Bandwidth Aggregation over Multihoming Links

Adrian Sterca[a,*], Darius Bufnea[a], Virginia Niculescu[a]

[a]*Dept. of Computer Science, Babes-Bolyai University, Romania*

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. 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.

Keywords: multihoming, multipath load-balancing, multipath routing, ECMP routing

1. Introduction and Problem Formulation

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). But this is also becoming a typical situation for the end user too; for example, a mobile phone can be connected to a 4G/LTE network and at the same time to the local wireless network. 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 *Path1* and the latter *Path2* 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 *Path1* (i.e. through ISP1) and over *Path2* (i.e. through ISP2). Similarly, Site B can send packets coming from B's local network over *Path1* (i.e. through ISP3) and over *Path2* (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 *Path1* or *Path2*) such that:

- the total, aggregated throughput from site A to site B is maximized (aggregated over *Path1* and *Path2*)
- it maintains a high degree of fairness between flows sent from Site A to Site B over *Path1* and *Path2*
- this routing policy works with existing Internet technology (i.e. it does not rely on feedback or actions from special equipment deployed inside the network core)
- routing local flows over *Path1* or *Path2* is transparent to local computers from site A and site B.

Because of the last condition, our solution comes in the form of a virtual tunnel interface between Site A and Site B (this is 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, as a local computer from Site A that sends data to a Site B computer through the tunnel is not aware of the physical path (either *Path1* or *Path2*) the packets will take towards 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 [5].

*corresponding author

Email addresses: `forest@cs.ubbcluj.ro` (Adrian Sterca), `bufny@cs.ubbcluj.ro` (Darius Bufnea), `vniculescu@cs.ubbcluj.ro` (Virginia Niculescu)

[1]Str. M.Kogalniceanu, No. 1, Cluj-Napoca, Romania

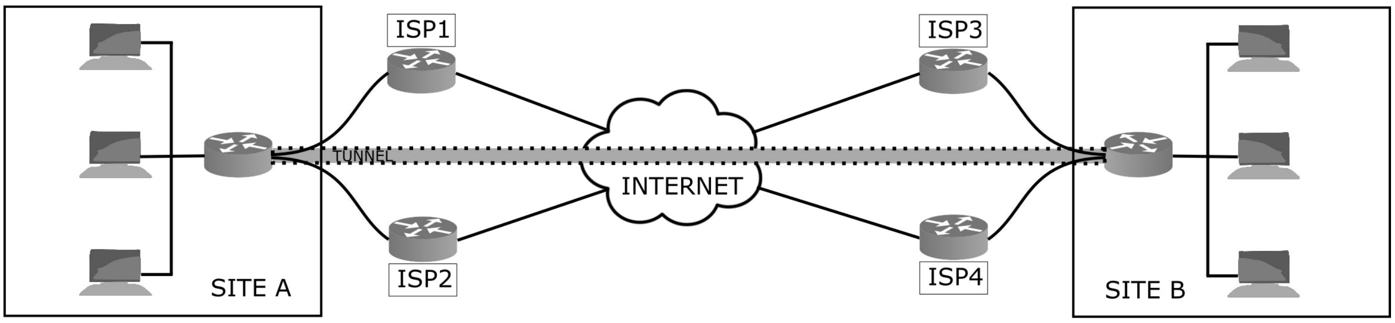

Figure 1: The typical network setup of the multihoming problem

The aforementioned goals should be achieved in a context of changing network conditions, i.e. available bandwidth and delay for both network paths may change in time depending on the network load. For instance, consider 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. Then it makes sense to send $N/2$ flows through *Path1* and $N/2$ flows through *Path2*. But at some point, due to an increased number of flows entering the network through *Path1*, this path becomes congested and the available bandwidth drops to $X/2$ Mbps. In these conditions, it is more efficient to 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. The company's employees from Site A open several TCP connections to computers from Site B through a VPN tunnel: audio-video conferencing, data transfer, remote management sessions etc. All employees from Site A want a maximum throughput for their flows and also want that all flows of all employees achieve approximately the same throughput - so that no flow/employee is favored over other flows/employees. In another example, 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. The replication process should not take too much time, so a maximum throughput is needed for all flows.

We will give in the following sections two such flow mapping policies that manage to adapt dynamically the number of flows on each physical network path to reflect the load of that path. One policy is based on estimating the network load on a link based on RTT (i.e. Round-Trip Time) measurements and the other policy estimates the available bandwidth on the link using throughput measurements. We will evaluate both policies extensively in Section 5 in order to identify their individual strengths and weaknesses. The rest of the paper is structured as follows. Section 2 discusses some assumptions and introduces some notations. Section 3 presents related work in the field of multipath data transfer. The main contribution of the paper, that is the mechanism for aggregating bandwidth over multihoming links is detailed in Section 4. Following, we present the evaluation experiments in Section 5 and end the paper with conclusions and future work ideas in Section 6.

2. Assumptions and notations

In this paper we assume that the network paths are set by an administrator and we do not deal with the problem of actually selecting network paths between Site A and Site B. For example, in Fig. 1 there are 4 possible network paths between Site A and Site B: ISP1-ISP3, ISP1-ISP4, ISP2-ISP3, ISP2-ISP4 but we consider that an administrator selected two paths : ISP1-ISP3 and ISP2-ISP4. In the following sections we will deal with the general problem where we have 2 network sites (i.e. LANs), Site A and Site B, just like in Fig. 1 and these sites are connected through m independent or quasi-independent network paths, labeled *Path1*, *Path2*, ... *Pathm*. By 'independent network paths' we mean that the network paths do not have network segments in common and by 'quasi-independent network paths' we refer to network paths that may share common network segments, but have different bottleneck links (i.e. the network segment with the lowest available bandwidth for a newcomer flow from that path). If this assumption would not be true (i.e. network paths are not quasi-independent), our bandwidth aggregation mechanism should not perform worse than a naive routing solution that maps N/m flows on each path (where N is the total number of flows). We consider that our multihoming bandwidth aggregation algorithm runs at Site A's edge router and the edge router from Site B is just an ordinary router with a typical hop-count based routing policy. Although, the same multihoming problem can be formulated for the reverse direction, i.e. for multihoming flows originating at Site B and sending data to destinations from site A LAN; in which case the bandwidth aggregation mechanism would run also on Site B's edge router. We have N flows that originate at Site A and send data to destinations from Site B network. These flows are distributed by Site A's edge router over the network paths *Path1*.. *Pathm*. We will refer to these flows as *local multihoming flows* in the rest of the paper. We assume the local multihoming flows are all TCP greedy flows (i.e. they always have data to send).

3. 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 multipath set (i.e. 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 consider a flow to be a stream of data packets all having the same source and destination endpoints (where endpoints can be identified by their IP addresses). 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 (a flow is assigned on a specific network path and it stays on that path throughout its lifespan), but different flows may be assigned to different paths), b) packet-level (each packet of every flow is assigned to a specific, possibly different, network path) or c) subflow-level (a part of a flow, either called flowlet or flow cell is assigned to a specific network path).

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, 3, 4] [2]. Conga [1] is implemented in the hardware of the switch and splits incoming flowlets (i.e. a burst of packets from a flow separated from the previous and following packets from the same flow by a minimum time interval) on switch links depending on the degree of utilization of the links striving to minimize the maximum utilization on these links. LetFlow [3] is also implemented in switch hardware and splits flowlets across switch links randomly. Similarly, Presto [2] and Clove [4] are implemented in virtual switches of the hypervisor and split flow cells / flowlets on switch links depending on the utilization of the link. All these solution have in common the fact that they 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) [6, 7, 8, 9, 10, 11, 12] and others perform traffic engineering across AS domains for BGP routing [14, 15]. Inside an AS domain the traffic splitting over multiple paths is performed at Ingress routers and the paths span until the Egress router. The classical way of performing load balancing across multiple paths is using Equal-Cost Multipath routing (ECMP) [5], 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. Because of this it can not easily adapt to changing traffic patterns. Another way of performing traffic engineering in an ISP network is the so called *predicted-based TE* where ISPs use traffic matrices that represent the traffic demand in the ISP network across a large time interval (e.g. months, weeks or days) and use this estimation to spread flows on multiple network paths [7, 8, 9, 10]. Because the traffic matrices are evaluated over such long periods of time, these techniques do not cope with fast changes (i.e. with timelines shorter than a day) in traffic for example due to diurnal variations, flash crowds, attacks, BGP re-routes, external or internal failures. A third way of doing TE in an ISP network is *online traffic engineering* exemplified by TeXCP [11]. TeXCP measures the path utilization at each router by actively probing these routers, so it relies on their support for getting the path utilization feedback. Based on this feedback, it adapts the load on each path by sending flows on low-used links, thus reducing the load on highly used links. 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. SmartTunnels [15] are tunnels that perform multipath load balancing combined with FEC coding in order to achieve reliability across ISP networks. The sender-part and receiver-part of the tunnel do not rely on feedback from intermediary router, but have a quite complex architecture, including buffers for flow splitting at packet level at the sender and for reordering flow packets at the receiver, FEC encoder/decoder. The buffers at the sender and receiver will only add additional delay to traffic. Galton [14] also employs a tunnel architecture with a complex sender and receiver in order to increase the goodput of real-time, audio-video traffic. The sender and receiver also have buffers for packet scheduling and packet reordering. It does not rely on feedback from the core routers, but uses active probing for monitoring the available bandwidth on different network paths. Contrary to these tunnels, our technique does not use buffers at the sender or receiver in order to schedule or reorder packets, 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 [16] or SCTP [17, 20] or by changes to classical TCP [19, 21, 22]. 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 classical TCP which is the most used transport-level protocol in the Internet today.

Various mechanisms of splitting a data transfer across multiple paths were also tried at the application-level [23, 24, 25]. Also in the context of SDN (Software Defined Networking)

[2] Actually they can be considered level 2/3 solutions since they also consider IP addresses when defining flows, but we chose to classify them as level 2 solutions since they are implemented in physical or virtual/hypervisor switches

for custom network architectures several solutions that perform multipath traffic engineering were developed [26, 27, 28] [3].

A more complete survey on multipath load balancing techniques can be found in [13].

4. 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 capacity and delay) and on their current network load (i.e. congestion level). If at some point, the load increases on a particular network path (due to new flows entering this path), it makes sense to move a part of the local multihoming flows assigned to this path on the other multihoming paths. Doing this, would theoretically increase the inter-flow fairness for the local multihoming flows and also would increase the aggregated throughput since more local multihoming flows go through high-capacity links. In order to do this, our mechanism requires two components:

- a component that measures the network load on each uplink (*network load estimation policy*)
- a component that manages local multihoming flows and maps them on outgoing links.

We will first describe the second component, that is the algorithm for mapping flows on the outgoing links. This algorithm is depicted in Fig. 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. There are two **for** loops in the algorithm. The first loop (i.e. lines 2-9) 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 $Path_i$). Then, in the second **for** loop (i.e. lines 10-18), we take each flow from the set R and assign them to the new, proper path (according to the new weights). 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:          {Sort descending the set of flows from $Path_i$ by last
            remapping time: youngest flow on the path first}
6:          SortByRemappingTime ($Flows(Path_i)$)
7:          $R = R +$ GetFlowsForRemap($Flows(Path_i), flows\_to\_remap$)
8:      end if
9:  end for
10: for $i = 1$ to $m$ do
11:     $flows\_to\_remap = \lfloor weight_i \cdot N \rfloor - \lfloor old\_weight_i \cdot N \rfloor$
12:     if $flows\_to\_remap > 0$ then
13:         for all $flow$ in GetFlowsForRemap($R, flows\_to\_remap$)
            do
14:             {assign $flow$ to $Path_i$}
15:             $flow.path = i$
16:         end for
17:     end if
18: end for
```

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

The first network load estimation policy that we introduce is based on RTT passive measurements. 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 [4]. This figure depicts graphically the evolu-

[3] We consider SDN to be an application-level solution because the SDN controller and SDN applications are usually implemented at application-level

[4] In order to ease the understanding of these RTT fluctuation types, we created Fig. 2 from artificial data, not from real, measured data, so that the fluctu-

tion of RTT samples as measured by returning (i.e. ACK) TCP packets that pass through a site router. All these TCP packets belong to TCP flows that share the same network path. There are 3 types of fluctuations presented in this figure:

- tiny-scale fluctuations - caused by other flows sharing a segment of the same network path that disrupt the RTT measurements done by the monitored flows (i.e. packets of these other flows get interleaved with packets from the monitored flows in the bottleneck link's queue causing tiny, under 10 milliseconds gains or drops of RTT)
- small-scale fluctuations - caused by the typical operation of TCP flows / router queuing system where TCP flows probe greedily for more available throughput until they overshoot the network capacity and the queue in routers overflows causing TCP flows to drop throughput, then again TCP flows probe for additional throughput until they overflow the network capacity and the process continues indefinitely causing periodic cycles in the measured RTT; this type of RTT fluctuations are used, for example, by TCP Vegas [29] and GCC for WebRTC [31] to adjust the transmission window.
- large-scale fluctuations or fluctuations caused by significant network load change - caused by a significant set of flows entering or leaving the network (a large-scale fluctuation can be seen in Fig. 2 in between the two small scale fluctuation cycles, containing a period of heavy load when the RTT is kept at a relatively constant, high value).

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)** we do 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 (classical) average of RTT values in a cycle to remove the second type of fluctuations (i.e. small-scale fluctuations).

For the first smoothing stage, each time we receive a new ACK package, we take the most recent 16 RTT samples window 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 value 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. We have tried in our tests various average functions: equal-weighted average on a 16-values window, exponentially moving average on a 16-values window (with a weight larger than 0.8 for the previous average and a weight smaller than 0.2 for the current RTT sample), also various choices for the size of the window: 8, 16, 32, 64, 128, but the one that achieved the best results was the mixed equal+exponential weighted average on windows of 16 RTT samples. The size of the window did not have a big impact, meaning that 32, 64 and 128 were just as good as 16, but we chose 16 in order to reduce the memory overhead.

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**. In order to define what a **RTT cycle** is, we need to first introduce additional concepts. We call a subsequence (RTT_i, RTT_j) made of 2 RTT samples, *quasi-constant* if $RTT_i \in [RTT_j - thresh, RTT_j + thresh]$ where $thresh$ is a positive threshold. Similarly, this subsequence is called *ascending* if $RTT_i > RTT_j + thresh$ and is called *descending* if $RTT_i < RTT_j - thresh$. After the first smoothing function is applied, the resulted RTT sequence will contain many segments (i.e. subsequences) of the following types:

1) *ascending segment* - is the largest continuous subsequence of RTT samples $(RTT_m, RTT_{m+1}, ..., RTT_n)$ that has the following properties:

- $\forall i \in [m, n)$, the subsequence (RTT_i, RTT_{i+1}) is either quasi-constant or ascending
- the longest quasi-constant subsequence from this segment does not have a length larger than $stable\_run\_thresh$
- and the ends of the subsequence are ascending (i.e. $RTT_n > RTT_m + thresh$)

where $thresh > 0$, $stable\_run\_thresh > 0$ are thresholds, RTT_m is the first sample and RTT_n is the last sample of the ascending segment.

2) *descending segment* - is the largest continuous subsequence of RTT samples $(RTT_m, RTT_{m+1}, ..., RTT_n)$ that has the following properties:

- $\forall i \in [m, n)$, the subsequence (RTT_i, RTT_{i+1}) is either quasi-constant or descending
- the longest quasi-constant subsequence from this segment does not have a length larger than $stable\_run\_thresh$
- and the ends of the subsequence are descending (i.e. $RTT_n < RTT_m - thresh$)

where $thresh > 0$, $stable\_run\_thresh > 0$ are thresholds, RTT_m is the first sample and RTT_n is the last sample of the descending segment.

3) *stable-run segment* - is a continuous subsequence of RTT samples with the property that any two samples are quasi-constant and the length of the segment is larger than $stable\_run\_thresh > 0$.

Please notice that an ascending and a descending segment can both contain quasi-constant subsequences, but they are not long enough to be considered stable-run segments. But when the network is not seriously congested, quasi-constant subsequences

ation types are better emphasized.

do not usually occur, because they are filtered out by the first smoothing function.

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, (i.e. containing one ascending and one descending segment), 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 (failing to map fewer flows on the heavy loaded network link) - since flow remappings happen only at the end of a RTT cycle.

We further smooth out the RTT samples string by averaging values over an RTT cycle, so that the number of flow remappings will be reduced. The algorithm used for computing the average value of an RTT cycle is detailed in listing 3 and will be discussed later in the paper. 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 real RTT measurements taken from a set of flows passing through a network path that has a low load/congestion level (i.e. a small, constant, number of flows are sharing the path) 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 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 that 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.

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 [5] 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

[5]https://tools.ietf.org/html/rfc7323

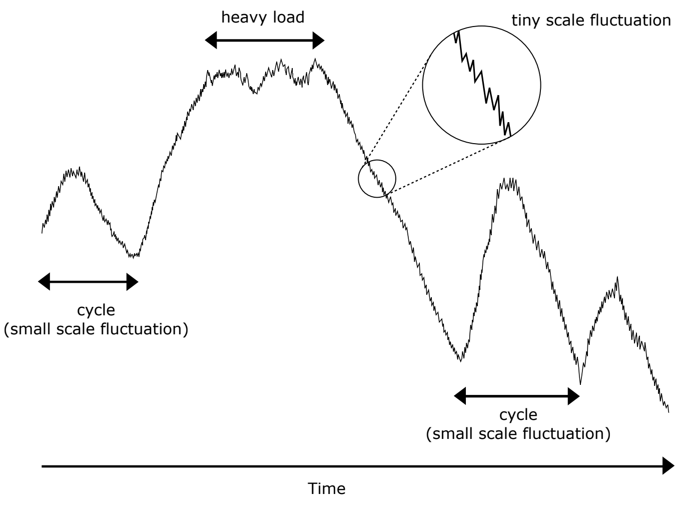

Figure 2: Typical RTT fluctuations of flows (idealized drawing)

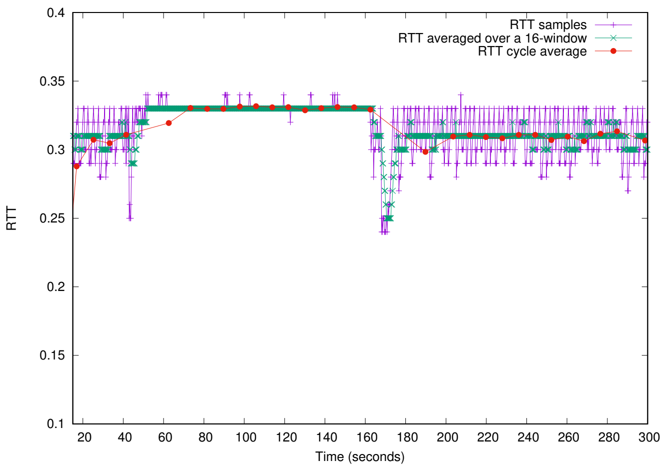

Figure 3: The two-stages smoothing performed on RTT samples

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 average (i.e. first smoothing function) is computed first in line 10. The *UpdateRTTWindow* algorithm computes the weighted average of the last 16 RTT samples recorded. It is a mixed equal and exponential weighted average. The weights used are the following:

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

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

Thus we have weight 1 for the most recent 8 RTT samples and then, the weights start dropping exponentially giving less weights to older samples. The value 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. Then, if the function *UpdateRTTCycle*, which is described in Listing 3, 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 of a path is computed as an inverse linear mapping of *cycle.average_rtt* from the interval [*min_cycle_avgrtt*,

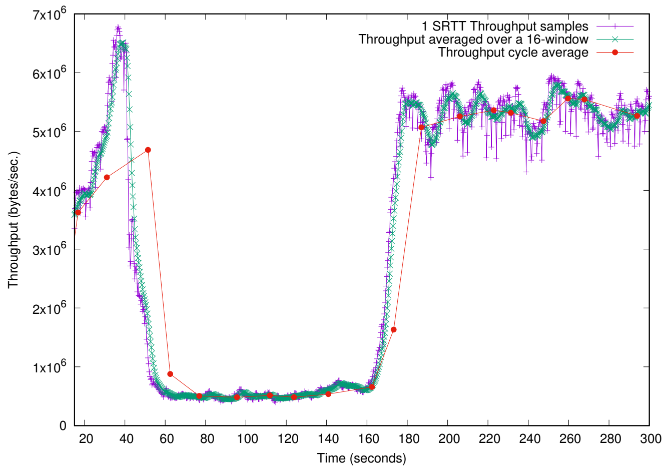

Figure 4: The two-stages smoothing performed on Throughput samples

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 simplest way of comparing the network load of two network paths would be to just compare their average RTT per cycle (i.e. *cycle.average_rtt*). As the network path gets more congested (i.e. load increases), its average RTT per cycle would also increase. But this does not work for asymmetric network paths where the transmission delays on these paths differ significantly. An RTT sample measurement includes two types of delays: transmission delays (i.e. the amount of time it takes for a packet to travel from one end to the other end of the path, assuming it is the only packet on the network path) and queuing delay (i.e. the delay caused by waiting on a queue in the bottleneck link's router). Since the network load influences only the queuing delay, not the transmission delay, but an RTT sample encapsulates both, we can not directly compare the RTT values of two asymmetric network paths in order to detect their relative network load difference. A better metric is to take the *cycle.average_rtt* - *min_rtt* difference for each network path (where *min_rtt* is the minimum RTT ever recorded on the path). Assuming we have enough history log, the *min_rtt* would be very close to the actual transmission delay on the network path and will include in its value a queuing delay close to zero. But the difference *cycle.average_rtt* - *min_rtt* estimates the network load on the path including the network load created by the local multihoming flows and we want to estimate only the network load induced inside the network by external flows. This is why, we consider the metric *cycle.average_rtt* - *min_cycle_avgrtt* where *min_cycle_avgrtt* is the minimum *cycle.average_rtt* recorded out of all RTT cycles on this path. The *min_cycle_avgrtt* would be a good estimator of the average queuing delay (i.e. network load) generated when only local multihoming flows pass through the network path. Still, network paths can be so diverse and if the bandwidth-delay product on the bottleneck link of a path is very different than the one from another network path, in order to compare the paths' network load we need to normalize the metric *cycle.average_rtt* - *min_cycle_avgrtt* to the interval [0,1]. Thus we obtain the network load metric from line 16 of the algorithm. When *cycle.average_rtt* is equal to *min_cycle_avgrtt* which means the network is not congested, line 16 gives a network load metric value of 0. But if the network path is severely congested and *cycle.average_rtt* is close to *max_rtt*, the network load metric computed by line 16 is 1. Of course, lines 14-21 of the algorithm *UpdateRTTState* must compute a weight for each network path, i.e. this weight specifies how many flows should be mapped on each path and is used in the algorithm *FlowRemapping*. This is why we need to inverse the network load metric computed in line 16. That is, a network load metric of 0 should give a maximum weight of 1 (i.e. all local multihoming flows should be mapped on this path because it has no network load) and conversely, a network load metric of 1 should give a weight of 0 (i.e. the path is highly congested and local multihoming flows should be mapped on other paths). This weight inversion is performed in lines 19-21 for all network paths.

The *UpdateRTTCycle* algorithm depicted in listing 3 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 $Path_k$. As we said earlier, a cycle is a sequence of continuous values containing either an ascending segment and a descending segment or a stable-run segment or a stable-run segment preceded by an ascending/descending segment. The state maintained for a cycle is the following (one cycle state is maintained for each network path):

cycle.duration: the length in time of the previous, completed cycle
cycle.average_rtt: the average value of the previous, completed cycle
cycle.start_time: the starting time of the current cycle
cycle.current_rtt: the current sample value from the current cycle (it's actually an average over a window of 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 (possible stable-run) subsequence in the current cycle
cycle.stableRun_startvalue: starting value of the last quasi-constant (possible stable-run) 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 13-21: we check if this 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 22-30 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 32-33. The initialization of a new cycle is performed in lines 35-44. Note that while most of the properties of a cycle are initialized in lines 35-44 for the following cycle, the properties *cycle.duration* and *cycle.average_rtt* actually hold data for the previous cycle (i.e. the one that just ended).

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

The throughput-based policy is very similar to the RTT-based policy. The throughput-based policy assumes that all multihoming flows are greedy TCP flows (i.e. they can use as much bandwidth as available, they are not self-limiting sources) and 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 (i.e. number of non-multihoming flows sharing the network path) has reduced.

The total throughput of all multihoming flows going out a network path is measured once per RTT. This throughput measure shows, as in the case of RTT measurements for the RTT-based policy, three types of fluctuations: *a)tiny-scale fluctuations* due to congestion window - router queue dynamics, *b)small-scale fluctuations* caused by the periodicity of TCP congestion control behavior (i.e. TCP increases the throughput until it overshoots the network capacity and then halves the throughput only to start increasing it again, searching for new, available bandwidth) and *c)large-scale fluctuations* caused by large flows crowds entering or leaving the network. Therefore we use the same methodology as for the RTT-based policy, in order to reduce tiny-scale and small-scale fluctuations of the measurements. More specifically, we apply the same mixed equal+exponential weighted average on windows of 16 throughput measurements in order to reduce tiny-scale fluctuations and then, we compute throughput cycles and use the average throughput value for a cycle in order to reduce small-scale

Algorithm 2 The RTT-based network load estimation policy is executed when a new return packet (i.e. TCP acknowledgment packet) arrives at the multihoming sender router:

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:  $curr\_rtt = now - p.TS\,ecr$
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**
```

fluctuations. Both smoothing stages work in the same way as their correspondent in the RTT-based load estimation policy, but with different thresholds when computing throughput cycles. The only thing that is different in the throughput-based policy is the algorithm used for updating the throughput state, i.e. the *UpdateThroughputState* algorithm depicted in listing 4 which will be discussed in the following paragraphs. But before we do that, we can see in Fig. 4 the effect of applying the two stage smoothing methodology on the throughput samples measured once per second. This figure is the correspondent of Fig. 3 for the throughput-based policy. The raw throughput samples are depicted with the label '1 SRTT Throughput samples', the 16 window averages are shown under the label 'Throughput averaged over a 16 window' and finally, the cycle-average values are shown under the label 'Throughput cycle average'. We can see here that the cycle average line is much stable than the other

Algorithm 3 The UpdateRTTCycle algorithm checks whether a new cycle is starting for $Path_k$

Input:
avg_rtt_window : the average RTT value over a window of 16 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
13:         { We are in the ascending phase }
14:         if $cycle.ascending = ended$ then
15:             return newcycle_init()
16:         end if
17:         if $cycle.descending = started$ then
18:             $cycle.descending = ended$
19:         end if
20:         $cycle.ascending = started$
21:     else if isDescending=TRUE then
22:         { We are in the descending phase }
23:         if $cycle.descending = ended$ then
24:             return newcycle_init()
25:         end if
26:         if $cycle.ascending = started$ then
27:             $cycle.ascending = ended$
28:         end if
29:         $cycle.descending = started$
30:     end if
31: end if
32: $cycle.cumulative\_rtt += avg\_rtt\_window$
33: $cycle.n++$
34: return false { The same cycle }

    The newcycle_init() function is:
35: $cycle.duration = now - cycle.start\_time$
36: $cycle.average\_rtt = cycle.cumulative\_rtt / cycle.n$
37: $cycle.start\_time = now$
38: $cycle.current\_rtt = avg\_rtt\_window$
39: $cycle.ascending = notstarted$
40: $cycle.descending = notstarted$
41: $cycle.stableRun\_starttime = now$
42: $cycle.stableRun\_startvalue = avg\_rtt\_window$
43: $cycle.cumulative\_rtt = avg\_rtt\_window$
44: $cycle.n = 1$
45: return true { New cycle }
```

two lines.

The throughput state is maintained by the *UpdateThroughputState* algorithm depicted in listing 4. This algorithm gets executed whenever a data packet coming from the local network arrives at the multihoming router and will be sent through the multihoming tunnel over the network path $Path_k$. The throughput is computed once per $srtt_k$. Therefore, if at least $srtt_k$ time has passed since the last throughput update (i.e. condition checked in line 1) we compute the new throughput value. Otherwise, the size of the current packet p is added to the current throughput in line 19. Line 2 of the algorithm computes the current throughput value for $Path_k$. Following, in line 3 we compute the average of the last 16 measured throughput values. Function *UpdateThroughputWindow()* is exactly the same as the function *UpdateRTTWindow()* used for the RTT-based policy, so we omit it in the article. In line 6 we check whether a throughput cycle has completed. The function *UpdateThroughputCycle()* computes throughput cycles in the same way as the function *UpdateRTTCycle()* for the RTT-based policy (depicted in listing 3) using the same *CTHRESH* and *MIN_REMAPPING_TIME_INTERVAL* thresholds, so we also omit it here. If a new throughput cycle has started, we compute the new path weights and perform a flow remapping if the weights changed in lines 8-16. The weight of a network path is equal to the average cycle throughput on that path $cycle\_average\_throughput_i$, divided by the number of local multihoming flows mapped on that path, $nflows_i$. At the end, we call function *FlowRemapping()* in line 16 which is the same function used in the RTT-based policy and was shown in listing 1, but because this function expects path weights to be normalized to the interval [0, 1] we do this normalization in lines 13-15. The function *FlowRemapping()* just reassigns flows to the multihoming paths according to the paths' new weights.

5. Evaluation

This section details the experiments we have performed in order to validate our bandwidth aggregation mechanism together with the two network load estimation policies, RTT-based and Throughput-based. We implemented our bandwidth aggregation mechanism as a multihoming routing classifier in the **ns** network simulator [6]. In all our experiments, we considered the common case where there are 2 independent network paths between the two multihoming sites (similar to Fig. 1). So, the whole presentation from this section takes place in this setup.

In our experiments we have used the algorithms described in Section 4, but we operated some slight changes on them that should improve their efficiency. These changes are detailed in the following lines. For both estimation policies, RTT-based and Throughput-based, we enforced that whenever the *FlowRemapping* algorithm is executed: *(a)* maximum 10% of the total number of multihoming flows can be moved simultaneously from one link to another (i.e. for each link, the absolute difference between the old weight and the new weight can not

[6]https://www.nsnam.org/

Algorithm 4 The Throughput-based network load estimation policy is executed when a new data packet coming from the local LAN arrives at the multihoming sender router:

Input:
p : a data packet coming from the local network and waiting to be sent through the multihoming tunnel over $Path_k$
m : the number of network paths
$srtt_k$: smoothed RTT for $Path_k$
$last\_throughput\_update_k$: last time the Throughput state was updated for $Path_k$
now : the current time
$throughput_k$: the current throughput on $Path_k$

The UpdateThroughputState algorithm is:

```
1:  if $last\_throughput\_update_k < (now - srtt_k)$ then
2:      $throughput_k = throughput_k/(now - srtt_k)$
3:      $avg\_window\_thrput$ = UpdateThroughputWindow($Path_k$, $throughput_k$)
4:      $throughput_k = 0$
5:      $last\_throughput\_update_k = now$
6:      if (UpdateThroughputCycle($Path_k$, $avg\_window\_thrput$) = 1 ) then
7:          { compute the weight for each Path }
8:          $sum = 0$
9:          for $i = 1$ to $m$ do
10:             $weight_i = cycle\_average\_throughput_i \,/\, nflows_i$
11:             $sum = sum + weight_i$
12:         end for
13:         for $i = 1$ to $m$ do
14:             $weight_i = weight_i \,/\, sum$
15:         end for
16:         FlowRemapping()
17:     end if
18: end if
19: $throughput_k += p.size$
```

be larger than 0.1) and *(b)* each link should have at least 10% of the total number of multihoming flows mapped on it (i.e. the weight of a link can not be smaller than 0.1). Condition *(a)* is meant to give a smooth transition between flow remapping periods, thus avoiding scenarios when almost all multihoming flows (e.g. 64 in our experiments) are moved from one uplink to another, all at once, causing large drops of throughput. Condition *(b)* ensures that both uplinks are utilized and no uplink remains with zero flows mapped on it.

The second type of changes we have done to the algorithms listed in Section 4 are some filters that are meant to reduce weight fluctuations when the two network paths are uncongested. For the RTT-based policy, when both network paths are uncongested, the difference $cycle.average\_rtt_i - min\_cycle\_avgrtt_i$ in Algorithm 2, line 16, becomes very small for both network paths, therefore making the $weight_i$ metrics computed by line 16 very similar. But due to the normalization process from line 20 in Algorithm 2, the relative difference between the two final weights increases and produces different-sized flow sets mapped on each network path (although both network paths are uncongested and there should be $N/2$ flows mapped on each path; i.e. $weight_1$ and $weight_2$ should be both 0.5). Therefore, when *(a)* the $weight_i$ computed by line 16 in Algorithm 2 is less then 0.2 for all network paths or when *(b)* the numerator of line 16 in Algorithm 2 is less then 5 milliseconds for all network paths, we set equal final weights for all network paths (e.g. if we have 2 paths, both paths would have the final weight 0.5 and each path will receive half of the multihoming flows). We have used a similar filter for the Throughput-based policy: if the absolute difference between the two unnormalized weights computed by line 10 in Algorithm 4 is less than 0.05 we set equal final weights for both network paths. We have come to these values through extensive testing detailed in the remaining of this section.

5.1. Network setup

The network setup of our experiments is presented in Fig. 5. 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 links: R_1-R_2 and R_1-R_3. 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 (e.g. if data packets of the TCP flow originating from the source node s_i arrive at the router R_4 through the R_3-R_4 link, then acknowledgment packets generated by the TCP receiver, d_i, will be sent back to the s_i node by the R_4 router through the same link/path, R_4-R_3). 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 inter-router links R_1-R_2, R_1-R_3 and R_3-R_4 is always set to 40 ms, while the transmission delay of link R_2-R_4 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. We have used several network capacities for the inter-router links across all our experiments, ranging from 20 Mbps to 1 Gbps. 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: the most common, Drop-Tail queuing, in order to see how our mechanism works with the most common queue management policy in the Internet and an active queue management policy, Random Early Detection (RED) [30], which is implemented in some commercial routers. We have set the *thresh_* and *maxthresh_* parameters of a RED queue to be 40% and respectively 60% of the total queue size. 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

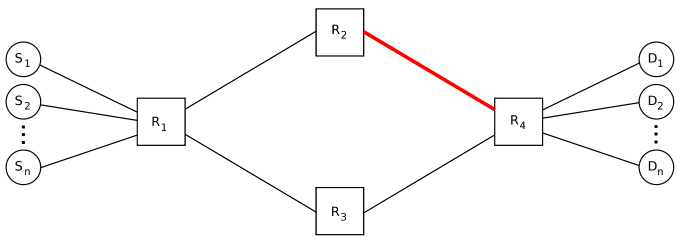

Figure 5: The network setup used in the experiments

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. 5) add network load on the network path $R_1 - R_2 - R_4$. 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 the network path $R_1 - R_2 - R_4$. 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 the path $R_1 - R_2 - R_4$ between seconds 40 and 400 of the simulation, thus forcing our multihoming sender router R_1 to send more flows on the other network path, $R_1 - R_3 - R_4$. 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. 5) create a steady-state load on the other network path, $R_1 - R_3 - R_4$, 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 flows.

5.2. Metrics used

We compared our bandwidth aggregation mechanism that maps multihoming flows on the two outgoing paths dynamically with a naive routing mechanism that splits multihoming flows equally between the two outgoing network paths - the ECMP (Equal-Cost Multipath routing [5]) classical routing scheme (assuming that the two outgoing paths/links have the same network capacity). We will emphasize the benefits brought by our bandwidth aggregation mechanism through the use of the following three metrics:

- *Average throughput per flow* = the average flow throughput of the 64 multihoming flows
- *Standard deviation of the flow throughput values* = the standard deviation of the 64 throughput values
- $\frac{Min\_throughput}{Max\_throughput}$ = the ratio of the minimum flow throughput and the maximum flow throughput out of the 64 multihoming flows.

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. between seconds 40 and 400 of the simulation). This throughput is computed for every flow in the 64 multihoming flows set. The first metric will show whether the 64 multihoming flows consumed more bandwidth using our bandwidth aggregation mechanism than when using ECMP routing. The second metric should be reduced when using our bandwidth aggregation mechanism, thus increasing the bandwidth fairness between competing multihoming flows. A large value for the third metric (i.e. a value close to 1) shows that the multihoming flows received similar throughputs while a small value implies increased throughput unfairness between competing multihoming flows.

5.3. Test results

We start with a common test in which we have a network capacity of 100 Mbps and a 40 ms transmission delay for each inter-router link. We first ran a simulation with an ECMP routing mechanism that splits the 64 multihoming flows equally between the two outgoing network paths: 32 flows on path $R_1 - R_2 - R_4$ and 32 flows on path $R_1 - R_3 - R_4$. Then we ran the simulation again with our bandwidth aggregation mechanism employed for router R_1 using the RTT-based policy for estimating the network load and finally, we ran the same simulation, but the bandwidth aggregation mechanism used the Throughput-based policy. We ran these 3 simulations with DropTail queuing employed at all routers and then we ran the simulations again with RED employed at routers. The results obtained are depicted in Fig. 6 - 13. In Fig. 6 we can see the evolution of the average RTT per cycle measured for each network path (*Path*0: $R_1 - R_2 - R_4$ and *Path*1: $R_1 - R_3 - R_4$) when the RTT-based mapping policy was used at router R_1. It can be noticed that while *cycle_average_rtt* remains relatively constant on *Path*1, it increases on *Path*0 between seconds 50-360 as the additional 448 TCP flows create an increased load on link $R_2 - R_4$ (remember, the 448 TCP flows start at random times between seconds 40-50 and complete at random times between seconds 320-400). This determines router R_1 to map more multihoming flows on *Path*1 than on *Path*0 between seconds 70-380; this is visible in Fig. 8. A similar result is seen in Fig. 7 for the *cycle_average_throughput* metric used when the Throughput-based mapping policy was used at router R_1: *cycle_average_throughput* drops on *Path*0 between seconds 60-380 due to the increased load on this path created by the 448 TCP flows. Consequently, more multihoming flows are mapped by R_1 on *Path*1 in this time interval (see Fig. 9).

Figures 10 and 12 show the same type of results as in figures 6 and 8, but this time RED queuing was used at routers (not DropTail). Similarly, for the Throughput-based mapping policy, we see in figures 11 and 13 the evolution of the *cycle_average_throughput* metric and, respectively, the flow mapping caused by this evolution at router R_1 when the RED queuing policy was employed at all routers.

Next, 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 3 experiments: one when an

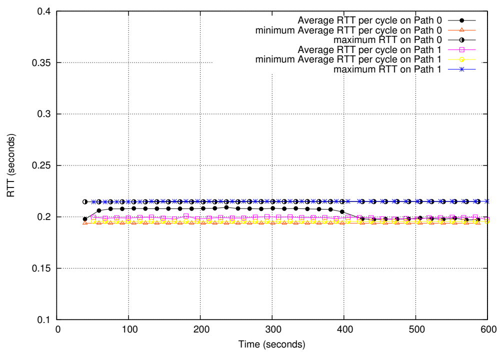

Figure 6: The *cycle_average_rtt* (i.e. average RTT per cycle) for both paths when the RTT-based mapping policy is used, DropTail queuing and 100Mbps capacity; the minimum and maximum values for the *cycle_average_rtt* across the simulation are also depicted

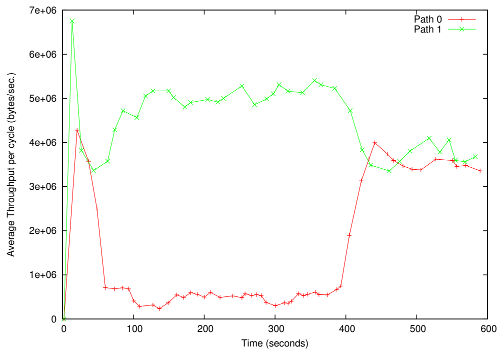

Figure 7: The *cycle_average_throughput* (i.e. average throughput per cycle) for each path when the Throughput-based mapping policy is used, DropTail queuing and 100Mbps capacity

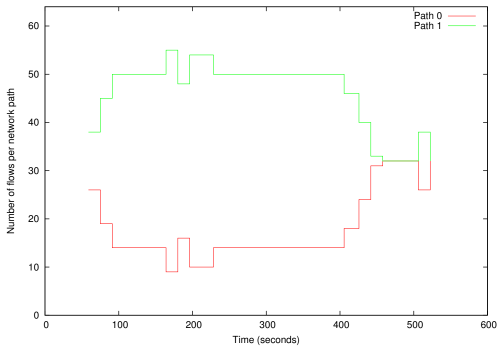

Figure 8: The number of multihoming flows mapped on each path, when the RTT-based mapping policy is used, DropTail queuing and 100Mbps capacity

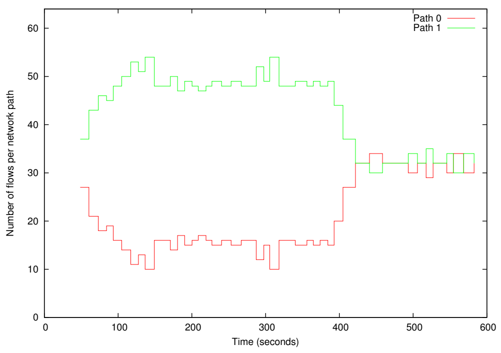

Figure 9: The number of multihoming flows mapped on each path, when the Throughput-based mapping policy is used, DropTail queuing and 100Mbps capacity

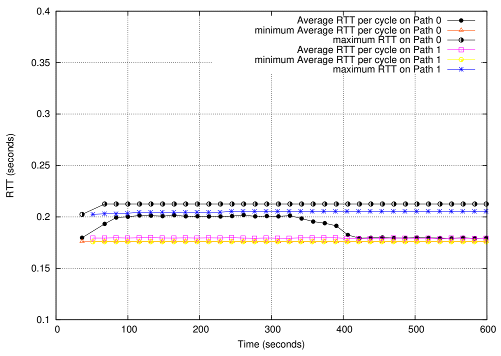

Figure 10: The *cycle_average_rtt* (i.e. average RTT per cycle) for both paths when the RTT-based mapping policy is used, RED queuing and 100Mbps capacity; the minimum and maximum values for the *cycle_average_rtt* across the simulation are also depicted

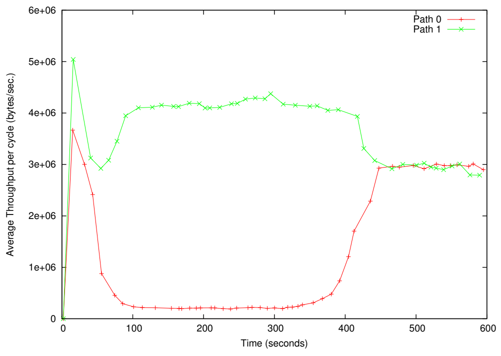

Figure 11: The *cycle_average_throughput* (i.e. average throughput per cycle) for each path when the Throughput-based mapping policy is used, RED queuing and 100Mbps capacity

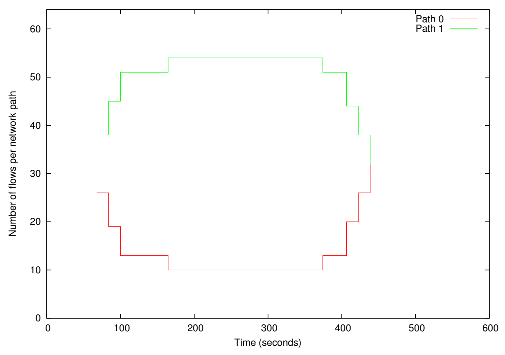

Figure 12: The number of multihoming flows mapped on each path, when the RTT-based mapping policy is used, RED queuing and 100Mbps capacity

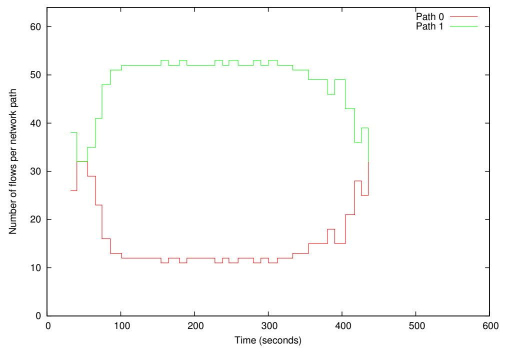

Figure 13: The number of multihoming flows mapped on each path, when the Throughput-based mapping policy is used, RED queuing and 100Mbps capacity

ECMP routing mechanism was used at router R_1, other when our bandwidth aggregation mechanism was used for router R_1 with the RTT-based policy and another one when our bandwidth aggregation mechanism was used for router R_1 with the Throughput-based policy. Each experiment consisted of a simulation being run 10 times with different, randomly generated, flow starting and ending times (for all TCP flows, multihoming and not multihoming) 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 obtained results are depicted in Table 1 for the DropTail queuing discipline and, respectively, in Table 2 for the RED queuing discipline. For the ECMP routing mechanism (i.e. ECMP mapping) we show the absolute values for the three metrics used: $\frac{Min\_throughput}{Max\_throughput}$ (denoted by *MMR* in the tables), *Standard deviation of the flow throughput values* (denoted by *STD* in the tables) and *Average throughput per flow* (denoted by *AVGT* in the tables), but for the bandwidth aggregation mechanism employed (i.e. RTT-based mapping and Throughput-based mapping) we show percentage improvement values for the metrics with respect to the corresponding metric used in ECMP mapping. For example, in Table 1, for a network capacity of 100Mbps, when the ECMP mapping is used, we obtained the metric values $MMR = 0.12$, $STD = 44018.7$ and $AVGT = 61591$. For the same capacity, when the Throughput-based mapping policy was used, we obtained a MMR metric

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	MMR: 0.12 STD: 44018.7 AVGT: 61591	MMR: 0.06 STD: 371675 AVGT: 435682	MMR: 0.06 STD: 836406 AVGT: 974754
RTT-based mapping (%)	MMR: 14.45 STD: 34.02 AVGT: 26.53	MMR: 11.51 STD: 31.73 AVGT: 13.86	MMR: 68.81 STD: 44.07 AVGT: 10.28
Throughput-based mapping (%)	MMR: 109.10 STD: 54.36 AVGT: 29.24	MMR: 432.73 STD: 69.10 AVGT: 24.08	MMR: 586.41 STD: 74.79 AVGT: 17.85

Table 1: Results for DropTail queuing, identical bandwidth and delay for both uplinks

that is 109.10 % better than the MMR obtained by the ECMP mapping policy for the network capacity of 100Mbps. That is, the absolute value of the MMR for the Throughput-based mapping policy, 100Mbps network capacity, is $0.12 \cdot \frac{100+109.10}{100} = 0.25$. Similarly, in Table 1 an $AVGT = 29.24$ value for the Throughput-based mapping policy when capacity is 100Mbps is a percentage improvement over ECMP mapping and results to an absolute value of $61591 \cdot \frac{129.24}{100} = 79452.39bps$. Things are a little bit different for the STD metric, because an improvement of this metric is actually a decrease of the metric. So, the $STD : 54.36$ value in Table 1, for the Throughput-based mapping, 100Mbps capacity, results in the absolute value $44018.7 \cdot \frac{100-54.36}{100} = 20090.13bps$.

Results are very similar for both queuing disciplines DropTail and RED, as seen in Tables 1 and 2. We can see, as expected, that our bandwidth aggregation mechanism, either RTT-based or Throughput-based, improved all three metrics with respect to ECMP routing, in all tested network capacities and queuing disciplines. We can see that as the network capacity increases, generally, the MMR and STD metric improvements are consolidated while the improvement in $AVGT$ decreases (in percentage values as depicted in the tables, but the improvement of $AVGT$ in absolute values is still consolidated). The throughput gain when using our bandwidth aggregation mechanism is between 5% and 30%, while the MMR and STD gains are much higher, sometimes more than 400%. We can also see in these tables that the throughput obtained for the DropTail queuing discipline was larger than the one obtained by RED. Hence the $AVGT$ gains obtained for the bandwidth aggregation mechanism were larger for DropTail than for RED.

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 the path $R_1-R_2-R_4$ that was more than 1.5 times the RTT on the network path $R_1-R_3-R_4$ (this is visible in Figure 14). As usual, for each (network capacity - queuing policy) combination we ran 3 experiments: one where an ECMP routing mechanism was used at router R_1, other when our bandwidth aggregation mechanism was used for router R_1 with the RTT-based policy and another one when

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	MMR: 0.12 STD: 45748.9 AVGT: 62136	MMR: 0.08 STD: 348482 AVGT: 430274	MMR: 0.07 STD: 818536 AVGT: 1004900
RTT-based mapping (%)	MMR: 22.88 STD: 36.34 AVGT: 21.04	MMR: 27.24 STD: 31.74 AVGT: 9.96	MMR: 47.24 STD: 40.94 AVGT: 5.57
Throughput-based mapping (%)	MMR: 151.26 STD: 61.87 AVGT: 24.04	MMR: 258.01 STD: 65.79 AVGT: 15.50	MMR: 434.51 STD: 75.47 AVGT: 11.55

Table 2: Results for RED queuing, identical bandwidth and delay for both uplinks

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	MMR: 0.11 STD: 48457.2 AVGT: 69607	MMR: 0.04 STD: 463800 AVGT: 503196	MMR: 0.02 STD: 1272350 AVGT: 1267670
RTT-based mapping (%)	MMR: 27.40 STD: 38.37 AVGT: 22.98	MMR: 29.97 STD: 35.29 AVGT: 15.13	MMR: 48.06 STD: 39.77 AVGT: 7.00
Throughput-based mapping (%)	MMR: 147.98 STD: 56.43 AVGT: 22.47	MMR: 469.17 STD: 61.52 AVGT: 17.97	MMR: 428.98 STD: 58.31 AVGT: 9.31

Table 3: Results for DropTail queuing, identical bandwidth for both uplinks, but asymmetric delays (link $R_2 - R_4$ has an 80ms transmission delay; link $R_3 - R_4$ has a 40ms transmission delay)

our bandwidth aggregation mechanism was used for router R_1 with the Throughput-based policy; one experiment consists of 10 simulations. The obtained results are depicted in Tables 3 and 4 for the DropTail and RED queue policy, respectively. We can see here the same improvements for all three metrics when the bandwidth aggregation mechanism was employed at router R_1 (with both RTT-based mapping and Throughput-based mapping policies), similar to what we have seen in the symmetrical RTT-bandwidth experiments (i.e. Tables 1 and 2). Although, the *AVGT* improvements of the bandwidth aggregation mechanisms are now smaller than the improvements obtained for the symmetrical RTT-bandwidth experiments; this is especially true for the 1Gbps network capacity, when RED queuing policy was used. From these tables we can also see that the Throughput-based mapping was better that the RTT-based mapping with respect to all three metrics.

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 the path $R_1 - R_2 - R_4$ were double the network capacity of links from the other network path, path $R_1 - R_3 - R_4$. The obtained results are depicted in Tables 5 and 6 for the DropTail and RED queue policy, respectively. We can notice here that the Throughput-based mapping is still better than the RTT-based mapping with respect to all three metrics and the *AVGT* gains obtained are larger than the ones obtained in the symmetrical RTT-bandwidth and, respectively, the asymmetrical RTT network setups. Please note

	Network capacity		
	100Mbps	500Mbps	1Gbps
ECMP mapping	MMR: 0.16 STD: 42211.7 AVGT: 64897	MMR: 0.10 STD: 340308 AVGT: 458444	MMR: 0.08 STD: 848286 AVGT: 1092360
RTT-based mapping (%)	MMR: 21.36 STD: 33.24 AVGT: 15.56	MMR: 55.35 STD: 37.06 AVGT: 4.79	MMR: 58.68 STD: 39.75 AVGT: 2.61
Throughput-based mapping (%)	MMR: 107.95 STD: 57.45 AVGT: 16.65	MMR: 210.98 STD: 64.72 AVGT: 8.51	MMR: 233.09 STD: 64.79 AVGT: 5.01

Table 4: Results for RED queuing, identical bandwidth for both uplinks, but asymmetric delays (link $R_2 - R_4$ has an 80ms transmission delay; link $R_3 - R_4$ has a 40ms transmission delay)

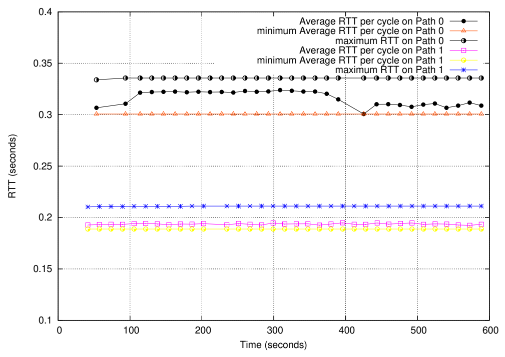

Figure 14: The *cycle_average_rtt* (i.e. average RTT per cycle), minimum and maximum RTT for both network paths when DropTail queuing is used and 1000Mbps network capacity

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 path $R_1 - R_2 - R_4$ by 66% and scaled the weight of path $R_1 - R_3 - R_4$ by 33% (because the network capacity of path $R_1 - R_2 - R_4$ is double the capacity of $R_1 - R_3 - R_4$). 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 the $R_1 - R_2 - R_4$ path and 33% of the flows on the $R_1 - R_3 - R_4$ path.

Next, we chose the identical network capacity and transmission delay setup and ran several tests to see whether small changes in the network capacity influences significantly the results obtained with our bandwidth aggregation mechanism employed. Each inter-router link had a 40ms transmission delay and the capacity is varied (for all inter-router links) from 40 Mbps to 140 Mbps across tests. Each test is run twice, once with the DropTail policy used at the routers and another with RED. We depicted in Figures 15 - 17 the percentage improvement values of the RTT-based and Throughput-based policies for the three metrics used so far, *MMR*, *STD* and *AVGT* metrics with respect to the corresponding metric used in ECMP mapping. In these figures we can see that the RTT-based policy and the Throughput-based policy indeed produce better results than the ECMP mapping strategy and the results are consistent with the tables 1 and 2. Also the results obtained for DropTail queues at the routers were generally better than the results obtained by RED queues. One additional note is that we have run the same experiments using asymmetrical delay times and asymmetrical network capacities for the two network paths and we obtained similar results.

	Network capacity		
	200Mbps/ 100Mbps	250Mbps/ 500Mbps	500Mbps/ 1Gbps
ECMP mapping	MMR: 0.19 STD: 48448.3 AVGT: 69483	MMR: 0.12 STD: 167787 AVGT: 196564	MMR: 0.10 STD: 436776 AVGT: 480950
RTT-based mapping (%)	MMR: 7.02 STD: 22.49 AVGT: 23.57	MMR: 24.72 STD: 29.18 AVGT: 20.09	MMR: 34.55 STD: 38.32 AVGT: 11.99
Throughput-based mapping (%)	MMR: 82.86 STD: 55.98 AVGT: 33.44	MMR: 169.29 STD: 64.50 AVGT: 36.21	MMR: 257.80 STD: 71.04 AVGT: 22.44

Table 5: Results for DropTail queuing, identical transmission delays for both uplinks, but asymmetric bandwidth capacities (path $R_1 - R_2 - R_4$ has the bandwidth capacities: 200Mbps/500Mbps/1Gbps which is twice the bandwidth capacity of the path $R_1 - R_3 - R_4$: 100Mbps/250Mbps/500Mbps)

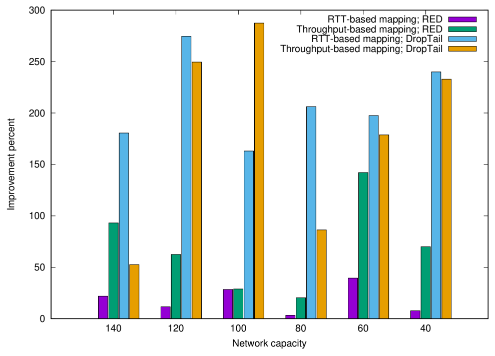

Figure 15: The $\frac{Min\_throughput}{Max\_throughput}$ (i.e. *MMR*) metric improvements of the RTT-based and Throughput-based policies with respect to the corresponding MMR metric obtained by ECMP mapping for different network capacities and queuing policies

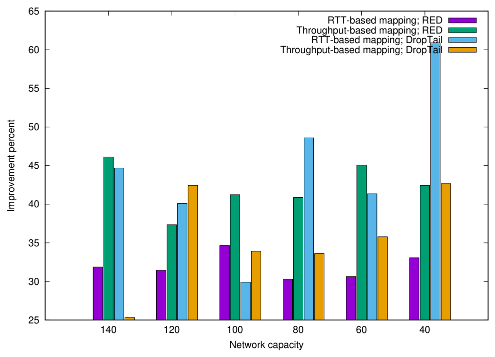

Figure 16: The *Standard deviation of the flow throughput values* (i.e. *STD*) metric improvements of the RTT-based and Throughput-based policies with respect to the corresponding *STD* metric obtained by ECMP mapping for different network capacities and queuing policies

	Network capacity		
	200Mbps/ 100Mbps	250Mbps/ 500Mbps	500Mbps/ 1Gbps
ECMP mapping	MMR: 0.22 STD: 41387.5 AVGT: 63080	MMR: 0.18 STD: 134089 AVGT: 183640	MMR: 0.15 STD: 327732 AVGT: 425396
RTT-based mapping (%)	MMR: 21.74 STD: 24.16 AVGT: 17.33	MMR: 18.91 STD: 24.33 AVGT: 11.82	MMR: 23.27 STD: 29.30 AVGT: 8.93
Throughput-based mapping (%)	MMR: 55.40 STD: 54.08 AVGT: 23.43	MMR: 87.20 STD: 58.18 AVGT: 20.12	MMR: 129.38 STD: 65.44 AVGT: 16.44

Table 6: Results for RED queuing, identical transmission delays for both uplinks, but asymmetric bandwidth capacities (path $R_1-R_2-R_4$ has the bandwidth capacities: 200Mbps/500Mbps/1Gbps which is twice the bandwidth capacity of the path $R_1 - R_3 - R_4$: 100Mbps/250Mbps/500Mbps)

We performed some other set of tests in order to see how a reduced load on the network path $R_1 - R_2 - R_4$ would influence our results. We used the identical network capacity and transmission delay setup (i.e. where all inter-router links have the same network capacity and queuing delay) and ran several tests for the network capacity of 500Mbps and a transmission delay of 40ms. As opposed to the experiments detailed by tables 1 and 2 where we had 512 TCP flows on the link $R_2 - R_4$ creating external load, we now have only 256 TCP flows on the link $R_2 - R_4$. We computed the percent improvements obtained by the RTT-based mapping and Throughput-based mapping policies for the three metrics used (i.e. *MMR*, *STD* and *AVGT*) with respect to the corresponding metric obtained by ECMP mapping in the same network setup an we depicted these in Figures 18 and 19 together with the values obtained by the 512 flows load case (shown in tables 1 and 2). Fig. 18 shows the

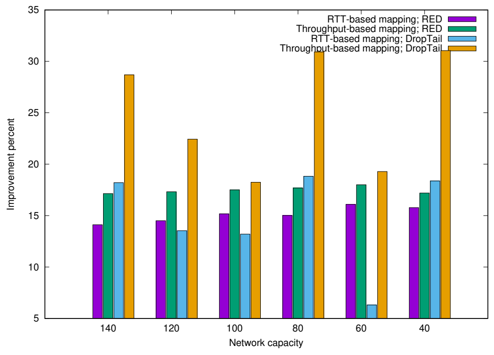

Figure 17: The *Average throughput per flow* (i.e. *AVGT*) metric improvements of the RTT-based and Throughput-based policies with respect to the corresponding *AVGT* metric obtained by ECMP mapping for different network capacities and queuing policies

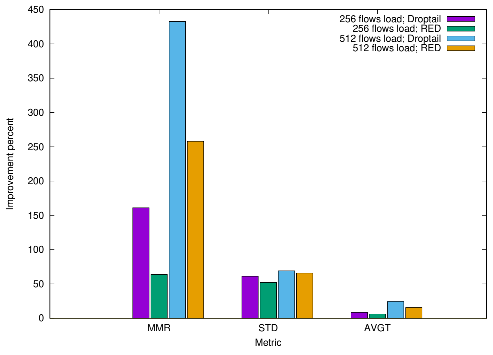

Figure 19: A comparison of the *MMR*, *STD* and *AVGT* metrics improvements for the cases of 256 flows load on the link $R_2 - R_4$ and 512 flows load on the link $R_2 - R_4$. The Throughput-based mapping policy is used at the multihoming router R_1. The values show the improvement in percents of each metric over the metric obtained by ECMP mapping in the same network setup. Network capacity is 500Mbps.

metrics observed by the RTT-based mapping policy while Fig. 19 shows the metric obtained by the Throughput-based mapping policy. We can see in these figures that as the load on link $R_2 - R_4$ drops to 256 flows, the benefits of RTT-based mapping and Throughput-based mapping decrease (with the rare exception of the *MMR* metric for the RTT-based mapping, DropTail queuing and 256 flows load), affecting more severely the RTT-based mapping policy especially when RED queuing is used; sometimes we even obtain setbacks for *MMR* and *AVGT*. This is because for the 256 flows load test, the load on the network path $R_1 - R_2 - R_4$ is not that high to make a RTT difference (the total number of flows on path $R_1 - R_2 - R_4$ is approximately 1.5 times the total number of flows on path $R_1 - R_3 - R_4$) and RED manages to keep the RTT measured on path $R_1 - R_2 - R_4$ approximately equal with the RTT measured on path $R_1 - R_3 - R_4$.

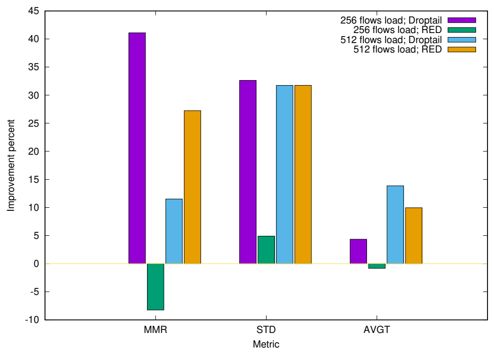

Figure 18: A comparison of the *MMR*, *STD* and *AVGT* metrics improvements for the cases of 256 flows load on the link $R_2 - R_4$ and 512 flows load on the link $R_2 - R_4$. The RTT-based mapping policy is used at the multihoming router R_1. The values show the improvement in percents of each metric over the metric obtained by ECMP mapping in the same network setup. Network capacity is 500Mbps.

Next we reduced the number of multihoming flows from 64 to 32 and performed the tests again. We used the identical network capacity and transmission delay setup (i.e. where all inter-router links have the same network capacity and queuing delay) and ran several tests for the network capacity of 500Mbps and a transmission delay of 40ms. Results are shown in Fig. 20 for the RTT-based mapping policy and, respectively, in Fig. 21 for the Throughput-based mapping policy. We can see that generally the metrics are improved better when the number of multihoming flows is lower (i.e. 32 multihoming flows). We have also tried increasing the number of multihoming flows to 128 and we have obtained similar results.

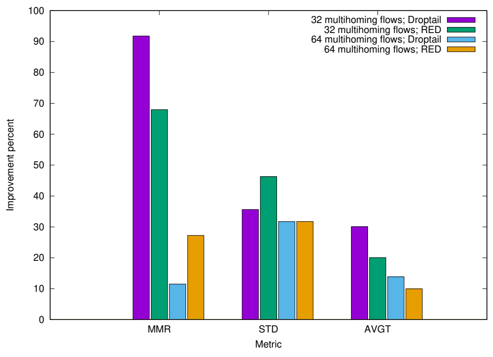

Figure 20: A comparison of the *MMR*, *STD* and *AVGT* metrics improvements for different numbers of multihoming flows: 32 and 64. The RTT-based mapping policy is used at the multihoming router R_1. The values show the improvement in percents of each metric over the metric obtained by ECMP mapping in the same network setup. Network capacity is 500Mbps.

The modified source code of the ns simulator we have used in our tests, together with the scripts necessary for plotting the figures from the paper (and many more) are available in [**?**].

6. Conclusions and Future Work

We have presented in the previous sections 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. 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

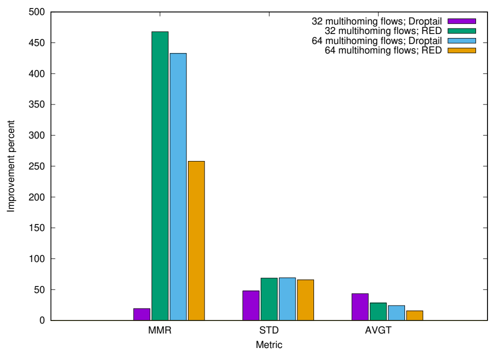

Figure 21: A comparison of the *MMR*, *STD* and *AVGT* metrics improvements for different numbers of multihoming flows: 32 and 64. The Throughput-based mapping policy is used at the multihoming router R_1. The values show the improvement in percents of each metric over the metric obtained by ECMP mapping in the same network setup. Network capacity is 500Mbps.

higher load receives fewer local multihoming flows than a network path with a light load. We used two different strategies in order to evaluate the load/congestion level on a network path: one is based on RTT sampling the return TCP ACK packets and the other one is based on measuring the effective aggregated throughput of all multihoming flows on that 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 both strategies in a simulated network and we have compared them with a classical ECMP-based solution and showed that our bandwidth aggregation routing using either of the two strategies performs better than the ECMP routing solution in terms of total aggregated throughput and fairness between multihoming flows. Even though the Throughput-based mapping proved superior to the RTT-based mapping in almost all test scenarios, the RTT-based mapping strategy should have an important advantage in that in opposition to the Throughput-based mapping strategy, it does not rely on the fact that all or most local multihoming flows are greedy flows (i.e. they always have data to send).

As future plans, we would like to evaluate the performance of both mapping strategies not only with greedy TCP flows, but also with self-limiting flows. Also we would like to study how two such multihoming routing solutions would interact with each other. More specifically, in our test setup we had a multihoming sender router at site A that maps local multihoming flows on the two outgoing network paths and on site B we had a simple router that maps outgoing packets on the same path it received packets from the same flow; so we can say that site B's router is inactive as it does not change the flow mappings set by site A's router. But we can consider a setup in which site B also has a multihoming sender router that maps flows on outgoing paths based on the load it measures on the two outgoing paths (just like site A's router does). Although in theory, both routers, from site A and site B, should map the same number of flows on a network path, they still need to negotiate when a flow is moved from one path to the other. This negotiation may come in the form of a state freeze for some flows, e.g. when a router detects that the other router has just moved a flow from one path to the other, it can not remap this flow on some other path for a fixed time interval. Another future direction would be to test the multihoming routing solution with more than two outgoing network links. In the paper we performed tests using the most common network setup with two outgoing multihoming links.

References

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*. In Proceedings of the 2014 ACM Conference on SIGCOMM, pp. 503514, New York, NY, USA, 2014.

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

[3] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. *Let it flow: resilient asymmetric load balancing with flowlet switching*. In Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17). Berkeley, CA, USA, pp.407-420, 2017.

[4] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. *Clove: Congestion-Aware Load Balancing at the Virtual Edge*. In Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '17). New York, NY, USA, pp.323-335, 2017.

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

[6] P. Merindol, J.J. Pansiot, S. Cateloin, *Improving Load Balancing with Multipath Routing*. In Proceedings of 17th International Conference on Computer Communications and Networks, USA, August 2008.

[7] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg. *COPE: traffic engineering in dynamic networks*. In Proceedings of the 2006 conference on SIGCOMM, New York, NY, USA, pp.99-110, 2006.

[8] D. Applegate and E. Cohen. *Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs*, In Proceedings of the 2003 conference on SIGCOMM, New York, NY, USA, pp.313-324, 2003.

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

[10] B. Fortz and M. Thorup. *Internet Traffic Engineering by Optimizing OSPF Weights*, In Proceedings of IEEE Infocomm, Israel, 2000.

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

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

[13] J. Domzal, Z. Dulinski, M. Kantor, J. Rzasa, R. Stankiewicz, K. Wajda, and R. Wojcik. *A survey on methods to provide multipath transmission in wired packet networks*, In Computer Networks, Volume 77, pp.18-41, 2015.

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

[15] Y. Li, Y. Zhang, L. L. Qiu, and S. Lam. *SmartTunnel: Achieving Reliability in the Internet*. In Proceedings of the IEEE INFOCOM 2007, Washington, DC, USA, 830-838, 2007.

[16] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. *Improving datacenter performance and robustness with multipath TCP*, In Proceedings of the ACM SIGCOMM 2011 conference. ACM, New York, NY, USA, pp. 266-277, 2011.

[17] 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, 2006.

[18] L. Budzisz, J. Garcia, A. Brunstrom, and R. Ferrus. *A taxonomy and survey of SCTP research*. In ACM Computing Surveys Vol. 44, Issue 4, 2012.

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

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

[21] H.-Y. Hsieh, R. Sivakumar. *A Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-Homed Mobile Hosts*. In Wireless Networks, No. 11, pp.99114, 2005.

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

[23] E. Arslan, B. Ross, and T. Kosar. *Dynamic Protocol Tuning Algorithms for High Performance Data Transfers*. In Proceedings of the European Conference on Parallel Processing, Springer, pp.725-736, 2013.

[24] E. Arslan, K. Guner, and T. Kosar. *HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-time Probing*. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, USA, 2016.

[25] T. Kosar, E. Arslan, B. Ross, and B. Zhang. *StorkCloud: data transfer scheduling and optimization as a service*. In Proceedings of the 4th ACM workshop on Scientific cloud computing. New York, NY, USA, pp.29-36. 2013.

[26] 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*. In Proceedings of the ACM SIGCOMM 2013 conference. New York, NY, USA, pp. 3-14, 2013.

[27] N. Gvozdiev, B. Karp, and M. Handley. *FUBAR: Flow Utility Based Routing*. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII). New York, NY, USA, pp. 12-18, 2014.

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

[29] L. Brakmo, S. OMalley, and L. Peterson. *TCP Vegas: New techniques forcongestion detection and avoidance*. In Proceedings of the ACM SIGCOMM conference, 1994.

[30] S. Floyd , and V. Jacobson. *Random Early Detection (RED) gateways for Congestion Avoidance*. In IEEE/ACM Transactions on Networking, Vol. 1, No. 4, pp. 397413, 1993.

[31] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. *Analysis and Design of the Google Congestion Controlfor Web Real-time Communication (WebRTC)*. In Proceedings of the ACM Multimedia Systems Conference,Klagenfurt, Austria, May 2016.