IP Multihoming Throughput Maximization based on Passive RTT Measurements

Adrian Sterca
Department of Computer Science
Babeş-Bolyai University
Cluj-Napoca, Romania
forest@cs.ubbcluj.ro

Darius Bufnea
Department of Computer Science
Babeş-Bolyai University
Cluj-Napoca, Romania
bufny@cs.ubbcluj.ro

Virginia Niculescu
Department of Computer Science
Babeş-Bolyai University
Cluj-Napoca, Romania
vniculescu@cs.ubbcluj.ro

***Abstract*—We present a routing solution for multihoming sites that maximizes the throughput of local flows. We consider the problem of transferring data between two multihomed network sites (i.e. network sites that have two or more uplinks to the Internet). Our routing solution is deployed at the edges of both multihomed sites and routes local flows dynamically through several outgoing network paths/links depending on the load (i.e. congestion level) on each path. If the load on a network path increases, fewer local flows are routed through it. We measure the load on a network path using passive RTT measurements. We performed a significant number of experiments in order to show that our multihoming solution performs better than an ECMP-based (i.e. Equal-Cost Multipath) solution in terms of total aggregated throughput and inter-flow fairness.**

***Index Terms*—multihoming, multipath load-balancing, multipath routing, ECMP routing**

I. Introduction

Nowadays, multihoming network setups have a strong presence in the industry, but they are beginning to be the status quo for end users, too. It is quite common for a company to have two or more network uplinks to different Internet service providers (ISP). The multihoming networking setup we consider in this paper has two network sites which are connected to each other through the public Internet and each network site is multihomed, having at least two uplink connections to different ISPs. A typical, although simplistic, drawing of our problem setup is depicted in Fig. 1. In this figure, Site A is a local area network (LAN) that is connected to the Internet through two different ISPs, ISP1 and ISP2 and similarly, Site B is a LAN which has two uplink connections, one to ISP3 and another to ISP4. Let's assume there are two physical network paths between Site A and Site B: one going through ISP1 and ISP3 and the other going through ISP2 and ISP4. Let's call the first physical network path *Path0* and the latter *Path1* and we assume the paths are independent (i.e. they do not share any network segment). The edge router from Site A can send packets coming from the local network over *Path0* (i.e. through ISP1) and over *Path1* (i.e. through ISP2). Similarly, Site B can send packets coming from B's local network over *Path0* (i.e. through ISP3) and over *Path1* (i.e. through ISP4). We assume there are a number of TCP connections between Site A and Site B. The goal of this paper is to find a routing policy for packets sent from Site A to Site B over the multipath network (i.e. to route packets either over *Path0* or *Path1*) such that: the total, aggregated throughput from site A to site B is maximized and the throughput differences between flows sent from Site A to Site B over *Path0* and *Path1* is minimized.

We want our routing decisions to be transparent to local computers from site A and site B, so the solution comes in the form of a virtual tunnel interface between Site A and Site B; the tunnel interface is located on the edge routers of Site A and Site B and splits data over *Path0* and *Path1*. In this paper we only deal with the routing algorithm, not the technicalities of the tunneling interface. This routing policy is actually a *flow mapping policy* meaning that rather than individual packets, whole TCP flows are mapped on either *Path0* or *Path1*. This is because, by sending packets from the same TCP flow through different network paths with different capacity/delay properties, it is highly likely that packets get reordered in the network and cause the transmission rate of the TCP flow to be halved, thus reducing the throughput [3].

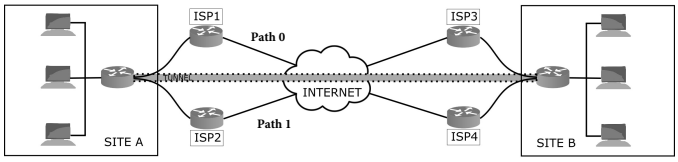

Fig. 1. The network setup of the multihoming problem

The aforementioned goals should be achieved in a context of changing available bandwidth and delay for both network paths, depending on the network load. For instance, if we have N TCP flows going through the tunnel from Site A to Site B and *Path1* and *Path2* have the same capacity of X Mbps, $N/2$ flows should be mapped on *Path1* and $N/2$ flows mapped on *Path2*. If at some point, due to increased load/congestion on *Path1*, the available bandwidth drops to $X/2$ Mbps, we should remap flows on the other path so that *Path1* carries $N/3$ flows and *Path2* carries $2N/3$ flows. This way, a larger aggregated throughput should be achieved by the N flows and also the inter-flow (throughput) fairness of the N flows should be increased.

As practical examples, you can imagine Site A and Site B to be two different buildings of the same company located far apart. Or Site A and Site B can be two data centers belonging

to the same authority. Site A transfers/replicates data to site B through a number of TCP connections and it wants to achieve maximum throughput with the smallest transfer completion time. We will give in the following sections a flow mapping policy that adapts dynamically the number of flows on each physical network path to reflect the load of that path. The load on a path is estimated based on passive RTT (i.e. Round-Trip Time) measurements.

II. Related Work

Our work largely falls in the field of multipath data transfer. This includes traffic engineering (TE), more specifically multipath load-balancing, where the packets from a set of flows need to be forwarded to the destination through a set of multiple network paths). But it also includes Concurrent Multipath Transfer (CMT) where the packets of a single flow need to be transported to the destination over multiple network paths concurrently. We can classify related work in this field according to the level of the OSI network model the solution functions at. For multipath load-balancing we can further classify solutions depending on the granularity level they use when performing the data split on multiple paths: a) flow-level, b) packet-level or c) subflow-level .

At level 2, the Data Link level, there are several solutions that split incoming traffic on multiple paths, most of them being designed for data transfer inside data centers having a Clos or fat-tree topology [1], [2]. All these solution are designed to work inside data centers having a Clos or fat-tree topology, so they would not work (at least not directly) outside the data center network. There are many solutions that perform multipath load-balancing at level 3 of the OSI model, some of them do traffic engineering inside an AS domain network (i.e. inside an ISP network) [4]–[9] and others perform traffic engineering across AS domains for BGP routing [10], [11]. The classical way of performing load balancing across multiple paths is using Equal-Cost Multipath routing (ECMP) [3], a feature supported by the main used intra-domain routing protocols, OSPF and IS-IS. If there are available several network paths with the same cost, this feature maps each packet on a network path depending on a hash function applied to the packet header fields (usually the IP addresses), thus all the packets belonging to the same flow follow the same network path and consequently, ECMP performs load-balancing. This is referred to as *oblivious routing* or *oblivious traffic engineering* because it does not take into account past traffic patterns. Another TE solution in an ISP network is *predicted-based TE* which uses traffic matrices that represent the traffic demand in the ISP network across a large time interval (e.g. months) and uses this estimation to spread flows on multiple network paths [5]–[7]. A third TE solution in an ISP network is *online traffic engineering* exemplified by TeXCP [8]. TeXCP measures the path utilization at each router by actively probing these routers and based on this feedback, it adapts the load on each path.As opposed to TeXCP, our technique does not rely on explicit feedback from routers in the ISP network. It only uses information available at the edge of the network, the customer site. All the above solutions try to minimize the maximum link utilization in the ISP network, while our mechanism strives to improve throughput and delay metrics, but also inter-flow fairness, only for the flows sourced at the multihoming site.

Concurrent Multipath Transfer was also approached at transport-level, either by new transport-level protocols like Multipath TCP [12] or SCTP [13], [14] or by changes to classical TCP [15] All these protocols send a flow on several network paths concurrently achieving a higher throughput at flow level.

III. The multihoming routing algorithm

In order to distribute a set of flows over multihoming links/paths depending on the links' properties (i.e. bandwidth capacity, delay) and on their current network load (i.e. congestion level), our multihoming routing algorithm requires two components:

- the network load estimation policy (estimates the network load on each path)
- the mapping function of local flows on outgoing links.

The second component, i.e. algorithm for mapping flows on the outgoing links, is depicted in Listing 1. The *FlowRemapping* algorithm is executed whenever the network load estimation policy decides that the conditions have changed in the network. The network load is estimated by the estimation policy and converted to weights (i.e. positive numbers normalized to the interval $[0, 1]$) which are assigned to each network path. A weight dictates how many local multihoming flows are mapped/sent on that path. The sum of all the weights equals 1. When entering the algorithm, $Path_i$ has $old\_weight_i \cdot N$ local multihoming flows mapped on it and after the algorithm is executed, $Path_i$ will have $weight_i \cdot N$ local multihoming flows mapped on it, where N is the total number of local multihoming flows passing through the gateway. The first For loop (i.e. lines 2-8) computes all the flows that need to be moved from their current network path (due to a drop in the path's weight) and adds them to the set *R*. The function $SortByRemappingTime(Flows(Path_i))$ sorts the set of flows currently mapped on $Path_i$ descending by the last remapping time and the function $GetFlowsForRemap(Flows(Path_i), flows\_to\_remap)$ removes and returns the set of first $flows\_to\_remap$ flows from the $Flows(Path_i)$ set (i.e. the first $flows\_to\_remap$ flows that were most recently remapped from another path to $Path_i$). The second For loop (i.e. lines 9-16) takes each flow from the *R* set and assigns them to the new path. So, while the first For loop considers paths that lose flows in the next epoch, the second For loop works with the paths that acquire new flows in the next epoch. We considered several alternatives for choosing the flows that should be removed from a network path when that path's weight decreases: 1) random choice of flows, 2) the flows that were most recently remapped on this path (i.e. youngest flows on this path) and 3) the oldest flows on the path. After initial tests performed

with all three alternatives, we went with *2) the youngest flows on this path* which achieved better results in terms of total throughput of multihoming flows.

Algorithm 1 The FlowRemapping algorithm is executed whenever a path's weight has changed:

Input:
$Path_i$: the i-th network path; $i \in [1, m]$
N: the number of local multihoming flows
old_weight_i: the current weight of $Path_i$; $i \in [1, m]$
$weight_i$: the new weight for $Path_i$; $i \in [1, m]$
$Flows(Path_i)$: the set of local multihoming flows currently mapped on $Path_i$

The FlowRemapping algorithm is:

1: $R = \{\}$
2: **for** $i = 1$ **to** m **do**
3: $flows\_to\_remap = \lfloor old\_weight_i \cdot N \rfloor - \lfloor weight_i \cdot N \rfloor$
4: **if** $flows\_to\_remap > 0$ **then**
5: SortByRemappingTime($Flows(Path_i)$)
6: $R = R$+GetFlowsForRemap($Flows(Path_i)$, $flows\_to\_remap$)
7: **end if**
8: **end for**
9: **for** $i = 1$ **to** m **do**
10: $flows\_to\_remap = \lfloor weight_i \cdot N \rfloor - \lfloor old\_weight_i \cdot N \rfloor$
11: **if** $flows\_to\_remap > 0$ **then**
12: **for** $flow$ in GetFlowsForRemap(R, $flows\_to\_remap$) **do**
13: $flow.path = i$ // assign $flow$ to $Path_i$
14: **end for**
15: **end if**
16: **end for**

The first component, i.e. the network load estimation policy, measures the load using passive RTT measurements. If we measure the RTT of a set of TCP flows passing through the same network path in normal conditions we would see that the array of these RTT samples will have an oscillating shape; the RTT metric will increase until it reaches a maximum value and than start decreasing until it reaches a minimum value and then start increasing again and so on. This corresponds to the TCP - router's queue operating regime: TCP increases congestion window, queue forms in the router, thus the RTT increases, then queue overflows, TCP decreases the congestion window, thus RTT decreases and the cycle restarts. The intuition behind the RTT-based network load estimation policy is that as the network gets significantly more congested, the average RTT measured by flows should experience a constant and consistent increase. We want to filter out these cycles from the RTT samples array so we pass this array through a two-stages smoothing process: 1) first a mixed equal+exponential weighted average on windows of 16 RTT samples in order to remove tiny-scale fluctuations and reduce the amplitude of the fluctuations and then, 2) we divide the RTT array into cycles and compute the average of RTT values in a cycle to remove small-scale fluctuations.

For the first smoothing stage we take groups of 16 consecutive RTT samples and apply a weighted average on them. The most recent 8 RTT samples have the weight 1 and then, the weights start decreasing exponentially giving less weight on older samples. This way, the RTT values are smoothed, but the most recent RTT samples have a larger contribution in this average. The weights used are the following [16]:

$$w_i = 1 \qquad \text{for } i = \overline{0,7}$$

$$w_i = 1 - \frac{i + 1 - mid}{mid + 1} \qquad \text{for } mid = 8 \text{ and } i = \overline{8,15}$$

After the first smoothing function is applied, ideally, the RTT array only contains small-scale fluctuations and possibly large-scale fluctuations. Because we do not want to perform flow remapping too often (since moving a flow from one link to another usually implies packet reorderings for this flow and thus, TCP throughput drop), we filter out small-scale fluctuations by considering an average value for a *RTT cycle*. The effect of the first and second smoothing function applied on measured RTT samples obtained through simulations is visible in Fig. 2. The line labeled 'RTT samples' presents real RTT measurements taken from a set of flows passing through a network path that has a low load/congestion level between seconds 20-50 and 160-300 and becomes severely congested between seconds 50-160 (a significant number of new flows enters the network). The line labeled 'RTT averaged over a 16-window' is the result of applying the first smoothing function on the RTT samples string. We can see this line is smoother than the line of raw RTT samples. Finally, the red line labeled 'RTT cycle average' shows the average points of each RTT cycle connected by a line. We can see that this line remains relatively constant in each of the two periods, low congestion in seconds 20-50 and 160-300 and, respectively, high congestion in seconds 50-160, while the value of this average remains consistently higher in the period of high congestion compared to that of the low congestion period.

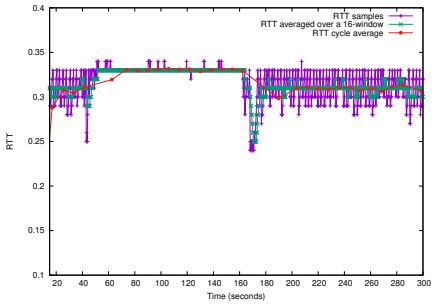

Fig. 2. The two-stages smoothing performed on RTT samples

The *UpdateRTTState* algorithm depicted in listing 2 is the RTT-based network load estimation policy. It is executed whenever a new return packet (i.e. TCP acknowledgment packet) arrives at the multihoming sender router. The algorithm updates the RTT state of the respective network path and when the state changes significantly, it computes new weights for each network path and calls the *FlowRemapping* algorithm from listing 1 to perform flow remapping. The current RTT sample for this packet is computed in line 1, by subtracting the TS Echo Reply field of the Timestamps Option in the TCP header from the current time (i.e. now). After that, it updates the minimum and maximum RTT in lines 3-8. Line 2 computes the $srtt$ (i.e. exponentially smoothed RTT) which is used in line 9 for deciding to consider or not this RTT sample

in the RTT state. When computing the RTT cycles and path weights (i.e. lines 9-24) we use only one RTT measurement per $srtt$ in order to reduce computations. The 16 window equal+exponential average (i.e. first smoothing function) is computed first in line 10 by function *UpdateRTTWindow*. Then, if the function *UpdateRTTCycle* detects the start of a new cycle, we compute the new weights for each network path in lines 14-21 and then call the *FlowRemapping* algorithm to perform flow remapping. The weight for a path specifies how many flows should be mapped on this path and is used in the algorithm *FlowRemapping*. The weight of a path is computed as an inverse linear mapping of *cycle.average_rtt* from the interval [*min_cycle_avgrtt*, *max_rtt*] to the interval [0, 1], where *cycle.average_rtt* is the average RTT for the current cycle, *min_cycle_avgrtt* is the minimum *cycle.average_rtt* recorded out of all RTT cycles and *max_rtt* is the maximum RTT ever recorded (for that specific network path). The justification for this formula is omitted here due to space constraints and the reader is referred to the longer version of this paper [16].

The *UpdateRTTCycle* algorithm checks if a new *RTT cycle* has completed and computes the average RTT value per *RTT cycle*. It is omitted here due to space constraints, but the reader can find the complete algorithm in [16]. A *RTT cycle* is just a sequence of RTT values from the RTT samples array that include an ascending phase and a descending phase; or in case of persistent congestion, a *RTT cycle* is a sequence of RTT samples with similar values, but the length of the sequence is above a threshold.

Algorithm 2 The RTT-based network load estimation policy

Input:
p : an ACK packet received on path $Path_k$
m : the number of network paths
min_rtt_k : minimum RTT value ever recorded for $Path_k$
max_rtt_k : maximum RTT value ever recorded for $Path_k$
$srtt_k$: smoothed RTT for $Path_k$
$last\_rtt\_update_k$: last time the RTT state was updated for $Path_k$
now : the current time

The UpdateRTTState algorithm is:

1: $curr\_rtt = now - p.TSecr$
2: $srtt_k = 0.8 \cdot srtt_k + 0.2 \cdot curr\_rtt$
3: **if** $curr\_rtt < min\_rtt_k$ **then**
4: $min\_rtt_k = curr\_rtt$
5: **end if**
6: **if** $curr\_rtt > max\_rtt_k$ **then**
7: $max\_rtt_k = curr\_rtt$
8: **end if**
9: **if** $last\_rtt\_update_k < (now - srtt_k)$ **then**
10: avg_rtt_window = UpdateRTTWindow($Path_k$, $curr\_rtt$)
11: $last\_rtt\_update_k = now$
12: **if** (UpdateRTTCycle($Path_k, avg\_rtt\_window$) = 1) **then**
13: { compute the weight for each Path }
14: $sum = 0$
15: **for** $i = 1$ **to** m **do**
16: $weight_i = \frac{cycle.average\_rtt_i - min\_cycle\_avgrtt_i}{max\_rtt_i - min\_cycle\_avgrtt_i}$
17: $sum = sum + weight_i$
18: **end for**
19: **for** $i = 1$ **to** m **do**
20: $weight_i = 1 - weight_i$ / sum
21: **end for**
22: FlowRemapping()
23: **end if**
24: **end if**

IV. Evaluation

This section presents a subset of the experiments we have performed in order to validate our multihoming routing algorithm. We can not present all the experiments here due to space constraints, but all the details of the experiments and additional experiments not shown here are given in the longer version of this paper [16]. We implemented our bandwidth aggregation mechanism as a multihoming router in the ns-3 network simulator.

The network setup of our experiments is presented in Fig. 3. The multihoming local network is behind router R_1 and is formed by the source nodes: s_1 .. s_n. The multihoming receiver network is behind router R_4 and is formed by the destination nodes: d_1 .. d_n. We have one multihoming TCP flow between each (s_i, d_i) node pair. Router R_1 is a multihoming sender router that splits incoming multihoming flows on the two outgoing network paths: *Path0*=$R_1 - R_2 - R_4$ and *Path1*=$R_1 - R_3 - R_4$. Router R_4 is a multihoming receiver router that maps reverse TCP packets (i.e. ACK packets) on the same link/path the original data packets came through. The capacity of the access links of source and destination nodes is always 1 Gbps and the transmission delay is randomly distributed between 1 ms and 10 ms. The transmission delay of the all inter-router links is always set to 40 ms, except the transmission delay of link $R_2 - R_4$ which changes across experiments. Similarly, during an experiment, the network capacities of the inter-router links $R_1 - R_2$, $R_1 - R_3$ and $R_3 - R_4$ are always equal, but the link $R_2 - R_4$ can have, depending on the experiment, a different network capacity. The router queue is always set to the bandwidth-delay product for that link, for all routers. We have used two queue drop policies for routers in our experiments: DropTail queuing and Random Early Detection (RED). We have 64 TCP flows originating in the multihoming local network (i.e. source nodes $s_i, i = 1..64$) and going to the destination nodes $d_i, i = 1..64$. These flows start in the beginning of the simulation at random times to remove phase effects and last until the simulation completes. Each simulation lasts 600 seconds. We have chosen this duration for a simulation so that a simulation lasts long enough for us to observe a steady-state behavior. Additional 512 TCP flows attached to source nodes connected to the R_2 router and destination nodes connected to the R_4 router (these nodes are not depicted in Fig. 3) add network load on *Path0*. 64 of these flows start in the beginning of the simulation and last until the end of the simulation creating a steady-state load on *Path0*. The remaining 448 flows start at random times between seconds 40-50 of the simulation and they finish at random times between seconds 320 and 400 of the simulation. These additional 448 TCP flows create an increased load on *Path0* between seconds 40 and 400 of the simulation, thus forcing our multihoming sender router R_1 to

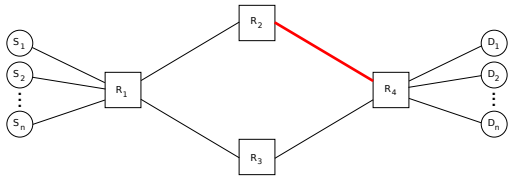

Fig. 3. The network setup used in the experiments

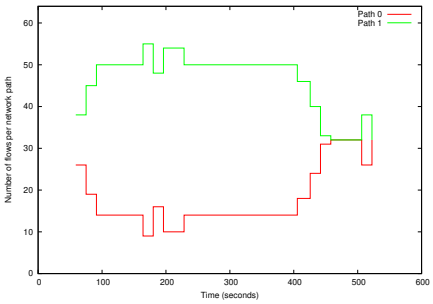

Fig. 4. The number of multihoming flows mapped on each path by the multihoming routing algorithm; DropTail queuing and 100Mbps capacity

send more flows on the other network path, *Path1*. Similarly, 64 TCP flows attached to source nodes connected to the R_3 router and destination nodes connected to the R_4 router (these nodes are also not depicted in Fig. 3) create a steady-state load on *Path1*, for the duration of the entire simulation. In addition, there are 32 TCP flows on the reverse link $R_4 - R_2$ and other 32 TCP flows on the reverse link $R_4 - R_3$ for an increased network dynamics. For the TCP flows used in our simulation, either the multihoming flows or the load flows, we used a mixture of TCP Linux Cubic, Sack and NewReno.

We compared our multihoming routing algorithm that maps multihoming flows on the two outgoing paths dynamically with an ECMP (Equal-Cost Multipath routing [3]) routing solution that splits multihoming flows equally between the two outgoing network paths. We will use two metrics for this comparison:

- AVGT(*Average throughput per flow*) = the average flow throughput of the 64 multihoming flows
- STD(*Standard deviation of the flow throughput values*) = the standard deviation of the 64 throughput values

The flow throughput used in the above metrics is the throughput computed for each multihoming flow during the increase load period of the simulation (i.e. seconds 40-400).

First, we considered three diverse network capacities of 100 Mbps, 500Mbps and 1Gbps and a 40 ms transmission delay for each inter-router link. The queuing policy at routers was either DropTail or RED. For each (network capacity - queuing policy) combination we ran 2 experiments: one when an ECMP routing mechanism was used at router R_1 and other when our multihoming routing algorithm was used for router R_1. Each experiment consisted of a simulation being run 10 times with different, randomly generated, flow starting and ending times (for all TCP flows) and access links delays. In the end, we computed for each experiment an average of the aforementioned metrics across all 10 simulations performed

TABLE I
DROPTAIL QUEUING; SYMMETRICAL BANDWIDTH, DELAY

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	STD: 44018.7 AVGT: 61591	STD: 371675 AVGT: 435682	STD: 836406 AVGT: 974754
RTT-based mapping (%)	STD: 34.02 AVGT: 26.53	STD: 31.73 AVGT: 13.86	STD: 44.07 AVGT: 10.28

TABLE II
RED QUEUING; SYMMETRICAL BANDWIDTH, DELAY

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	STD: 45748.9 AVGT: 62136	STD: 348482 AVGT: 430274	STD: 818536 AVGT: 1004900
RTT-based mapping (%)	STD: 36.34 AVGT: 21.04	STD: 31.74 AVGT: 9.96	STD: 40.94 AVGT: 5.57

for the same experiment. The obtained results are depicted in Table I for the DropTail queuing discipline and, respectively, in Table II for the RED queuing discipline. For ECMP routing we show the absolute values for the two metrics used, but for our multihoming routing algorithm (i.e. RTT-based mapping) we show percentage improvement values for the metrics with respect to the corresponding metric used in ECMP mapping. We can see that our multihoming routing algorithm improved all three metrics with respect to ECMP routing, in all tested network capacities and queuing disciplines. The throughput gain when using our multihoming routing algorithm is between 5% and 27%, while the STD gains are much higher, sometimes more than 68%. In Fig. 4 we can see that during the increased load period our multihoming routing solution mapped more local flows on *Path1* than on *Path0*.

In the next phase, we tried to see whether an asymmetric RTT on the two network paths would influence our results. We performed the same experiment as before, but this time, in all simulations, the transmission delay of link $R_2 - R_4$ was 80ms, while the transmission delay of all other links remained unchanged to 40ms. This led to a RTT on *Path0* that was more than 1.5 times the RTT on *Path1*.As usual, for each (network capacity - queuing policy) combination we ran 3 experiments: one where ECMP routing was used at router R_1 and other when our multihoming routing was used for router R_1; one experiment consists of 10 simulations. The obtained results are depicted in Tables III and IV for the DropTail and RED queue policy, respectively. We can see here the same improvements for all three metrics when the multihoming routing algorithm was employed at router R_1, similar to what we have seen in the symmetrical RTT-bandwidth experiments (i.e. Tables I and II). Although, the $AVGT$ improvements of the multihoming routing algorithm are now smaller than the improvements obtained for the symmetrical RTT-bandwidth experiments; this is especially true for 1Gbps, RED queuing.

Then we tried to see whether our mechanism works on a setup with asymmetric network capacity paths. We performed the same experiment as before, but this time, in all simulations,

TABLE III
DROPTAIL QUEUING; ASYMMETRICAL DELAY

Network capacity			
	100Mbps	500Mbps	1Gbps
ECMP mapping	STD: 48457.2 AVGT: 69607	STD: 463800 AVGT: 503196	STD: 1272350 AVGT: 1267670
RTT-based mapping (%)	STD: 38.37 AVGT: 22.98	STD: 35.29 AVGT: 15.13	STD: 39.77 AVGT: 7.00

TABLE IV
RED QUEUING; ASYMMETRICAL DELAY

Network capacity			
	100Mbps	500Mbps	1Gbps
ECMP mapping	STD: 42211.7 AVGT: 64897	STD: 340308 AVGT: 458444	STD: 848286 AVGT: 1092360
RTT-based mapping (%)	STD: 33.24 AVGT: 15.56	STD: 37.06 AVGT: 4.79	STD: 39.75 AVGT: 2.61

the network capacity on all links from *Path0* were double the network capacity of links from *Path1*. The obtained results are depicted in Tables V and VI for the DropTail and RED queue policy, respectively. Please note that for these asymmetrical network capacity experiments, we had to slightly modify the RTT-based mapping algorithm (i.e. the *UpdateRTTState* algorithm depicted in listing 2) so that after the weights for both network paths are computed we further scaled these weights as following: we scaled the weight of *Path0* by 66% and scaled the weight of *Path1* by 33% (because the network capacity of *Path0* is double the capacity of *Path1*). At the same time, in order to facilitate fair competition we modified the ECMP mapping for these experiments so that the ECMP multihoming router R_1 always maps 66% of the multihoming flows on *Path0* and 33% of the flows on *Path1*.

TABLE V
DROPTAIL QUEUING; ASYMMETRICAL BANDWIDTH

Network capacity			
	200Mbps/ 100Mbps	250Mbps/ 500Mbps	500Mbps/ 1Gbps
ECMP mapping	STD: 48448.3 AVGT: 69483	STD: 167787 AVGT: 196564	STD: 436776 AVGT: 480950
RTT-based mapping (%)	STD: 22.49 AVGT: 23.57	STD: 29.18 AVGT: 20.09	STD: 38.32 AVGT: 11.99

TABLE VI
RED QUEUING; ASYMMETRICAL BANDWIDTH

Network capacity			
	200Mbps/ 100Mbps	250Mbps/ 500Mbps	500Mbps/ 1Gbps
ECMP mapping	STD: 41387.5 AVGT: 63080	STD: 134089 AVGT: 183640	STD: 327732 AVGT: 425396
RTT-based mapping (%)	STD: 24.16 AVGT: 17.33	STD: 24.33 AVGT: 11.82	STD: 29.30 AVGT: 8.93

V. CONCLUSIONS AND FUTURE WORK

We have presented in the previous sections a multihoming routing solution for throughput maximization. Our solution comes in the form of a virtual tunnel that connects two sites through multiple independent or quasi-independent network paths. Our routing solution maps local multihoming flows on the possible outgoing network paths so that these flows use a larger aggregated available bandwidth in changing network conditions. The routing solution dynamically adapts the flow mappings on the outgoing network paths so that a path with a higher load receives fewer local multihoming flows than a network path with a light load. We estimated the load on a network path using passive RTT measurements. We applied a 2-step smoothing process on the RTT sample array and then used these values to compute paths' weights. We have tested our bandwidth aggregation mechanism in a simulated network and showed that it performs better than ECMP routing in terms of total aggregated throughput and fairness between multihoming flows.

REFERENCES

[1] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, *CONGA: Distributed Congestion-aware Load Balancing for Datacenters*, ACM Conference on SIGCOMM, 2014, pp.503-514.

[2] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, *Presto: Edge-based Load Balancing for Fast Datacenter Networks*, 2015 ACM Conference on SIGCOMM, New York, USA, 2015, pp.465-478.

[3] D. Thaler, C. Hopps, *Multipath Issues in Unicast and Multicast Next-Hop Selection*, RFC 2991, IETF, November 2000.

[4] P. Merindol, J.J. Pansiot, S. Cateloin, *Improving Load Balancing with Multipath Routing*, 17th International Conference on Computer Communications and Networks, Virgin Islands, USA, 2008, pp.54-61.

[5] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg, *COPE: traffic engineering in dynamic networks*, 2006 ACM Conference on SIGCOMM, New York, USA, 2006, pp.99-110.

[6] D. Applegate and E. Cohen, *Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs*, 2003 ACM Conference on SIGCOMM, New York, USA, 2003, pp.313-324.

[7] B. Fortz, J. Rexford, and M. Thorup, *Traffic engineering with traditional IP routing protocols*, IEEE Communications Magazine, Vol. 40, Issue 10, pp.118-124, October, 2002.

[8] S. Kandula, D. Katabi, B. Davie, and A. Charny, *Walking the tightrope: responsive yet stable traffic engineering*, 2005 ACM Conference on SIGCOMM, New York, USA, 2005, pp.253-264.

[9] E. Keller, M. Schapira, and J. Rexford, *Rehoming edge links for better traffic engineering*, SIGCOMM Computer Communications Review, Vol. 42, Issue 2, pp.65-71, March, 2012.

[10] J. Wu, C. Yuen, B. Cheng, Y. Shang, and J. Chen, *Goodput-Aware Load Distribution for Real-time Traffic over Multipath Networks*, IEEE Transactions on Parallel and Distributed Systems, Vol. 26 , Issue 8, pp. 2286-2299, August, 2015.

[11] Y. Li, Y. Zhang, L. L. Qiu, and S. Lam, *SmartTunnel: Achieving Reliability in the Internet*, 2007 IEEE Conference on Computer Communications, Washington, USA, 2007, pp.830-838.

[12] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, *Improving datacenter performance and robustness with multipath TCP*, 2011 ACM Conference on SIGCOMM, New York, USA, 2011, pp.266-277.

[13] J. R. Iyengar, P. D. Amer, and R. Stewart, *Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths*, IEEE/ACM Transactions on Networking, Vol. 14, Issue 5, pp.951-964, October, 2006.

[14] W. Yang, H. Li, F. Li, Q. Wu, and J. Wu, *RPS: range-based path selection method for concurrent multipath transfer*, 6th International Wireless Communications and Mobile Computing Conference, New York, USA, 2010, pp.944-948.

[15] J. Wang, J. Liao, and T. Li, *OSIA: Out-of-order Scheduling for In-order Arriving in concurrent multi-path transfer*, Journal of Network and Computer Applications, Vol. 35, Issue 2, pp.633-643, March, 2012.

[16] A. Sterca, D. Bufnea, V. Niculescu, *Bandwidth Aggregation over Multihoming Links*, Technical Report, http://www.cs.ubbcluj.ro/forest/research/papers/ip-multihoming-techrep.pdf, 2019.