roads lead to Rome". In addition to correctly forwarding the multiple TCP subflows of the same MPTCP connection to the same BE, we aim at high performance in terms of throughput and ease of deployment.

We enhance the current state-of-the-art (MPTCP-unaware) LBs in two ways to achieve correctness and high performance: (1) We move the MPTCP key/token generation function to LB and use two local connection tracking tables, one for ordinary TCP and the other for MPTCP, to ensure LB can forward multiple TCP subflows of the same MPTCP connection, identified by the unique token, to the same BE. (2) When there are multiple LBs, we divide the MPTCP token space using consistent hashing to guarantee uniqueness and forward packets between LBs to avoid tight global synchronization. Compared with previous proposals, our design is widely deployable because it does not change the service provider's network configuration, makes no modification to ordinary TCP protocol and only minimal modification to MPTCP connection setup process, and supports multiple LBs for better scalability. Moreover, we discuss the desired properties of an LB-friendly multipath L4 protocol based on our experience with *RR*, which fills a critical gap of previous multipath L4 protocol designs regarding their practical deployment issues.

To summarize, our main contribution is the followings:

- We propose *an easily deployable MPTCP-aware L4 LB design with a high performance* called *RR*. *RR* does not change the service provider's network configuration. Moreover, *RR* makes no modification to ordinary TCP and only minimal modification to the MPTCP connection setup process without requiring new option fields. Last, *RR* supports the case of multiple LBs for large scale deployment.
- We implement *RR* in the form of software LB, although the design itself has the potential to support both software and hardware implementations. Extensive experiments demonstrate the correctness and high performance of *RR*. We *achieve 100% correctness at steady state, and no connection disruption during LB join/leave/failure events*. In addition, although we compromise some throughput for different SYN packets to make the LB MPTCP-aware, we still *achieve line-rate processing for packets from 5-tuple seen before*.
- *We shed light on the desired properties of an LB-friendly multipath L4 protocol* based on our experience with *RR*, to provide guidelines for future multipath protocol design. In order to be L4 LB-friendly, the multipath L4 protocol should have the following two properties. First, the unique connection identifier (CID) should be generated only at the client side and server side CID should be avoided. Second, the unique CID should be presented in every packet. MPTCP does not have these two properties thus it makes *RR* design relatively complex compared with MPTCP-unaware LBs.

II. Background

A. L4 Load Balancing in Data Center

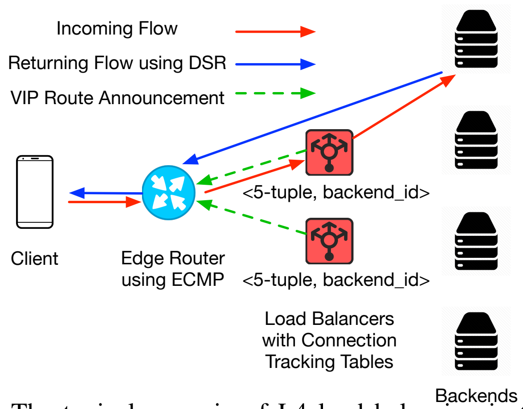


Fig. 1: The typical scenario of L4 load balancing in today's data center.

Fig. 1 illustrates how a service provider uses L4 load balancing to scale a service offered over TCP/IP that needs to be scaled to millions of users. For each service supported in a data center, one public IP address, called Virtual IP (VIP) is advertised via DNS to end-users (clients). However, to achieve high availability and scalability, a large group of BEs serves the incoming flows simultaneously. Thus, LBs are configured between the edge router[1] and the BE pool. Each LB announces a route to the VIP with the same cost to the edge router so that incoming flows to this VIP are evenly distributed across all LBs using Equal Cost Multi Path (ECMP) routing (i.e. L3 load balancing). They then decide which BE an incoming flow goes to and forward the packets to the decided BE. For a new incoming flow, identified by the 5-tuple, an LB usually uses consistent hashing on the 5-tuple to decide which BE the packets are forwarded to and adds an entry to its connection tracking table. For all subsequent packets of known flows, the packets are forwarded based on the information in the connection tracking table. For the returning traffic, state-of-the-art designs usually support direct server return (DSR), i.e. the returning traffic directly goes to the edge router without passing through LBs, which makes serving a very large number of flows possible. In this case, the LB does not split the L4 connections, which is different from that of popular reverse proxies (L7 LBs) such as HAProxy [8], which split TCP connections. Current implementations of LB include software LB [9], [10], switching ASIC/programmable hardware [11], and the hybrid [12].

Additionally, in a larger context where multi-level load balancing is needed, the BEs in Fig. 1 can be L7 LBs, which are the endpoints of L4 connections from clients.

B. MPTCP Connection Setup

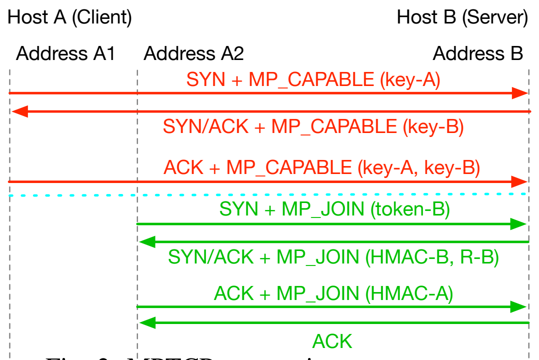


Fig. 2: MPTCP connection setup process.

MPTCP is an extension of the TCP protocol using the TCP option standardized in RFC6824 [13]. An MPTCP connection usually contains several subflows, each of which is an ordinary TCP flow. Thus, MPTCP is an L4 protocol that sits on top of TCP protocol and below the application layer.

Fig. 2 shows the process to successfully set up an MPTCP connection with multiple TCP subflows. The primary flow is established first via a 3-way handshake with an MP_CAPABLE option, during which 64-bit keys of both sides are exchanged. Then, in order to add a secondary flow to this MPTCP connection, the client sends a SYN MP_JOIN packet with token-B. Token-B is the most significant 32 bits of SHA1 hash on key-B and it uniquely identifies this MPTCP

[1]With ICMP support of the edge router and the client's path MTU discovery mechanism, we do not consider IP fragmentation in LB design.

Proposals	**Change Net Config.**	**Modify TCP**	**Modify MPTCP**	**Support Multi. LBs**
Ideal	No	No	No	Yes
RR	No	No	Yes, minor	Yes
P1 (Proposal1 of [2], [1])	Yes	No	No[1]	Yes
P2 (Proposal1 of [3], [4])	Yes	No	No[2]	Yes
P3 ([7])	No	No	No	No
P4 (Proposal2 of [2], Proposal2 of [3], [6])	No	Yes	Yes, major	No
P5 ([5])	No	No	Yes, major	Yes

[1] It forces BE to announce its second port number after primary flow setup before secondary flow setup, which needs both client/server OS changes.

[2] It forces BE to announce its second public IP after primary flow setup before secondary flow setup, which needs both client/server OS changes.

TABLE I: Comparison of the ideal solution, *RR*, and previous proposals.

connection on the server side. Three packets afterward finish the authentication process to establish the secondary flow[2].

In the context of a data center, from the mobile client's point of view, all TCP subflows of the same MPTCP connection need to go to the same BE in order to fully take advantage of MPTCP. However, it is apparent that this requirement cannot be guaranteed in the LB design in § II-A. From the service provider's point of view, LBs being unaware of MPTCP may generate a large number of unnecessary outgoing RST packets because BEs receive unexpected TCP packets, which also hurts the performance of the data center.

C. Deficiencies of Previous Proposals

We describe previous proposals to solve this problem and why there are unable to be widely adopted. Table I summarizes their properties and compares with the ideal solution and *RR*.

P1 (Proposal1 of [2] and [1]): This method changes the network configuration of the service provider so that every BE has a different second port number. Thus, after primary flow setup, the BE informs the client its second port number and the client can use this as the destination port number for secondary flow so that LBs can decide which BE this traffic goes to. However, even if not considering the number of BEs can be larger than possible available port numbers, this proposal complicates the NAT and firewall configuration of the service provider. In Ananta [10], every outbound packet is NATed from internal IP to VIP. Suppose a BE supports two services (VIPs), the second port number does not provide any information which service the packet belongs to, thus it cannot be NATed.

P2 (Proposal1 of [3] and [4]): This method similarly changes the network configuration of the service provider, but uses a different second public IP rather than a port number to distinguish each BE. This is also undesirable because public IP is a scarce/costly resource.

P3 ([7]): This method parses the ACK MP_CAPABLE packet to get key-B generated at BE and thus is able to forward secondary flow correctly. However, it does not consider the scenario that the primary and secondary flow reach different LBs for multiple LBs case and possible token collision when serving a large number of clients. Furthermore, it actually parses every ACK packet from 5-tuple seen before, so its throughput performance is not satisfying.

[2]HMAC-A=HMAC(Key=key-A+key-B, Msg=R-A+R-B), HMAC-B= HMAC(Key=key-B+key-A, Msg=R-B+R-A). R-A and R-B are random nonces.

P4 (Proposal2 of [2], Proposal2 of [3], [6]): This method generates token-B/key-B at the LB and embeds 14 bits of the token into the TCP timestamp option of every packet so that the LB can know the BE of secondary subflow. The timestamp option is commonly used by current TCP implementations, so it is a drastic change, especially at the client side. This method also significantly modifies MPTCP because the LB needs to inform the BE the selected key-B using a new IP option, which requires IETF approval. Moreover, neither does it consider the case that the primary and secondary flow reach different LBs.

P5 ([5]): This method modifies the token generation process so that each BE can only generate token within its own range, thus the LBs can infer which BE the secondary flow goes to. Nevertheless, it significantly modifies MPTCP such that the connection setup process is drastically different from what we described in § II-B. Tokens rather than keys are exchanged in the first two packets, and (non-standardized) MPTCP SYN cookies must be enabled.

For an ideal solution, it should not change network configurations of service providers, should make no modifications to ordinary TCP and MPTCP, and support multiple LBs for good scalability. However, none of the previous proposals satisfy these requirements simultaneously. If it is impossible to meet all the requirements, sacrificing MPTCP slightly is tolerable compared with violating other requirements, given that it is still not widely deployed.

III. *RR* Design

In this section, we present the design of *RR*, a widely deployable MPTCP-aware L4 LB. We start with our design principles, assumptions, and constraints. Next, we consider the simple case where we have only one LB, and then extend the design to the case where we have multiple LBs. Finally, we summarize the pros and cons of our design. Note that we mainly focus on data plane design and it is compatible with most modern control plane designs, e.g. logically centralized but physically distributed controller in SDN [1], [9], [10].

A. Principles, Assumptions, & Constraints

Design principles: In our design, we aim at high packet processing speed and ease of deployment, in addition to always correctly forwarding multiple TCP subflows of the same MPTCP connection to the same BE.

Design assumptions: We assume that the client has multiple IP addresses/ports and the service provider has only one VIP address and port number per service known to the client.

Constraints: Based on these design principles and assumptions, we have the following constraints on the design space: (1) Our design cannot modify the traditional TCP protocol and can only make minimal modification on the MPTCP protocol without requiring new option fields that need IETF standardization. (2) There is no constraint on the client's configuration, but only minimal modification on the service provider's network configurations (e.g. NAT and firewall) is possible. Moreover, clients are agnostic to the service provider's network configurations. (3) We also need to support DSR and multiple LBs to have good scalability. (4) Only a small disruption is tolerable when a BE or an LB goes up or down. (5) Our design should have no constraints on the implementation method, and should be amenable to software, switching ASIC, or hybrid implementations.

B. The Case of Single Load Balancer

We present our design by addressing the following three questions. Without any loss of generality of our design, hereon we only consider (MP)TCP traffic of one service.

Who generates the unique connection identifier key-B/token-B? Because BE is the endpoint of the MPTCP connection, on the first thought, without considering the LB, the BE generates key-B/token-B. However, since the token is used to identify an MPTCP connection, in the case when the number of BEs is large, it is easier to guarantee its uniqueness if the LB generates the token. Moreover, in DSR, the SYN/ACK packet with key-B will not be routed via the LB if the BE generates it. Therefore, if key-B/token-B are generated at BE, we need to tightly synchronize this information between BE and LB. Nevertheless, experience from previous work [9], [10] has shown that global synchronization should be avoided due to its high cost. Thus, we move the key/token generation function to the LB.

How to inform the BE of the selected key-B? We take advantage of existing fields in MPTCP suboptions, which is a nonintrusive way to inform this information. Recall that the third packet in the 3-way handshake to set up the primary flow is an ACK MP_CAPABLE packet with both key-A and key-B. We, therefore, decide to piggyback the key-B selected by the LB into the MP_CAPABLE option field similar to the ACK MP_CAPABLE packet. Note that this method requires a small modification to the MPTCP server side code to extract key-B and the LB needs to recompute the TCP header checksum and the IP header checksum. However, it does not require a new TCP option, MPTCP suboption, or IP option.

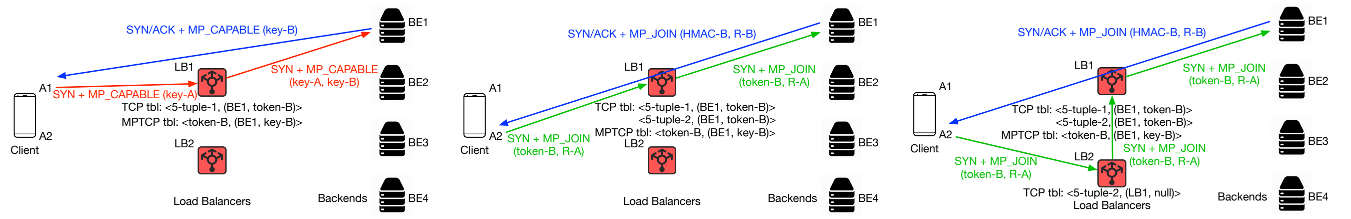
How to ensure packets of secondary subflows are sent to the same BE, i.e., the correctness of flow forwarding in this case? We maintain two connection tracking tables, one for ordinary TCP connections and the other for MPTCP connections. The ordinary TCP connection tracking table is defined as <5-tuple, (BE_id, token-B)> and the MPTCP connection tracking table is defined as <token-B, (BE_id, key-B)>. When the primary flow arrives, we allocate a unique key/token and decide which BE it goes to using consistent hashing on the 5-tuple of the primary flow. We add one entry to each table to record the necessary information. Then, when the secondary flow arrives, we look up the MPTCP connection tracking table using the token-B in the SYN MP_JOIN packet to find the BE it goes to, and add an entry to the ordinary TCP connection tracking table to record this decision. Fig. 3(a) and Fig. 3(b) illustrate the process (ignore the second LB for now). For all subsequent packets of primary or secondary flow, we look up the ordinary TCP connection tracking table to get the BE_id and the token-B, then look up the MPTCP connection tracking table using the token only for the purpose of updating the last used time of this entry. In this way, *RR* guarantees the correctness for the single LB case. Note that we also include key-B in the MPTCP table because it is possible to receive duplicate SYN MP_CAPABLE packets from the same client. In this case, we should not allocate a new key/token. Instead, we should use the same CID allocated previously. Therefore, including this field makes *RR* more robust to SYN MP_CAPABLE flood.

C. The Case of Multiple Load Balancers

In the following, we extend the design of *RR* for the single LB to support multiple LBs and achieve scalability, and similarly present our design by addressing four questions.

Does an LB need to know the key/token generated by others? If the allocated key/token information is available to all LBs, the process of generating a unique new key/token is very simple. One can simply randomly generate the key and check whether its corresponding token is in the MPTCP connection tracking table. However, making allocated key/token information available to all LB needs costly global synchronization. Thus, an *RR* LB does not know which key/tokens are allocated by others in our design.

How to ensure the generated key/token is unique among multiple LBs? Without knowing the allocated key/tokens of others, we can still guarantee the uniqueness by dividing the token space into non-overlapping subspaces and each LB can only generate a key whose corresponding token falls into its own subspace. In order to have minimal disruption when an LB goes up or down, we still apply the idea of consistent hashing to divide the token space. For example, we can use a hash function on the LBs' MAC addresses to map them onto the token space and require each one can only generate token falling into its right-hand side. Fig. 4 illustrates this idea. Squares represent different LBs and circles represent different allocated tokens. Each LB can only generate a key whose corresponding token falls into its own subspace, indicated by the same color. Note that the LBs need to regenerate the key if the key does not meet the requirement, and the average number of tries to generate a valid key is roughly equal to the number of LBs because SHA1 is one-way function and its uniformity can guarantee the size of each subspace is roughly the same. If a valid token/key cannot be generated in a predefined number of trials since the system is overwhelmed, the LB drops the packet to force the client to fall back to ordinary TCP. In addition to key/token uniqueness, dividing token space using consistent hashing also ensures that each LB is aware of every other LB's subspace.


SYN/ACK + MP_CAPABLE (key-B)
BE1
LB1
SYN + MP_CAPABLE
(key-A, key-B)
BE2
A1
SYN + MP_CAPABLE (key-A)
TCP tbl: <5-tuple-1, (BE1, token-B)>
MPTCP tbl: <token-B, (BE1, key-B)>
A2
Client
LB2
BE3
Load Balancers
Backends
BE4
SYN/ACK + MP_JOIN (HMAC-B, R-B)
BE1
LB1
SYN + MP_JOIN
(token-B, R-A)
BE2
A1
TCP tbl: <5-tuple-1, (BE1, token-B)>
<5-tuple-2, (BE1, token-B)>
MPTCP tbl: <token-B, (BE1, key-B)>
A2
SYN + MP_JOIN
(token-B, R-A)
Client
LB2
BE3
Load Balancers
Backends
BE4
SYN/ACK + MP_JOIN (HMAC-B, R-B)
BE1
LB1
SYN + MP_JOIN
(token-B, R-A)
BE2
A1
TCP tbl: <5-tuple-1, (BE1, token-B)>
<5-tuple-2, (BE1, token-B)>
MPTCP tbl: <token-B, (BE1, key-B)>
A2
Client
SYN + MP_JOIN
(token-B, R-A)
LB2
SYN + MP_JOIN
(token-B, R-A)
BE3
TCP tbl: <5-tuple-2, (LB1, null)>
Load Balancers
Backends
BE4

(a) Primary flow setup. (b) Secondary flow setup if it reaches the same LB (c) Secondary flow setup if it reaches a different LB.

Fig. 3: MPTCP connection setup process through *RR* LBs. Only the initial 2-packet exchange is shown.

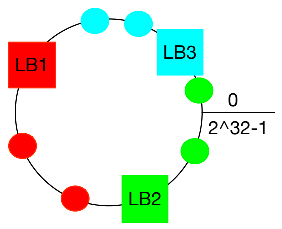


Fig. 4: Token space division using consistent hashing.

How to ensure subflows reaching different LBs go to the same BE, i.e., the correctness of flow forwarding in this case? Suppose the primary flow is established as shown in Fig. 3(a). Then, LB1 knows which BE this MPTCP connection goes to and token-B is within LB1's token subspace. If the secondary subflow goes to LB2 as shown in Fig. 3(c), LB2 does not know which BE this MPTCP connection goes to based on this token-B, but it knows that LB1 is responsible for this token-B. Thus, LB2 forwards this flow to LB1, and LB1 can forward it to the right BE. LB2 also adds an entry <5-tuple-2, (LB1, null)> to its ordinary TCP connection tracking table to record this decision. Note that "null" represents an invalid token and LBs can be considered as a special group of BEs. In this way, *RR* guarantees the correctness when there are multiple LBs. Although redirection is not very desirable in terms of some performance metrics, e.g. latency, it is unavoidable given no global information. Redirection is also involved in Maglev [9] to handle IP fragments and Beamer [1] to handle BE changes.

How to support events of LB join/leave/failure? To add minimal complexity compared with ordinary LBs, these events are not handled differently except for LBs being informed that new token space division should be computed. Note that the control plane is involved during these events, e.g. route announcement/retrieval and health check request/response. However, with the help of SDN, which offers more QoS to control plane traffic compared with data plane traffic, we do not consider complicated control plane issues, e.g. lost/slow health check response, because they are not MPTCP specific problems and should be solved by control plane itself.

For LB join, one obvious negative effect of this simple solution is the broken secondary flows, which are routed to the new LB by the edge router. In addition, there are two issues that need analysis, i.e., possible connection disruptions and the negative effects of possible token collisions. For possible connection disruption, the necessary and sufficient condition of making connection disruption impossible is that there is no BE change since serving the oldest connection still in the system, because consistent hashing on primary flow's 5-tuple can find the right BE. This necessary and sufficient condition of making connection disruption impossible is the same as MPTCP-unaware LB's join event. If this condition is not met, the possibility of connection disruption is in fact slightly better than MPTCP-unaware LB under the same BE changes because the new LB can route the secondary flow to the right BE with a tiny probability. However, it is actually worse than the case that multiple paths experience independent failures. For the negative effects of possible token collisions, let the collided MPTCP connections are 1) C_o, established via LB_o and goes to BE_o, and 2) C_n, trying to set up via new LB_n to BE_n. If $BE_n \neq BE_o$, C_n will set up successfully, and the secondary flow of C_n can be added smoothly even if it is routed to LB_o by the edge router because LB_o is not responsible for this token now. [3] If $BE_n = BE_o$, which can happen with a tiny possibility, C_n will fall back to ordinary TCP at the BE because this token is already in use. Another corner case is C_o has a new secondary flow. It will be forwarded to BE_n by LB_n but the authentication process in Fig. 2 will fail. Thus, although it cannot be added to C_o successfully, it does not affect C_n.

Overall, the join procedure of *RR* is the same as that of MPTCP-unaware LB except for computation of the new token space division, and the possibility of connection disruption is negligibly better. The downsides are (i) a portion of established secondary flows are aborted harshly because they are routed to the new LB by the edge router, (ii) a portion of established MPTCP connections cannot be added with new secondary flows because the new LB is responsible for but unaware of their tokens, and (iii) a new MPTCP connection can be forced to fall back to ordinary TCP with a tiny possibility due to token collision.

We directly give the conclusions of LB leave (soft failure) as the following and omit the analysis similar to LB join. Overall, the leave procedure of *RR* is the same as that of MPTCP-unaware LB except for computation of new token space division, and the possibility of connection disruption is negligibly better. The downsides are (i) a portion of established secondary flows are aborted harshly because they either were routed from other LBs to the left LB or were routed from the edge router to the left LB but are now routed from the edge router to an LB which was not the final hop, (ii) a portion of established MPTCP connections cannot be added with new secondary flows because their information is gone with the

[3]However, C_o's established subflows are still being served by LB_o.

left LB, and (iii) a new MPTCP connection can be forced to fall back to ordinary TCP with a tiny possibility due to token collision.

For hard failure, it is equivalent to soft failure conceptually. The difference in practice is that it needs some time for the edge router and other LBs to notice the failure, usually on the scale of several seconds (default health check response timeout) to a few minutes (default BGP hold time).

D. Pros and Cons

We summarize the pros and cons of our design within the constrained design space as follows.

Pros: Our design can achieve high packet processing speed, as it does not require any state information to be tightly synchronized among all LBs or BEs. In addition, since we use consistent hashing to divide the token space into non-overlapping subspaces, there can be only $O(1/n)$ (n is number of LBs) of token space changes when an LB join/leave/failure. Moreover, because we use consistent hashing on the primary flow's 5-tuple to decide which BE an MPTCP connection goes to, the LBs can still find the right BE with high probability for the primary flow when there is an LB churn or even worse a major traffic shuffle due to bad ECMP implementation. In other words, almost no MPTCP connection is disrupted.

Cons: Our design is stateful and needs two connection tracking tables. Thus, we have to reserve more memory for these tables on the LBs. Furthermore, when there are multiple LBs, one may need to try several times (on the order of $O(n)$) in order to get a valid new key/token, which adds a computation overhead. Additionally, when there is an LB churn, a significant portion of secondary flows cannot find the right BE with high probability. Lastly, since we are redirecting the secondary flow to one more LB if it reaches a different LB from the primary flow, we add marginally more latency to the secondary flow packets, and it may cost double bandwidth, CPU cores, etc.

IV. Implementation

The design of *RR* supports both software and hardware implementation. However, in this section, we only present the details of software implementation. The reasons that we choose to implement *RR* in software first rather than hardware are the following. First, software LB offers higher flexibility and can be implemented as well as deployed on ordinary service machines in a data center. On the contrary, hardware implementation requires specific switching ASICs. Second, state-of-the-art switching ASICs have only 100MB SRAM [11], limiting the size of the connection tracking tables. Nevertheless, the design of *RR* requires two connection tracking tables, which demands more SRAM than MPTCP-unaware LBs. Lastly, *RR* is designed to mainly support mobile clients and to promote the deployment of MPTCP protocol in mobile clients as well as service providers. Thus, given the still growing number of MPTCP-enabled mobile clients, it is tolerable to not support full bisection traffic and have latency on the order of 1ms or less.

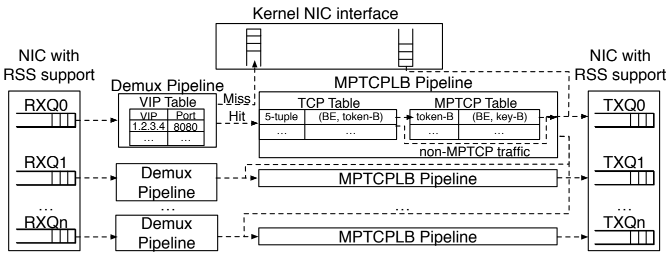


Fig. 5: The software implementation of *RR*.

Fig. 5 gives an overview of the software implementation of *RR*. We use an open-source kernel bypass platform called Data Plane Development Kit (DPDK) offered by Intel [14]. Compared with kernel resident solutions, it has a better throughput performance on the data plane. In order to reach the highest performance, we leverage the Receive Side Scaling (RSS) function of NIC hardware so that we can receive packets from multiple RX queues (RXQs) in parallel. *RR* mainly contains two types of packet processing pipelines, which are Demux pipelines and MPTCPLB pipelines. Each pipeline is realized in a separate thread and mapped to a different CPU core. The purpose of the Demux pipelines is to demultiplex the traffic based on its requested service. The MPTCPLB pipelines realize MPTCP-awareness described in § III. If the chosen BE of a packet is not another MPTCPLB pipeline, the pipeline sends the packets to its corresponding TX queue (TXQ) of the NIC. We also use a kernel NIC interface to forward the control plane traffic to the LB, e.g. health check request/response, BGP traffic. In addition, previous layer-4 LB implementations usually leverage least recently used (LRU) hash table [11] to realize lookup tables. We, however, use extendable hash table, which reserves a relatively small amount of memory that can be used when a bucket of the hash table is full. We also add a timestamp field to each entry so that if the time difference between when the entry is looked up and when the entry is created is larger than a threshold, it is considered as outdated. The reason of these two modifications is that we require entries to expire in a timely manner such that the LB can record more valid routes for secondary flows than LBs using LRU tables.

For the service BE implementation, we modify the MPTCP Linux Kernel implementation [15] so that the server can extract key-B from SYN MP_CAPABLE packet and calculate the corresponding token-B if it contains both key-A and key-B. If the token-B is still available at this BE, the BE uses this token-B to identify this MPTCP connection. If it is not, which can happen when the BE serves multiple MPTCP based services or during LB changes, the server simply considers this packet as an ordinary SYN packet. The patch based on the above logic is only 30 lines of C code, and the application of it to all server machines in a data center is much easier compared to the client side kernel upgrade, which is required by **P1/P2/P4/P5**.

V. Evaluation and Results

We evaluate *RR* through extensive experiments and focus on *RR*'s correctness (§ V-A) and its performance (§ V-B), i.e. throughput. We compare *RR* with an MPTCP-unaware LB similar to the design of Google Maglev [9] but implemented in DPDK (denoted as TCPLB hereafter), as well as **P1** to **P3**. **P4** and **P5** are excluded from quantitative comparison due to their significant modifications to (MP)TCP protocols (i.e. OS kernel). Moreover, we do not evaluate complicated corner cases due to control plane issues because it is beyond the scope of this paper. We also omit the evaluation of *RR*'s load balancing evenness since the evenness of Maglev hashing has been shown [9] and we use the same algorithm. The results of latency performance is omitted due to space limit, but experiments show that we introduce sub-millisecond latency in the worst case, which is tolerable for mobile clients.

All experiments in the following are conducted in CloudLab [16] using machines that have two 14-core 2.00 GHz Intel CPUs, 256GB RAM, and dual-port Intel 10GbE NIC with ixgbe driver in the same intranet. Moreover, if controlled traffic is needed, its packets are generated using Scapy [17], a python supportive packet manipulation tool. Mobile clients with WiFi and cellular access are not used for experiments due to the uncontrollability of link quality, latency, etc.

A. Correctness

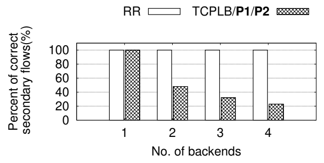
1) Single Load Balancer Case: **Light Load Setting.** In this setting, we first send a SYN MP_CAPABLE packet from a client using a specific IP address and port number to the LB, and the LB assigns a BE to serve this request. Then, we send 100 SYN MP_JOIN packets with the corresponding token-B from the same IP address but 100 random port numbers to the LB. We vary the number of BEs from 1 to 4 and record the percent of correctly forwarded SYN MP_JOIN packets (secondary flows). Note that for the LB, the lookup hash table has enough number of entries so that no entry will be added unsuccessfully due to hash collision and lack of space, i.e. the LB is lightly loaded. Fig. 6(a) shows the result. We can see that *RR* achieves 100% correctness as the number of BEs increases. On the contrary, the correctness of TCPLB decays as $1/n$ (n is the number of BEs), which complies with the theoretical analysis. Moreover, **P1** and **P2** have the same performance with TCPLB because in our setting there is no second port number/public IP for each BE.

Heavy Load Setting. When the number of active flows and the number of entries of the lookup hash table are on the same order, failures to insert an entry into the table happen frequently due to hash collision and lack of space, and the subsequent packets of secondary flows cannot find the correct BE with high probability in this case. Thus, in heavy load setting, we change the number of SYN MP_JOIN packets from the same IP address and random port numbers in the previous setting to the number of entries of the lookup hash table (without counting the extra space for extendable hash table) and see how many percents of entry insertion failures we encounter. We choose the LRU hash table as the baseline and vary the extra space of *RR*'s extendable hash table from 1/8 of the number of lookup hash table's entries to 3/8 of it. Note that the default number of entries per bucket in the hash table is 4 in our implementation (same for extra space of extendable hash table). Fig. 6(b) shows that for the LRU hash table, it encounters 20% entry insertion failures due to hash collision and lack of space when the hash table is large enough. By reserving extra space, *RR* decreases the percent of entry insertion failures significantly and reaches 0.5% when we reserve 3/8 of the number of lookup hash table's entries.

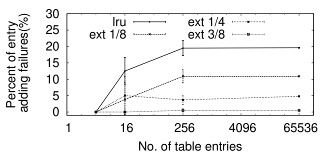
2) Multiple Load Balancers Case: **Steady State.** We modify the light load setting by using two LBs and sending the SYN MP_CAPABLE packet and SYN MP_JOIN packets to different LBs. We compare *RR* with **P3** and the result is the same as Fig. 6(a) with **P3** equivalent to TCPLB/**P1**/**P2**, which complies with our analysis in § II-C.

Hard Failure. In the previous settings, there is no LB or BE failure. In this setting, we focus on LB failure. We serve 100K continuous MPTCP connections from the clients with multiple LBs and 100 BEs. Each connection has one primary flow and one secondary flow using random port numbers. After primary and secondary flows are established, we make a hard failure to one LB. Moreover, there is no BE change during this test. Fig. 6(c) shows the percent of broken connections, broken secondary flows, and affected connections, i.e. they cannot accept new secondary flows after the hard failure. From Fig. 6(c) we notice that $O(2(n-1)/n^2)$ secondary flows (where n is the number of LBs) are broken due to the LB hard failure, including $O(1/n^2)$ each from $n-1$ other LBs and $O(1/n)$ from the edge router minus $O(1/n^2)$ ones which are now directly routed to the original second hop LB. In addition, roughly $O(1/n)$ connections are affected. However, because we use consistent hashing on primary flow's 5-tuple to decide the BE, no MPTCP connection is broken after the hard failure regardless of the number of LBs. Even if there is a concurrent BE change, the possibility of a broken MPTCP connection is minimized. This shows *RR*'s robustness to LB failures.

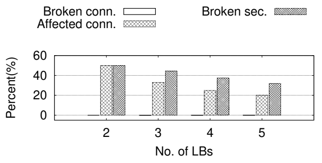
Dynamic LB Join/Leave. Similar to the previous setting, we serve 100K connections with multiple LBs and 100 BEs. The duration of each connection follows a normal distribution with mean 60s and standard deviation 10s, and each connection restarts using new random ports after its current duration is finished. At time=0s, we have 5 LBs. One LB leaves the system at time=120s, and rejoins the system at time=240s. Another LB leaves the system at time=360s, and rejoins the system at time=480s. In addition, 25 more BEs are added to the system at time=320s. Fig. 6(d) shows the percent of broken connection, broken secondary flows, and affected connections, i.e. the connections cannot be added with new secondary flows or fall back to ordinary TCP, as functions of time. For LB join, $O(1/n)$ secondary flows are broken and $O(1/n)$ connections are affected. For LB leave, $O(2(n-1)/n^2)$ secondary flows are broken and $O(1/n)$ connections are affected. Nevertheless, no connection is broken during the first LB join/leave because the necessary and sufficient condition of making connection disruption impossible is satisfied. For the second LB leave, this condition is not satisfied but only 1.55% connections are broken. It is 0.02% smaller than TCPLB in the same setting, which shows the robustness provided by multipath is not



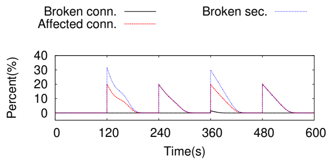
(a) Correctness of single LB when it is lightly loaded.



(b) Correctness of single LB when it is heavily loaded.



(c) Correctness of multiple LBs under hard failure.



(d) Correctness of multiple LBs under dynamic join/leave.

Fig. 6: Correctness of *RR* LB in various settings.

obvious for LB churns. Moreover, it takes roughly one average connection duration for the system to becomes stable, which is tolerable. Overall, *RR* is robust to dynamic LB join/leave.

B. Performance

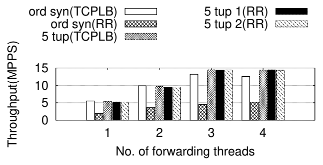
We evaluate the throughput of *RR* and compare with TCPLB and **P3**. Results of **P1** and **P2** are omitted due to their poor correctness in our setting. We first consider the throughput of a single *RR* LB and then multiple LBs, for a single service.

Throughput comparison with TCPLB/P3. Since TCPLB only differentiates two kinds of packets, i.e. ordinary SYN (ord syn) and packets from 5-tuple seen before (5 tup), we show the throughput for these two kinds of packets as a function of the number of forwarding threads in Fig. 7(a). For ordinary SYN packets, we can see that the throughput of TCPLB reaches line-rate when the number of forwarding threads is no less than 3. However, *RR* can only support roughly 1/3 of the throughput of TCPLB. The reason for this throughput decrement is that we need to parse the TCP option field and check whether a SYN packet contains MPTCP options, i.e. MP_CAPABLE or MP_JOIN, even if it does not. For packets from 5-tuple seen before, we observe no difference between the throughput of *RR* and TCPLB and between ordinary TCP packets going through TCP table only (5 tup 1) and MPTCP packets going through both TCP and MPTCP tables (5 tup 2) in *RR*. Line-rate is also reached when the number of forwarding threads is no less than 3 for packets from 5-tuple seen before. For **P3**, multi-threading is equivalent to multiple LBs, which it does not support, so we only give its throughput of one thread without including it in Fig. 7(a). Its throughput for packets from 5-tuple seen before is only 2.5MPPS, 1/2 of other LBs, because it needs to parse every ACK to see if it is an ACK MP_CAPABLE. Moreover, its throughput for ordinary SYN packets is 2MPPS, which is the same as *RR*.

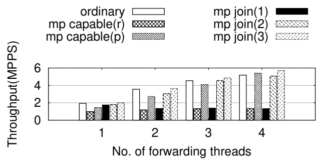
Throughput for different SYN. Fig. 7(b) shows the throughput of *RR* for different SYN packets, including ordinary SYN, SYN MP_CAPABLE, and SYN MP_JOIN. We have two versions of implementation for SYN MP_CAPABLE. (1) In a reactive implementation (mp capable(r)), key/token are allocated in a lazy manner. The MPTCPLB pipeline only starts computing for a valid key/token when a new MPTCP connection request arrives. (2) In a proactive implementation (mp capable(p)), we have a separate thread that does key/token generation for each MPTCPLB pipeline only, so that the MPTCPLB pipeline can always pull a valid key/token from it. We additionally differentiate SYN MP_JOIN into three cases, i.e. going to the same forwarding thread within an LB by forwarding between different threads (mp join(1)), going to all forwarding threads inside an LB without forwarding between threads (mp join(2)), going to a different LB (mp join(3)). Ordinary SYN and the case 2 & 3 of SYN MP_JOIN have the highest throughput because only TCP table entry adding is needed. The case 1 of SYN MP_JOIN has relatively stable throughput regardless of the number forwarding threads because packets are forwarded to one particular thread eventually. The throughput for SYN MP_CAPABLE in reactive implementation is the lowest and stays at 1MPPS without increasing much when using multiple forwarding threads. The reason is that by using multiple forwarding threads, the number of trials to generate a valid key/token also significantly increases. However, in proactive implementation, SYN MP_CAPABLE has similar throughput compared with ordinary SYN and the case 2 & 3 of MP_JOIN. In addition, the throughput of **P3** (with one forwarding thread) for SYN MP_CAPABLE is the same as its throughput for ordinary SYN because it does not differentiate these two. For SYN MP_JOIN, **P3**'s throughput is 1.9MPPS, which is the same as the case 1 & 2 of SYN MP_JOIN in *RR*.

Throughput using different hash tables. Fig. 7(c) shows the throughput using three different hash tables, i.e. LRU hash table (lru), extendable hash table without timely entry timeout support (ext), and extendable hash table with timely entry timeout support (time). From Fig. 7(c) we observe that there is no significant difference in throughput for ordinary SYN packets and packets from 5-tuple seen before. The throughput of extendable hash table with timely entry timeout support for ordinary SYN is slightly lower than the other two because we remove the outdated entries in the same bucket first before adding a new entry into the bucket.

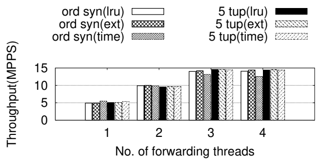
Throughput of multiple LBs. The throughput for other packets can be easily generalized to the case of multiple LBs except for SYN MP_CAPABLE. Fig. 7(d) compares the throughput of multiple LBs all using a single forwarding thread and the throughput of multiple forwarding threads in a single LB for both reactive and proactive implementation. We notice from Fig. 7(d) that for both reactive and proactive implementation, the throughput for SYN MP_CAPABLE using multiple LBs is slightly higher than that using multiple forwarding threads. The reason is that even if the multiple forwarding threads are mapped to different cores of CPUs, they still contend for other resources, e.g. memory of packet queues. Thus, if there are machines available, having multiple LBs is more desirable than having multiple forwarding threads. Moreover, the throughput for SYN MP_CAPABLE in proactive implementation has better scalability than that



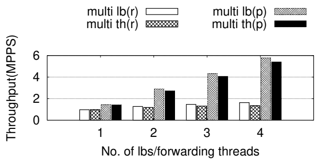
(a) Single LB throughput comparison with baseline.



(b) Single *RR* LB throughput for different SYN packets.



(c) Throughput using different hash tables.



(d) Throughput for SYN MP_CAPABLE of multiple LBs and multiple threads.

Fig. 7: Throughput performance of *RR* in various settings.

in reactive implementation. This shows, with careful implementation, the throughput of SYN MP_CAPABLE can meet the needs of different types of services. In our real deployment, we use 3 forwarding threads/machine because line-rate processing of packets from 5-tuple seen before is achieved and we believe 4MPPS/(machine·service) throughput for SYN MP_CAPABLE is enough given the maximal number of concurrent MPTCP clients in our setting is $2^{32} \approx 4295\text{M}$ (32 bits token space).

VI. Lessons Learned

We now take a step back and discuss the lessons learned about the desired properties of an LB-friendly multipath L4 protocol based on our experience with *RR*. These lessons fill a critical gap of previous multipath L4 protocol designs regarding their practical deployment issues.

The unique connection identifier (CID) should be generated only at the client side and server side CID should be avoided. One difficulty of *RR* design is how to handle MPTCP server side key-B/token-B, i.e., server side CID. If BE generates the CID, we need a synchronization mechanism between LB and BE. If LB generates the CID, we need a way to inform BE of the selected CID. If it is an inband method, i.e. by manipulating the multipath L4 protocol packet itself like what we do in *RR*, it compromises throughput as shown in § V-B. If the protocol is further authenticated or encrypted, inband methods are barely possible. If it is an outband method, a synchronization mechanism is still needed. This dilemma can be avoided by generating CID purely at the client side. Unlike MPTCP's small token space (32 bits), if the size of CID is large enough (at least 64 bits), the probability of collision is negligible for the number of users on a million scale. Encryption/authentication offered by L4 protocol itself or higher layer protocol can be leveraged to further handle the case of CID collision. Moreover, the distinction of MPTCP key and token further complicates CID generation in multiple LBs case because SHA1 is one-way.

The unique CID should be presented in every packet. Another challenge that makes MPTCP not LB-friendly is that it only contains its CID in its incoming SYN MP_CAPABLE, ACK MP_CAPABLE, and SYN MP_JOIN packets. Thus, if an LB receives a SYN MP_JOIN packet with an unknown token or a non-SYN packet of a secondary subflow from an unknown 5-tuple due to route changes, it cannot even make a good guess based on the 5-tuple that is better than a random guess. As a result, this leads to the design decision of token space division and packet redirection, and requires the LB to reserve a large amount of memory to remember the next hops for all secondary subflows. Nevertheless, all these problems can be easily solved by incorporating CID into every packet, so that we can use CID rather than 5-tuple to do load balancing[4], which can avoid redirection and save memory. In addition, the unique CID is better to be of fixed length at a fixed location. This ensures LB can get the CID directly rather than parsing the packet one field after another like how we parse (MP)TCP options, which also decreases throughput.

VII. Conclusion

In this paper, we have presented our MPTCP-aware LB *RR*, which offers high performance and ease of deployment. We believe it removes a key hurdle in the rapid adoption of MPTCP as a critical technology to exploit multipath connectivity for mobile clients. Although its performance in very large scale deployment still needs to be examined, we foresee that *RR* will be an integral part of ultra-broadband, ultra-reliable, ultra-low delay 5G mobile and fixed access. We also envision the lessons we learned are applicable in designing future multipath protocols and support load balancing of other multipath protocols, e.g., multipath QUIC.

References

[1] Olteanu *et al.*, "Stateless datacenter load-balancing with beamer," in *NSDI*. USENIX, 2018, pp. 125–139.
[2] ——, "Layer 4 Loadbalancing for MPTCP," IETF, Internet-Draft, 2016.
[3] Duchene *et al.*, "Multipath TCP Load Balancing," IETF, Internet-Draft, 2017.
[4] ——, "Making multipath tcp friendlier to load balancers and anycast," in *ICNP*. IEEE, 2017, pp. 1–10.
[5] Paasch *et al.*, "Multipath TCP behind Layer-4 loadbalancers," IETF, Internet-Draft, 2015.
[6] Olteanu *et al.*, "Datacenter scale load balancing for multipath transport," in *HotMiddleboxes*. ACM, 2016, pp. 20–25.
[7] Liénardy *et al.*, "Towards a multipath tcp aware load balancer," in *ANRW*. ACM, 2016, pp. 13–15.
[8] "Haproxy," http://www.haproxy.org/.
[9] Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer." in *NSDI*. USENIX, 2016, pp. 523–535.
[10] Patel *et al.*, "Ananta: Cloud scale load balancing," in *SIGCOMM*. ACM, 2013, pp. 207–218.
[11] Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *SIGCOMM*. ACM, 2017, pp. 15–28.
[12] Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," 2015, pp. 27–38.
[13] Ford *et al.*, "TCP Extensions for Multipath Operation with Multiple Addresses," IETF, RFC 6824, 2013.
[14] Intel, "Data plane development kit," http://dpdk.org/.
[15] "Multipath tcp - linux kernel implementation," https://www.multipath-tcp.org/.
[16] "Cloudlab," https://www.cloudlab.us/.
[17] "Scapy," https://scapy.net/.

[4]However, there may be a backward compatibility issue with TCP.