Bandwidth Aggregation over Multihoming Links

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 introduce in this paper a bandwidth aggregation routing solution for multihoming sites. Our routing solution interconnects two distinct multihomed network sites (i.e. network sites that have two or more uplinks to the Internet) and routes local flows between these two network sites. It routes local flows dynamically through several outgoing network paths/links depending on the load (i.e. congestion level) on each path. If a network path/uplink becomes more congested, fewer local flows are routed through it. We detail two path load estimation strategies: one based on RTT measurements and the other based on throughput measurements, both implying passive network measurements. Our multihoming solution outperforms the 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 and Problem Formulation

We consider in this paper a network setup like the one in Fig. 1 with two multihomed sites, Site A and Site B (i.e. both network sites have 2 uplink connections to the Internet through different ISPs). We assume there are two physical network paths between Site A and Site B: *Path1* going through ISP1 and ISP3 and *Path2* going through ISP2 and ISP4. We assume the paths are independent or quasi-independent (i.e. the paths may share common network segments, but have different bottleneck links). 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 *Path1* or *Path2*) such that: **a)** the total, aggregated throughput from site A to site B is maximized; **b)** maintains high inter-flow fairness for flows going from Site A to Site B; **c)** this routing policy works with existing Internet technology (i.e. it does not rely on specialized feedback/actions from core routers) **d)** routing local flows over *Path1* or *Path2* is transparent to Site A and Site B local computers.

Because of the last condition, our solution comes in the form of a virtual tunnel interface between Site A and Site B (depicted in Fig. 1). This tunnel is just a virtual network path over the two physical networks paths, *Path1* and *Path2* (between Site A and Site B). The virtual tunnel interface makes this routing decision transparent to local computers from site A, and respectively site B. This routing policy is actually a *flow mapping policy* meaning that rather than individual packets, whole TCP flows are mapped on either *Path1* or *Path2*. 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 inside the network and they cause the transmission rate of the TCP flow to be halved, thus reducing the throughput [4].

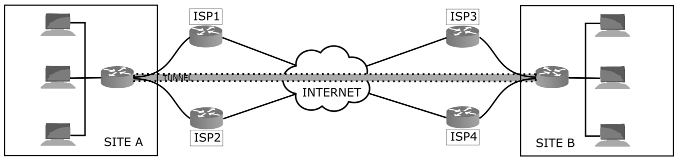


Fig. 1: The typical network setup of the multihoming problem

The aforementioned goals should be achieved in a context of changing network conditions for *Path1* and *Path2*. In other words, if *Path1* has a higher load than *Path 2* (i.e. is more congested), the tunnel interface should map more local flows on *Path 2* than on *Path 1.* As practical examples, you can imagine Site A and Site B to be two different buildings of the same company located far apart. Users from Site A sending data to Site B want maximum throughput and inter-flow fairness. 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 on that path. The load on a path is estimated based on passive RTT (i.e. Round-Trip Time) or throughput 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 according to the level of the OSI network model the solution functions at. Also multipath load-balancing techniques can split data on multiple paths at flow, packet or subflow (i.e. part of flow, flowlet) 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]–[3]. They are implemented either in the hardware of the switch or in virtual switches of the hypervisor split flow cells on switch links depending on the utilization of the link These solutions are designed to work inside data centers having a Clos or fat-tree topology, so they would not work 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) [5]–[7] and others perform traffic engineering across AS domains for BGP routing [8], [9]. Classical load balancing across multiple paths is performed with Equal-Cost Multipath routing (ECMP) [4], 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 based on a hash function applied to the packet header fields (i.e. IP addresses), thus all the packets belonging to the same flow follow the same network path. This is *oblivious traffic engineering* because it does not take into account past traffic patterns. *Predicted-based TE* 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]. Because the traffic matrices are evaluated over long periods of time, these techniques do not cope with fast changes in traffic, e.g. due to diurnal variations. TeXCP [6] performs *online traffic engineering* in an ISP network, as it 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 and works across ISP networks. 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 (across ISP networks). SmartTunnels [9] and Galton [8] are complex tunneling architectures that employ buffers at the sender and receiver for scheduling and reordering packets, perform multipath load balancing, FEC coding. Contrary to these tunnels, our technique is passive and does not use scheduling/reordering buffers, thus eliminating the transport delay incurred by these buffers. Concurrent Multipath Transfer was also approached at transport-level, either by new transport-level protocols like Multipath TCP [10] or SCTP [11], [13] or by changes to classical TCP [12], [14]. All these protocols send a flow on several network paths concurrently achieving a higher throughput at flow level. But exactly for this reason they are not fair to TCP (i.e. a MPTCP flow contains a subflow for each network path and a subflow is equivalent to a single TCP connection). Various mechanisms of splitting a data transfer across multiple paths were also tried at the application-level [15]–[17] and in the context of SDN (Software Defined Networking) [18]–[20].

III. The Bandwidth Aggregation Mechanism for Multihoming Links

The goal of our bandwidth aggregation mechanism is to distribute a set of flows over a number of multihoming links/paths depending on the links' properties (i.e. bandwidth and delay) and on their current network load (i.e. congestion level). If at some point, the load increases on a particular network path, a subset of local multihoming flows assigned to this path should move on the other multihoming paths. In the following sections we will deal with the general problem where we have m network paths (not just 2) connecting Site A and Site B, labeled *Path1*, *Path2*, ... *Pathm*. Our solution requires two components:

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

The second component (i.e. flow mapping function) 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. This algorithm just moves flows from one path to another depending on the old weight and the new weight on each path. If $old\_weight > new\_weight$ for a path, then $(old\_weight - new\_weight) \cdot N$ flows are removed from this path (where N is the total number of multihoming flows). Otherwise, $(new\_weight - old\_weight) \cdot N$ flows are added to this paths from other paths. The actual algorithm is presented in the longer version of this paper [22] and omitted here due to space constraints. The following 2 sections outline two network load estimation policies.

A. The RTT-based policy for estimating the network load

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. Figure 2 shows the three typical types of RTT fluctuations encountered by a set of TCP flows passing through the same network path:

- tiny-scale fluctuations - caused by other flows sharing a segment of the same network path that disrupt the RTT measurements of the monitored flows
- small-scale fluctuations - caused by the typical operation of TCP flows (i.e.TCP flows probe for more available bandwidth, when they overshoot the network capacity, the queue overflows causing TCP flows to drop throughput; this causes periodic cycles in the measured RTT; this type of RTT fluctuations are used by TCP Vegas and GCC for WebRTC [21] to adjust the congestion window)
- large-scale fluctuations - determined by a significant network load change; caused by a significant set of flows entering or leaving the network.

We want our RTT-based network load metric to be sensitive only to the last type of fluctuations and ignore the first two types. In order to do this, we pass the RTT samples 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. The values of the weights are: 1, 1, 1, 1, 1, 1, 1, 1, 0.88, 0.77, 0.66, 0.55, 0.44, 0.33, 0.22, 0.11. This way, the RTT values are smoothed, but the most recent RTT samples have a larger contribution in this average.

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 and thus, TCP throughput drops), we filter out small-scale fluctuations by considering an average value for a **RTT cycle**. In order to define what a **RTT cycle** is, we need to first introduce additional concepts. We call a subsequence (RTT_i, RTT_{i+1}) made of 2 RTT samples, *quasi-constant* if $RTT_{i+1} \in [RTT_i - thresh, RTT_i + thresh]$ where $thresh$ is a positive threshold. Similarly, this subsequence is called *ascending* if $RTT_{i+1} > RTT_i + thresh$ and is called *descending* if $RTT_{i+1} < RTT_i - thresh$. After applying the first smoothing function, the RTT sequence will contain segments (i.e. subsequences) of the following types:

1) *ascending segment* - is the largest continuous sequence of RTT samples containing only ascending and quasi-constant subsequences and the longest quasi-constant subsequence from this segment has a length not larger than $stable\_run\_thresh$
2) *descending segment* - is the largest continuous sequence of RTT samples containing only descending and quasi-constant subsequences and the longest quasi-constant subsequence from this segment has a length not larger than $stable\_run\_thresh$
3) *stable-run segment* - is a continuous sequence of RTT samples with the property that any two samples are quasi-constant and the length of the segment is larger than $stable\_run\_thresh > 0$.

We define two types of **RTT cycles** and these can be seen in Fig. 2. First, an **RTT cycle** is a continuous sequence of RTT samples that consists of an ascending segment and a descending segment (not necessarily in this order). Secondly, an **RTT cycle** can also be a continuous sequence of RTT samples that end with a stable-run segment. This second type of cycle can contain an ascending segment or a descending segment preceding the stable-run or it can contain just the stable-run segment as you can see in Fig. 2 (i.e. the time interval marked with ‘heavy load‘). If we consider only the first type of RTT cycles, when the network becomes heavy loaded as seen in Fig. 2, the RTT cycle would end only after the heavy load period had passed and thus, a flow remapping will occur too late - since flow remappings happen only at the end of a RTT cycle.

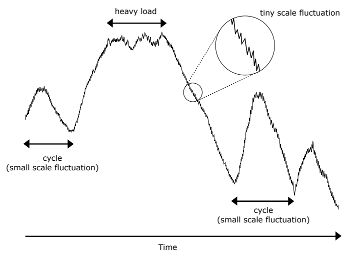


Fig. 2: Typical RTT fluctuations of flows (idealized drawing)

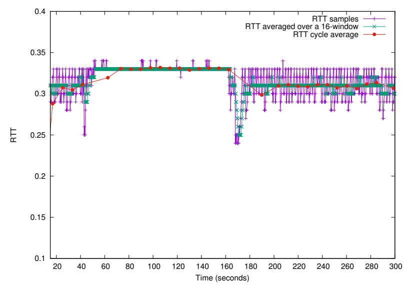


Fig. 3: The two-stages smoothing performed on RTT samples

We further smooth out the RTT samples string by averaging values over an RTT cycle (using the algorithm in listing 2), so that the number of flow remappings will be reduced. The effect of the first and the second smoothing function applied on measured RTT samples obtained through simulations is visible in Fig. 3. The line labeled 'RTT samples' presents raw RTT measurements taken from a set of flows passing through a network path that has a low load between seconds 20-50 and 160-300 and becomes severely congested between seconds 50-160 (when a significant number of new flows enter the network). The green line shows the result of applying the first smoothing function. We can see that this line is smoother than the raw RTT samples line. Finally, the red line shows the average points of each RTT cycle connected by a line. This line remains relatively constant in each of the two periods, low congestion (sec. 20-50, 160-300) and, respectively, high congestion (sec. 50-160), while the value of this average remains consistently higher during high congestion compared to that of the low congestion period.

The *UpdateRTTState* algorithm depicted in listing 1 is the RTT-based network load estimation policy. It is executed whenever a new return packet (i.e. TCP ACK 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 to perform flow remapping. Line 1 computes the RTT sample from the packet by subtracting the TS Echo Reply field of the Timestamps Option in the TCP header from the current time (i.e. now). Then it computes $srtt$ (i.e. exponentially smoothed RTT) and updates the minimum and maximum RTT in lines 2-3. When computing the RTT cycles and path weights (i.e. lines 5-18) 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 performed in line 5 by function

UpdateRTTWindow. Then, if *UpdateRTTCycle* (described in Listing 2) detects the start of a new cycle, we compute the new weights for each network path in lines 9-16 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 *UpdateRTTCycle* algorithm depicted in listing 2 is responsible for managing RTT cycles. This algorithm gets called from the *UpdateRTTState* algorithm approximately once per $srtt_k$ for each network path $Path_k$, when a new acknowledgment packet arrives at the multihoming sender router on path $Path_k$. The state maintained for a RTT *cycle* is the following (one cycle state is maintained for each network path):

cycle.duration: length in time of the previous, completed cycle
cycle.average_rtt: average value of the previous, completed cycle
cycle.start_time: starting time of the current cycle
cycle.current_rtt: current sample value from the current cycle (i.e. average over 16 RTT samples)
cycle.ascending: state of the ascending phase in the current cycle: notstarted/started/ended
cycle.descending: state of the descending phase in the current cycle: notstarted/started/ended
cycle.stableRun_starttime: starting time of the last quasi-constant subsequence in the current cycle
cycle.stableRun_startvalue: starting value of the last quasi-constant subsequence in the current cycle
cycle.cumulative_rtt: sum of all samples from the current cycle
cycle.n: number of samples in the current cycle

Please note that while a *cycle* is made of a sequence of samples, each sample value from a *cycle* is actually an average over an 16 RTT samples window. The *isAscending* condition tests whether the next RTT window average, *avg_rtt_window* and the current sample value of the cycle, *cycle.current_rtt*, form an ascending subsequence. *CTHRESH* is the changing threshold from the definition and we used a value of 5% in the evaluation tests. Similarly, the *isDescending* condition is true when *avg_rtt_window* and *cycle.current_rtt* form a descending subsequence. If the *avg_rtt_window* and *cycle.current_rtt* are quasi-constant, we check in lines 3-7 whether we have a stable-run segment and if this is true, we close the current cycle and initialize a new cycle. In our evaluation tests, we considered that a quasi-constant sequence is a stable-run segment if its length in time is larger than *MIN_REMAPPING_TIME_INTERVAL* (i.e. $stable\_run\_thresh$ from the definition) and it lasts more than half the length of the previous cycle. If *isAscending* is true we do the following things in lines 13-19: we check if this

Algorithm 1 The RTT-based network load estimation policy

Input:
p : an ACK packet received on path $Path_k$
$min\_rtt_k, max\_rtt_k, srtt_k$: minimum RTT, maximum RTT and smoothed RTT for $Path_k$
$last\_rtt\_update_k$: last time the RTT state was updated for $Path_k$

The UpdateRTTState algorithm is:

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

cycle already contains an ascending segment (*cycle.ascending = ended*) which means we should close this cycle and initialize a new one; we check if we were previously on a descending segment (*cycle.descending = started*) and we end this segment; we set *cycle.ascending = started* to signalize we are on an ascending segment. Similar things happen in lines 21-27 if *isDescending* is true. In the end of the algorithm, if we do not have a new cycle, we add the window average RTT to the *cumulative_rtt* of this cycle and increase the number of samples in lines 30-31. The initialization of a new cycle is performed in lines 33-42.

B. The throughput-based policy for estimating the network load

The throughput-based policy is very similar to the RTT-based policy. It estimates the current load of a network path by computing the total throughput of all multihoming flows going through that path. If the total throughput of multihoming flows traversing a network path increases consistently, it means that more bandwidth was available on that network path, so the load on that path has reduced. Similar algorithms like *UpdateRTTState* and *UpdateRTTCycle* are used, the most significant difference being the way path weights are computed. The weight for $Path_k$ is computed as: $weight_k = \frac{cycle.average\_throughput_k}{n_k} / sum$ where $cycle.average\_throughput_k$ is the average throughput of the current cycle for $Path_k$, n_k is the number of local multihoming flows currently assigned to $Path_k$ and sum is the sum of all weights (for normalization to [0,1] purposes). The actual algorithms are detailed in the longer version of this paper [22].

Algorithm 2 Updates the RTT cycle state for $Path_k$

Input:
avg_rtt_window : average of 16 RTT samples for $Path_k$
$cycle$: the data structure for the current RTT cycle of $Path_k$
Returns: True if a new cycle starts or False otherwise

The UpdateRTTCycle algorithm is:

```
1:  isAscending = $avg\_rtt\_window > cycle.current\_rtt \cdot (1 + CTHRESH)$;
2:  isDescending = $avg\_rtt\_window < cycle.current\_rtt \cdot (1 - CTHRESH)$;
3:  if (!isAscending and !isDescending) then
4:      { We are in a stable-run phase }
5:      if ($now - cycle.stableRun\_starttime > cycle.duration/2$)
        and ($now - cycle.stableRun\_starttime > MIN\_REMAPPING\_TIME\_INTERVAL$) then
6:          return newcycle_init()
7:      end if
8:  else {This is not a stable-run phase}
9:      $cycle.current\_rtt = avg\_rtt\_window$
10:     $cycle.stableRun\_starttime = now$
11:     $cycle.stableRun\_startvalue = avg\_rtt\_window$
12:     if isAscending=TRUE then {in ascending phase}
13:         if $cycle.ascending = ended$ then
14:             return newcycle_init()
15:         end if
16:         if $cycle.descending = started$ then
17:             $cycle.descending = ended$
18:         end if
19:         $cycle.ascending = started$
20:     else if isDescending=TRUE then {descending phase}
21:         if $cycle.descending = ended$ then
22:             return newcycle_init()
23:         end if
24:         if $cycle.ascending = started$ then
25:             $cycle.ascending = ended$
26:         end if
27:         $cycle.descending = started$
28:     end if
29: end if
30: $cycle.cumulative\_rtt+ = avg\_rtt\_window$
31: $cycle.n++$
32: return false { The same cycle }
    The newcycle_init() function is:
33: $cycle.duration = now - cycle.start\_time$
34: $cycle.average\_rtt = cycle.cumulative\_rtt/cycle.n$
35: $cycle.start\_time = now$
36: $cycle.current\_rtt = avg\_rtt\_window$
37: $cycle.ascending = notstarted$
38: $cycle.descending = notstarted$
39: $cycle.stableRun\_starttime = now$
40: $cycle.stableRun\_startvalue = avg\_rtt\_window$
41: $cycle.cumulative\_rtt = avg\_rtt\_window$
42: $cycle.n = 1$
43: return true { New cycle }
```

IV. Evaluation

This section presents a subset of the experiments we have performed in order to validate our bandwidth aggregation mechanism. We can not present here all the experiments due to space constraints, but we point the reader to the longer version of this paper [22] for details. We implemented our bandwidth aggregation mechanism in the ns-3 network simulator.

The network setup of our experiments is shown in Fig. 4. The multihoming sender network is behind router R_1 (i.e. the source nodes: s_1 .. s_n). The multihoming receiver network is behind router R_4 (i.e. destination nodes: d_1 .. d_n). There are $n = 64$ local multihoming TCP flows going from source to destination nodes. Router R_1 is a multihoming sender router that splits incoming multihoming flows on the two outgoing paths: *Path1*: $R_1 - R_2 - R_7 - R_4$ and *Path2*: $R_1 - R_3 - R_{10} - R_4$. 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. Routers R_2, R_7, R_3 and R_{10} are ECMP routers that split randomly and equally incoming flows on outgoing links (i.e. R_2 splits flows going to R_7 equally on links $R_2 - R_5$ and $R_2 - R_6$; similarly R_7 splits flows going to R_2 equally on links $R_7 - R_5$ and $R_7 - R_6$). 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 outer routers links $R_1 - R_2$, $R_1 - R_3$, $R_7 - R_4$ and $R_{10} - R_4$ is always set to 10 ms, the transmission delay of the core links $R_3 - R_8$, $R_3 - R_9$, $R_8 - R_{10}$ and $R_9 - R_{10}$ is always 40ms, while the transmission delay of links depicted with red in Fig. 4 is also 40ms, if not specified otherwise. During an experiment, the capacities of all the links from the same network path (i.e. *Path1* or *Path2*) are always equal, although *Path1* capacity value may be different than *Path2* capacity value. The router queue is DropTail and is always set to the bandwidth-delay product for that link, for all routers. The 64 local multihoming flows start in the beginning of the simulation at random times and last until the simulation completes (i.e. 600 seconds). Additional 512 TCP flows attached to source nodes connected to the R_2 router and destination nodes connected to the R_7 router (these nodes are not depicted in Fig. 4) add network load on *Path1*. 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 *Path1*. 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, creating an increased load on *Path1*. Similarly, 128 TCP flows attached to source nodes connected to the R_3 router and destination nodes connected to the R_{10} router (not depicted in Fig. 4) create a steady-state load on the other network path, *Path2*, for the duration of the entire simulation. In addition, there are 32 TCP flows on the reverse path $R_7 - R_2$ and other 32 TCP flows on the reverse link $R_{10} - R_3$ for an increased network dynamics. In our simulations we used a mixture of TCP Linux Cubic, Sack and NewReno flows.

We compared our bandwidth aggregation mechanism that maps multihoming flows dynamically on the two outgoing paths with an ECMP (Equal-Cost Multipath routing [4]) classical routing scheme. We used the following metrics:

- AVGT(*Average throughput per flow*) = the average flow throughput of the 64 multihoming flows

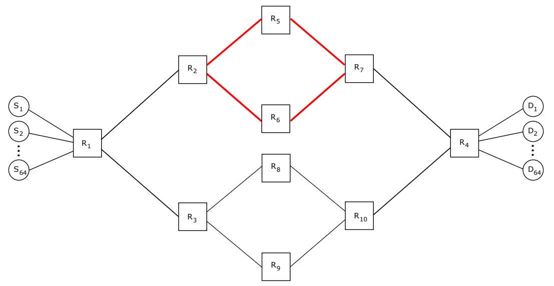


Fig. 4: The network setup used in the experiments

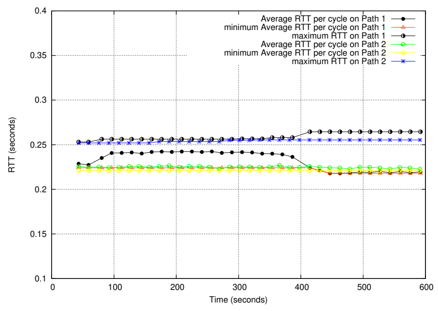


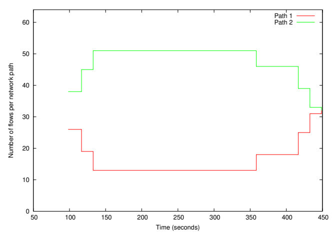
Fig. 5: The $cycle\_average\_rtt$ with minimum and maximum values for each path; RTT-based mapping policy, 100Mbps capacity

- STD(*Standard deviation of the flow throughput values*) = the standard deviation of the 64 throughput values

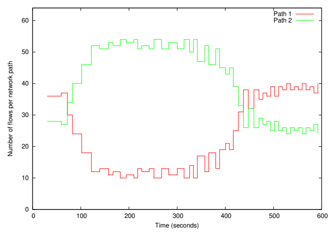
The throughput mentioned above is computed for each multihoming flow during the increase load period of the simulation (i.e. between seconds 40 and 400 of the simulation).

In the first test we have a network capacity of 100 Mbps and the same transmission delay for both network paths, *Path1* and *Path2*. We ran 2 simulations: a simulation with our bandwidth aggregation mechanism employed for router R_1 using the RTT-based load estimation policy and a simulation with our mechanism used at router R_1 with the Throughput-based policy. In Fig. 5 we can see the evolution of the average RTT per cycle measured for each network path when the RTT-based mapping policy was used at router R_1. We notice that while $cycle\_average\_rtt$ remains relatively constant on $Path2$, it increases on $Path1$ between seconds 50-400 as the additional 448 TCP flows create an increased load on links $R_2 - R_7$. This determines router R_1 to map more multihoming flows on $Path2$ than on $Path1$ between seconds 70-380; this is visible in Fig. 6a. The flow mapping for the Throughput-based mapping policy is depicted in Fig. 6b.

Next, we considered three diverse network capacities of 100 Mbps, 500Mbps and 1Gbps and the same transmission delay for both network paths, *Path1* and *Path2*. For each network capacity we ran 3 experiments: one with ECMP routing, second with the RTT-based mapping policy and third with the Throughput-based mapping policy at router R_1. Each experiment consisted of a simulation being run 10 times with different, randomly generated, flow starting and ending times and access links delays. In the end, we computed for each experiment an average of the aforementioned metrics across all 10 simulations performed for the same experiment. The



(a) RTT-based mapping policy



(b) Throughput-based mapping policy

Fig. 6: Number of multihoming flows mapped on each path; 100Mbps capacity

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	STD: 75349.3 AVGT: 112892	STD: 623291 AVGT: 757574	STD: 1083420 AVGT: 1452690
RTT-based mapping (%)	STD: 41.48 AVGT: 31.47	STD: 31.97 AVGT: 22.07	STD: 35.67 AVGT: 29.30
Throughput-based mapping (%)	STD: 43.00 AVGT: 32.53	STD: 57.15 AVGT: 25.14	STD: 45.81 AVGT: 30.20

TABLE I: Results for identical bandwidth and delay for both network paths

obtained results are depicted in Table I. For the ECMP mapping we show the absolute values for the two metrics, but for the bandwidth aggregation mechanism employed (i.e. RTT-based and Throughput-based mapping) we show percentage improvement values for the metrics with respect to the corresponding metric used in ECMP mapping. We can see, as expected, that our bandwidth aggregation mechanism improved both metrics with respect to ECMP routing, in all tested network capacities. The improvements are above 20% for $AVGT$ and above 30% for STD.

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 links depicted with red in Fig. 4 was 60ms, while the transmission delay of all other links remained unchanged. This led to a RTT on *Path1* that was more than 1.5 times the RTT on the network *Path2*. The obtained results are depicted in Table II. We have the same improvements for both metrics when the bandwidth aggregation mechanism was employed at router R_1 (with both RTT-based and Throughput-based mapping policies), similar to what we have seen in the symmetrical RTT-bandwidth experiments (i.e. Table I). Although, the $AVGT$ improvements of the bandwidth aggregation mechanisms are now smaller than the improvements obtained for the symmetrical RTT-bandwidth experiments.

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, the network capacity on all links from *Path1* were double the network capacity of links from *Path2*. The obtained results are depicted in Table III. Please note that for these asymmetrical network capacity experiments, we had to slightly modify the RTT-based mapping algorithm (i.e. the *UpdateRTTState*

Network capacity			
	100Mbps	500Mbps	1Gbps
ECMP mapping	STD: 74639.7 AVGT: 111059	STD: 837451 AVGT: 889329	STD: 1561710 AVGT: 1717350
RTT-based mapping (%)	STD: 43.39 AVGT: 32.72	STD: 50.42 AVGT: 8.75	STD: 51.16 AVGT: 13.56
Throughput-based mapping (%)	STD: 55.18 AVGT: 29.88	STD: 65.24 AVGT: 8.29	STD: 61.45 AVGT: 13.91

TABLE II: Results for identical bandwidth for both paths, but asymmetric delays (links $R_2 - R_5 - R_7$ and $R_2 - R_6 - R_7$ have a 60ms transmission delay; links $R_3 - R_8 - R_{10}$ and $R_3 - R_9 - R_{10}$ have a 40ms transmission delay)

Network capacity			
	200Mbps/ 100Mbps	250Mbps/ 500Mbps	500Mbps/ 1Gbps
ECMP mapping	STD: 71503 AVGT: 117002	STD: 284829 AVGT: 343431	STD: 935580 AVGT: 888124
RTT-based mapping (%)	STD: 35.43 AVGT: 20.30	STD: 30.44 AVGT: 28.88	STD: 42.95 AVGT: 13.13
Throughput-based mapping (%)	STD: 43.78 AVGT: 27.06	STD: 54.33 AVGT: 37.21	STD: 72.66 AVGT: 12.37

TABLE III: Results for identical transmission delays for both paths, but asymmetric bandwidth capacities (every link of *Path1* has 200Mbps/500Mbps/1Gbps bandwidth capacity which is twice the bandwidth capacity of the links from *Path2*: 100Mbps/250Mbps/500Mbps)

algorithm depicted in listing 1) so that after the weights for both network paths are computed we further scaled these weights as following: we scaled the weight of *Path1* by 66% and scaled the weight of *Path2* by 33% (because the network capacity of *Path1* is double the capacity of *Path2*). 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 *Path1* and 33% of the flows on *Path2*.

V. Conclusions and Future Work

We have presented a multihoming routing solution for bandwidth aggregation. Our solution comes in the form of a virtual tunnel that connects two sites through multiple independent or quasi-independent network paths. The routing solution maps local flows on the possible outgoing network paths so that these flows use a larger aggregated 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 developed a RTT-based and a Throughput-based strategy for estimating the load (congestion) on a network path. Both strategies involve only passive measurements and they require maintaining a fairly low amount of state per flow at the edge router. We have tested our bandwidth aggregation mechanism in a simulated network and we showed that our routing solution performs better than ECMP routing in terms of total aggregated throughput and fairness between multihoming flows. As future plans, we want to evaluate the performance of both mapping strategies with self-limiting flows (non-greedy TCP flows). Also we would like to study how two such multihoming routing solutions interact with each other.

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*, 2014 ACM SIGCOMM Conf., USA, 2014, pp.503-514.

[2] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, T. Edsall, *Let it flow: resilient asymmetric load balancing with flowlet switching*, 14th USENIX Conf. on Networked Systems Design and Implementation,2017,pp.407-420.

[3] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford, *Clove: Congestion-Aware Load Balancing at the Virtual Edge*, 13th International Conference on Emerging Networking Experiments and Technologies, USA, 2017, pp.323-335.

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

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

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

[7] 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.

[8] 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.

[9] Y. Li, Y. Zhang, L. L. Qiu, S. Lam, *SmartTunnel: Achieving Reliability in the Internet*, 2007 IEEE Infocomm,USA,2007,pp.830-838.

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

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

[12] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka *A new TCP for persistent packet reordering*, IEEE/ACM Trans. on Networking, Vol. 14, Issue 2, pp.369-382, April, 2006.

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

[14] 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.

[15] E. Arslan, B. Ross, and T. Kosar, *Dynamic Protocol Tuning Algorithms for High Performance Data Transfers*, European Conference on Parallel Processing, Germany, 2013, pp.725-736.

[16] E. Arslan, K. Guner, and T. Kosar, *HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-time Probing*, Conference for High Performance Computing, Networking, Storage and Analysis, USA, 2016, pp.288-299.

[17] T. Kosar, E. Arslan, B. Ross, and B. Zhang, *StorkCloud: data transfer scheduling and optimization as a service*, 4th ACM workshop on Scientific Cloud Computing, USA, 2013, pp.29-36.

[18] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat, *B4: experience with a globally-deployed software defined wan*, 2013 ACM SIGCOMM Conf., USA, 2013, pp.3-14.

[19] N. Gvozdiev, B. Karp, M. Handley, *FUBAR: Flow Utility Based Routing*, 13th ACM Workshop on Hot Topics in Networks,2014,pp.12-18.

[20] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, *Flexible Traffic Splitting in OpenFlow Networks*, IEEE Trans. on Network and Service Management, Vol. 13, Issue 3, pp.407-420, September, 2016.

[21] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, *Analysis and Design of the Google Congestion Controlfor Web Real-time Communication (WebRTC)*, ACM Multimedia Systems Conference,2016,pp.1-12.

[22] 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.