Revisiting Congestion Control for Multipath TCP with Shared Bottleneck Detection

Simone Ferlin, Özgü Alay, Thomas Dreibholz
Simula Research Laboratory
{ferlin,ozgu,dreibh}@simula.no

David A. Hayes, Michael Welzl
Universitetet i Oslo, Institutt for informatikk
{davidhay,michawe}@ifi.uio.no

***Abstract*—Multipath TCP (MPTCP) enables the simultaneous use of multiple links for bandwidth aggregation, better resource utilization and improved reliability. Its coupled congestion control intends to reap the increased bandwidth of multiple links, while avoiding being more aggressive than regular TCP flows on every used link. We argue that this leads to a very conservative behavior when paths do not share a bottleneck. Therefore, in this paper, we first quantify the *penalty* of the coupled congestion control for links that do not share a bottleneck. Then, in order to overcome this penalty, we design and implement a practical shared bottleneck detection (SBD) algorithm for MPTCP, namely MPTCP-SBD. Through extensive emulations, we show that MPTCP-SBD outperforms all currently deployed MPTCP coupled congestion controls by accurately detecting bottlenecks. For the non-shared bottleneck scenario, we observe throughput gains of up to 40% with two subflows and the gains increase significantly as the number of subflows increase, reaching more than 100% for five subflows. Furthermore, for the shared bottleneck scenario, we show that MPTCP-SBD remains fair to TCP. We complement the emulation results with real-network experiments justifying its safeness for deployment.**

Keywords: Multipath TCP, MPTCP, Shared Bottleneck Detection, Congestion Control, Coupled Congestion Control.

I. Introduction

When the Transmission Control Protocol (TCP) and the Internet Protocol (IP) were first specified, Internet hosts were typically connected to the Internet via a single network interface and TCP was built around the notion of a single connection between them. Nowadays the picture is rather different with the rapid increase in the number of hosts with more than one network interface (multi-homed). For example, mobile devices today often accommodate multiple wireless technologies (e.g. 3G/4G and WiFi). Standard TCP is not able to efficiently explore the multi-connected infrastructure as it ties applications to network interface specific source and destination IP addresses.

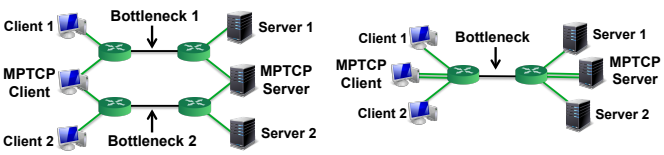
Multipath TCP (MPTCP) closes the gap between networks with multiple paths between destinations and TCP's single-path transport by allowing the use of multiple network paths for a single data stream. This provides potential for a higher overall throughput and ensures application resilience if some network paths suffer performance degradation or failure. MPTCP is designed to look like regular TCP from the network's perspective making it deployable in today's Internet. The Internet Engineering Task Force's (IETF) Multipath TCP working group continues MPTCP's development and standardisation. Available implementations include: Linux distributions (Debian, Ubuntu, etc.), FreeBSD, iOS, Mac OS and Yosemite. Current commercial deployments do not exploit MPTCP's full capabilities, instead tending to only take advantage of MPTCP's resilience characteristics.

Three goals capture the desired operation of MPTCP [1]:

1) *Improve Throughput*: A multipath flow should perform at least as well as a single path flow would on the best of the paths available to it.
2) *Do Not Harm*: A multipath flow should not take more from any of the resources shared by its different paths than if it was a single path flow.
3) *Balance Congestion*: A multipath flow should move as much traffic as possible off its most congested paths, subject to meeting the first two goals.

The first goal is the primary incentive behind the design of MPTCP [2], while the second design goal guarantees fairness at the bottleneck. The third goal addresses the resource pooling principle [3]. Focusing on the first two goals, we target the multipath congestion control mechanism that determines how network resources should be shared efficiently and fairly between competing flows at shared bottlenecks. Different coupled congestion control algorithms have been proposed for MPTCP in the literature [1], [4]–[7]. However, due to the lack of practical Shared Bottleneck Detection (SBD) mechanisms, the design of these algorithms always assumed shared bottlenecks, resulting in sub-optimal performance even when this is not the case.

This paper proposes a dynamic coupled congestion control for MPTCP with a practical shared bottleneck detection mechanism, namely, MPTCP-SBD. The proposed MPTCP-SBD congestion control algorithm can dynamically decouple subflows that are not sharing bottlenecks, unlocking the full throughput potential of the links. When there is a shared bottleneck, MPTCP-SBD keeps the subflows coupled, remaining fair to competing TCP connections. We designed MPTCP-SBD as a light-weight extension to standard MPTCP and implemented it into MPTCP v0.89.5. Through extensive emulations, we show that MPTCP-SBD can accurately detect bottlenecks. For the non-shared bottleneck scenarios, we observe throughput gains of up to 40% with two subflows and the gains increase significantly as the number of subflows increases, reaching gain values of more than 100% with five subflows. While achieving such performance gains in non-shared bottleneck, we show that MPTCP-SBD remains fair to TCP in shared bottlenecks. We also confirm the robustness of the algorithm



(a) Non-Shared Bottleneck (NSB) (b) Shared Bottleneck (SB)

Figure 1. Bottleneck Scenarios

by showing how it adapts to shifting bottlenecks. The emulation results are complemented by real-network experiments that justifies the effectiveness of the algorithm and shows its safeness for deployment.

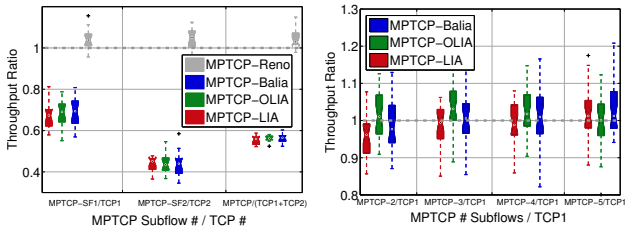
II. Motivation

In order to motivate the need for MPTCP-SBD, this section highlights the shortcomings of current MPTCP congestion control algorithms with respect to their interaction with bottlenecks. Specifically, we looked at Linux implementations of: uncoupled congestion control where each link applies standard TCP congestion control (we refer to as Reno[1]) [8], [9], Linked Increase Adaptation (LIA) [10], Optimized Linked Increase Adaptation (OLIA) [11] and Balanced Link Adaptation (Balia) [12]. We evaluated two scenarios from [13]: *Non-Shared Bottleneck* (NSB) and *Shared Bottleneck* (SB).

In Figure 1(a), we illustrate the NSB scenario where two disjoint paths have distinct bottlenecks. We measured MPTCP with two subflows (SF), MPTCP-SF1 and MPTCP-SF2, where each subflow is sent over a path. On each path, we also had a regular TCP flow, TCP1 and TCP2, each competing against the corresponding MPTCP subflow. The SB scenario is illustrated in Figure 1(b), where there is a single shared bottleneck through which all MPTCP subflows (MPTCP-N where N indicates the number of subflows) and one regular TCP flow, TCP1, are sent. The details for traffic generation and measurement setup are described in Section IV.

For the NSB scenario, we studied relative performance of congestion control algorithms in terms of the individual throughput ratio (i.e. the ratio of each MPTCP subflow with respect to its corresponding TCP flow) as well as the sum throughput ratio (i.e. the ratio of all MPTCP subflows to the sum of the TCP flows). Figure 2(a) shows the comparative results for the 4 different tested congestion control algorithms. We observe that all coupled congestion controls (LIA, OLIA and Balia) have similar performances, where the throughput ratio is around 0.70 for MPTCP SubFlow 1 (MPTCP-SF1), 0.45 for MPTCP SubFlow 2 (MPTCP-SF2) and close to 0.6 for the combined result. However, the Reno uncoupled congestion control (i.e. both subflows run regular TCP) provides a ratio that is close to 1, which is the desirable result for this particular scenario. Therefore, coupled congestion control in a NSB scenario results in more than 40% drop in the overall throughput when compared to uncoupled congestion control. Further, we observe that as the number of NSB subflows increases, the

[1]Reno refers to congestion control including NewReno and SACK as implemented in Linux, kernel version 3.14.33.



(a) Non-Shared Bottleneck (NSB) (b) Shared Bottleneck (SB)

Figure 2. MPTCP Performance in NSB and SB Scenarios with synthetic background traffic expressed as ratio of MPTCP to TCP flow(s). Boxes span the 25^{th} to 75^{th} percentile, with a notch at the median and whiskers extending to the lesser of the extreme point or 1.5 times the interquartile range.

penalty in the total throughput increases, reaching more than 60% penalty (Figure 5(b)) for the 5 subflow case.

Figure 2(b) shows the relative performance of the different MPTCP congestion control algorithms for the SB scenario. We show the ratio of the sum of all MPTCP subflows to competing TCP at the shared bottleneck for different number of subflows. We observed here that MPTCP to TCP is close to 1 for all MPTCP coupled congestion control algorithms.

Remark 1: The Linux kernel has *packet* as the unit for both congestion window ($cwnd$) and slow-start (SS) threshold ($ssthresh$). Reno defines $min\_ssthresh$=2, and all Linux congestion controls are implemented to enter SS when $cwnd \leq ssthresh$. In MPTCP-OLIA and Balia specifications [11], [12], $min\_ssthresh$=1 is required for connections that have at least 2 subflows. However, we found out that this was not honored in the Linux implementation of MPTCP-OLIA. Based on our preliminary analysis, we observed that MPTCP-OLIA assigns different rates to the subflows that share a bottleneck. MPTCP-OLIA with $min\_ssthresh$=2 enters SS more frequently compared to $min\_ssthresh$=1, especially for a larger number of subflows, adversely increasing aggressiveness of MPTCP-OLIA. Therefore, in this paper, all figures are obtained for $min\_ssthresh$=1 following the specification.

Summary and discussion of findings: Our analysis shows that MPTCP's coupled congestion controls are fair to TCP in the SB scenario, but do not achieve their fair portion of the link capacity in the NSB scenario. Based on these results, we argue that current coupled congestion controls are unnecessarily conservative when there is no shared bottleneck. The incorporation of a shared bottleneck detection algorithm in MPTCP can help MPTCP to get its fair share of the link capacity for non-shared bottlenecks while ensuring fairness to TCP in shared bottlenecks.

III. System Design and Implementation

The achievable throughput for MPTCP, when there is no shared bottleneck, is limited as a result of MPTCP's congestion control design goal 2 (see Section I). This design goal forces a fairness notion that includes two separate aspects:

1) Fairness to TCP if flows share a bottleneck with it.
2) Fairness such that resource use is limited to the amount of resources that would be used by a single flow on one of

the paths (called "fairness in the broader, network sense" in RFC6356 [10]).

These two aspects are intrinsically bound in the coupled congestion controls of MPTCP. As we will see, shared bottleneck detection makes it possible to untie them, so that we can support the first aspect without necessarily ensuring the second. Our intention is not to advocate one particular fairness notion, but to provide another degree of freedom by *enabling* the separation of the two aspects above.

Our goal in this paper is to decouple subflows if they do not traverse a shared bottleneck, so that they can behave as regular TCP and achieve their fair share on their respective bottlenecks. For the subflows sharing a bottleneck, we maintain MPTCP's default coupled congestion control, MPTCP-OLIA, as it was shown to be fair to regular TCP, see Figure 2(b) and [6]. In order to achieve this, we implement a practical shared bottleneck detection algorithm for MPTCP. In the following subsections, we describe the SBD algorithm and detail the sender and receiver side implementation together with the signalling procedure.

A. Shared Bottleneck Detection Algorithm

The current Internet is unable to explicitly inform hosts about which flows share bottlenecks. Instead, they need to infer this information from packet loss and delay. Since MPTCP uses packet loss to indicate congestion, it seems to be a natural choice for bottleneck detection. However, it is relatively rare signal in a well-operating network (e.g. $<3\%$), and often not well correlated across flows sharing the bottleneck. Packet delay provides a frequent, but noisy signal. Packets traversing a common bottleneck will encounter quite widely varying bottleneck queue lengths, with this bottleneck induced delay being further perturbed by every other device along the path. This along with differing path lags makes it difficult to correlate the delay of different flows.

Recently, a way of addressing these issues is proposed using the distribution of packet delay measurements and grouping flows that have similar statistical characteristics [14], [15]. Three key summary statistics are used as a basis for grouping flows that share a common bottleneck and One-Way Delay (OWD) is used as the base measurement. Although Round Trip Time (RTT) is easier to measure, it includes noise introduced by every device on the return path to the bottleneck delay signal. Therefore, using OWD potentially removes up to half of the path noise from the delay signal. Since these statistics are calculated with respect to the mean OWD, only the relative OWD is required – meaning that sender and receiver clocks do not need to be synchronised. Although this mechanism is used with RTP media, we believe that the algorithm is general enough to apply to MPTCP.

We base our MPTCP-SBD algorithm implementation on the specification and parametrisation from [15], since it includes improvements to original algorithm in [14]. The algorithm is based on three key statistics: skewness, variability, and key frequency. Skewness is estimated by counting the number of OWD measurements above and below the previous mean. Mean Absolute Deviation (MAD) is used to quantify the variability. A key frequency characteristic is quantified by counting and normalising the number of times a short-term mean OWD significantly crosses a longer-term mean OWD. Thus, it is a measure of the low-frequency oscillation of OWDs at the bottleneck. We follow [15] calculating these statistics as:

$$\overline{\mathrm{OWD}}_n = \frac{\sum_{c=1}^{C_n} \mathrm{OWD}_c}{C_n} \tag{1}$$

where c identifies a particular OWD measurement over the interval T, C_n represents n th stored number of OWD measurements in T, the base time interval.

$$\overline{\mathbf{OWD}} = \frac{\sum_{n=1}^{N} \overline{\mathrm{OWD}}_n}{N} \tag{2}$$

where N is the number of stored base statistics

$$\mathbf{skew\_est} = \frac{\sum_{n=1}^{N} \mathrm{skew\_base}_n}{\sum_{n=1}^{N} C_n} \tag{3}$$

where

$$\begin{aligned} \mathrm{skew\_base}_n = &\sum_{c=1}^{C_n} \left[\mathrm{OWD}_c < \overline{\mathrm{OWD}}\right] \\ &- \sum_{c=1}^{C_n} \left[\mathrm{OWD}_c > \overline{\mathrm{OWD}}\right] \end{aligned} \tag{4}$$

$$\mathbf{var\_est} = \frac{\sum_{n=1}^{N} \mathrm{var\_base}_n}{\sum_{n=1}^{N} C_n} \tag{5}$$

where

$$\mathrm{var\_base}_n = \sum_{c=1}^{C_n} \left|\mathrm{OWD}_c - \overline{\mathrm{OWD}}_{n-1}\right| \tag{6}$$

$$\mathbf{freq\_est} = \frac{\mathrm{number\_of\_crossings}}{N} \tag{7}$$

where number_of_crossings is a count of $\overline{\mathrm{OWD}}_n$ values that cross $\overline{\mathrm{OWD}}$ by more than $p_v\overline{\mathrm{OWD}}$. Note that base calculations are made for a number of statistics over each T. The implementation calculates the statistics incrementally with a cyclic buffer of N base entries ($n = 1$ is the most recent).

Flows where Skew_est (Eq. (3)), var_est (Eq. (5)) and freq_est (Eq. (7)) have similar values (within a certain threshold), are grouped together. Flows that are grouped together are deemed to be sharing a common bottleneck.

Flows are grouped according to the simple grouping algorithm outlined in [15]. Key to this algorithm's operation is to only attempt to group flows that are traversing a bottleneck (i.e. a congested link), since the summary statistics of flows not traversing a bottleneck are really only a measure of the path noise. The algorithm does this by only choosing flows whose estimate of skewness or high packet loss indicates that they are traversing a bottleneck.

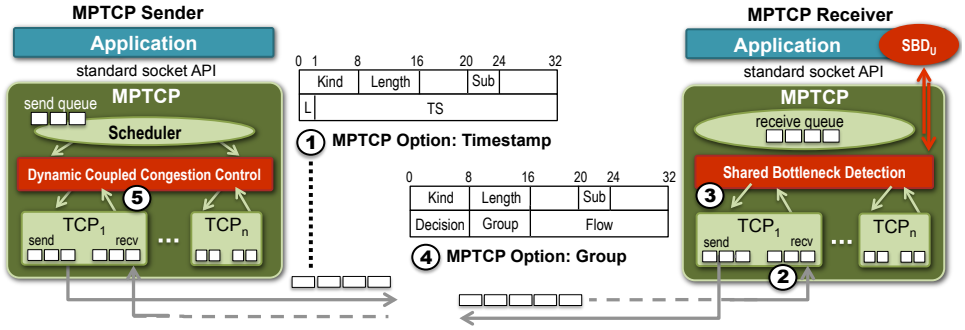


Figure 3. MPTCP-SBD Design: The MPTCP sender transmits local timestamps with 31 Bits precision in the MPTCP TS option in each subflow's packet ①. The receiver extracts each timestamp after the stack's basic header checksum and calculates the relative OWD ②, which is the input for SBD ③ (part of the SBD algorithm is implemented in user-space, SBD_U). The receiver feeds back the SBD information ④ to the sender, which uses the information to couple flows in the same group and decouple those in different groups ⑤.

B. MPTCP-SBD Implementation

This section presents MPTCP-SBD's design and integration into MPTCP. Our system design is depicted in Figure 3 and has two main components as highlighted in red: (i) shared bottleneck detection mechanism at the receiver (③) and (ii) dynamic congestion control mechanism at the sender (⑤). For the signalling between the sender and the receiver, MPTCP options (①, ④) are used. One-way delay computations are carried out at the receiver (②). The implementation is available for Linux MPTCP v0.89.5 based on kernel v3.14.33.

Referring to Figure 3, the elements are now presented:

① ***MPTCP Timestamp:*** Since the standard TCP time stamp option [16] precision is too low in the Linux (v3.14.33) TCP implementation, we introduced a new MPTCP Time Stamp (TS) option containing a time stamp with microsecond precision. It is carried by every data segment leaving the sender[2]. The sender fills this field immediately prior to passing the packet down to the IP layer.

Packet loss is used to supplement skewness at extreme loads [15]. We reserve a flag in the timestamp option in order to transmit loss counts incrementally.

② ***One Way Delay Computation at the Receiver:*** The relative OWD is calculated by subtracting the arriving MPTCP TS from the host's version of current time. This calculation is done right after the basic header checksum to avoid extra delays, introduced by host. These are the input values for the statistics in Eqs. (3), (5) and (7), which are updated for the current interval T. The kernel stores the statistics for the last N intervals. At the end of an interval, the statistics can be retrieved by the SBD decision mechanism (③).

③ ***SBD Decisions:*** After every $T = 350\,\text{ms}$, the SBD mechanism runs with OWD statistics collected over the last N intervals. The result of every SBD run is: i) a list of non-congested flows, and ii) a list of flows grouped by common shared bottleneck. We refer to this set of flow state and groupings as an *SBD observation*. Every SBD observation could be sent to the sender directly. However, we noticed that flows which were first grouped together can occasionally be separated. This could be due to an erred SBD grouping decision or a fluctuating bottleneck, i.e. temporarily congestion abated so there was no bottleneck for a time. To avoid having the sender reacting to such transient grouping changes, the mechanism collects 10 observations and decides that flows share a stable bottleneck if the majority of the observations are consistent (i.e. the same observation was made at least 5 times), see Subsection IV-B. We refer to the result of this observation filter as an *SBD decision*. A decision is transmitted to the sender every 3.5 s.

Parts of SBD are implemented in user-space (SBD_U) to aid experimentation[3]. The parts that are currently implemented in user-space are: Equation 7, grouping and decision making. All other steps are implemented in the kernel.

④ ***SBD Signalling:*** Grouping information is transmitted in the form of a vector that maps each flow to a group identifier. Non-congested flows have a reserved group ID. This information is conveyed in the ACKs via an MPTCP option containing the group ID, flow ID, and decision ID. The flow ID allows a subflow to inform the sender not only about itself, but also about others in a round-robin manner. The decision ID allows the sender to detect when a new complete decision was received. Note that a decision remains valid until it is updated by the sender.

⑤ ***Dynamic Coupled Congestion Controller based on SBD:*** The sender, using *SBD decision* feedback, decouples subflows that are in distinct bottlenecks during congestion avoidance. We develop MPTCP-SBD based on MPTCP's default coupled congestion control OLIA. Let F be the set of all subflows, $M \subseteq F$ the subset of flows with maximum `cwnd`, $B \subseteq F$ the subset of flows that are the *best* subflows based on interloss probability estimation and `cwnd`. OLIA works as follows: When OLIA is called for a subflow $x \in F$, it considers all F building M and B sets. The `cwnd` update for subflow x is calculated based on M and B [11].

The SBD feedback is integrated into OLIA with an additional initial step. Let $S \subseteq F$ be the subset of flows that share a bottleneck with subflow x. First the subset S is built containing all flows that share a bottleneck with x. Subsequent steps follow OLIA, but using S as

[2] [17] was used as reference for the design.

[3] A pure kernel implementation of SBD is already under testing.

Table I
SBD MECHANISM PARAMETERS, FOLLOWING [15]

T (ms)	N	Thresholds							
		c_s	c_h	p_f	p_s	p_v	p_mad	p_d	p_l
350	50	-0.01	0.3	0.1	0.1	0.7	0.1	0.1	0.1

the base set instead of F. Moreover, within the subset S, MPTCP-SBD increases the `cwnd` of the subflows as for single path TCP, and it also performs load-balancing among them, maintaining MPTCP's design goals 1) and 2). When computing the `cwnd` for x, subflows that are in distinct bottlenecks are not considered, and flow x is decoupled from them. Further, if it can be inferred from *SBD decision* that flow x does not share a bottleneck with other subflows, Reno is used, making x behave like a regular TCP flow.

Remark 2:

The current implementation does not cater for relative clock skew in its calculation of the summary statistics. In real-network experiments, the hosts' clock skew may affect SBD. Although the skew is usually not significant over the time intervals of the mechanism, a separate contribution related to this will be added to MPTCP-SBD.

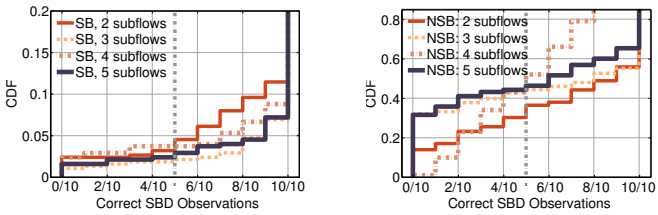
IV. EMULATION EXPERIMENTS

We first assessed MPTCP-SBD's performance on an emulation environment where we have more control and can revalidate the tests performed by [7] and [13].

A. Measurement Setup

We used the CORE network emulator [18] with MPTCP for our emulation experiments. We followed the same values in [7], where for both shared and non-shared bottleneck scenarios, the bottlenecks had 20 Mbps capacity and 20 ms RTT. The droptail bottleneck queue was set to 1 Bandwidth×Delay Product (BDP), about 35 packets in these tests, which allows for 100% link utilization with a congestion control that halves its `cwnd` upon congestion (which MPTCP does just like TCP Reno) [19], [20]. Linux MPTCP v0.89.5 was used with socket buffer size recommended by [21], i.e., buffer = $\sum_{i=1}^{n} \text{bandwidth}_i \times \text{RTT}_{\max} \times 2$. To ensure independence between runs, cached TCP metrics were cleared after each run. We focused on congestion avoidance, discarding the initial connection phase, analyzing 120s of each run.

A synthetic mix of TCP and UDP was generated with D-ITG [22] as background traffic, creating a more realistic emulation environment. The TCP traffic was composed of greedy and rate-limited TCP flows with exponential distributed mean rates of 150 pps. The UDP traffic was composed of flows with exponentially distributed mean rates between 50 and 150 pps and Pareto distributed on and exponentially distributed off times with on/off intervals betwen 1 s and 5 s. Packet sizes varied with a mean of 1000 Bytes and RTT between 20 and 80 ms. We used similar proportions and traffic characteristics from [13], [14] to dimension the background traffic.



(a) SB Scenario (b) NSB Scenario

Figure 4. Cumulative Distribution of *correct* SBD observations within 10 observations SBD window. The gray line shows the decision threshold.

B. SBD Decision Threshold

As mentioned in Section III-B, the SBD mechanism generates one *SBD observation* every $T = 350$ ms and takes a *SBD decision* based on a window of 10 observations (3.5s). The window size can be adjusted to balance delay and stability of decision making. For example, a larger window will increase the delay in reaching a decision, whereas the shorter window may not be able to effectively eliminate transients in the observations. We found a window size of 10 to be a good compromise between these two aspects. A decision is reached based on a simple threshold, counting how often any two flows were observed to be on the same bottleneck. In order to show the effect of the threshold, we ran preliminary experiments. We define a *correct* observation such that the SBD arrives at the *expected*[4] grouping for each case. In other words, a *correct* observation indicates that all subflows belong to the same group for the shared bottleneck scenario, and that all subflows belong to distinct groups for the non-shared bottleneck scenario.

Figures 4(a) and 4(b) show both NSB and SB scenarios, depicting the distribution of *correct* observations within the SBD decision windows. For the SB scenario, we observed that in about 90% of all SBD decision windows, 10/10 observations in the window correctly grouped the subflows into the same group, regardless of the number of subflows. Similarly, for the NSB scenario, we observed that in approximately 45% of all SBD decision windows, 10/10 observations correctly placed the subflows into different groups.

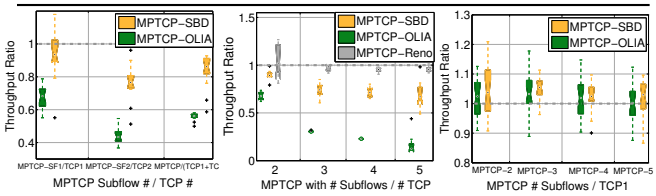
The decision threshold tunes the stability of the final SBD decision. For example, choosing a low threshold favors grouping yielding more conservative behavior. We use a decision threshold such that at least 50% of the observations are consistent (i.e. the same observation was made at least 5 times) where in a tie event, we bias toward grouping. We found that this threshold yields good results in a wide range of scenarios.

Remark 3: Our goal was for the SBD decision to favor coupling to avoid false decisions of decoupling. Note that for the non-shared bottleneck case, an incorrect decision means MPTCP-SBD falling back to MPTCP's normal behaviour.

[4]Note that in our experiments, we generated background traffic in order to create bottlenecks. However, due to the traffic characteristics, the bottleneck may fluctuate during the experiment. Therefore, what we deem the *correct* observation may not reflect the actual link condition in some circumstances.

Table II
SBD Decision ACCURACY IN NSB AND SB SCENARIOS

Scenario	1 Group	2 Groups	3 Groups	4 Groups	5 Groups
NSB-2	0.25	0.75	-	-	-
NSB-3	0	0.26	0.74	-	-
NSB-4	0	0.14	0.15	0.71	-
NSB-5	0	0.02	0.118	0.16	0.702
SB-2	0.97	0.03	-	-	-
SB-3	0.975	0.018	0.003	-	-
SB-4	0.977	0.022	0	0	-
SB-5	0.973	0.024	0.0026	0	0



(a) NSB Scenario (b) 2, 3, 4, and 5 NSBs (c) SB Scenario

Figure 5. MPTCP with and without SBD with 2, 3, 4 and 5 subflows, for NSB and SB Scenarios with synthetic background traffic. Boxes span the 25^{th} to 75^{th} percentile, with a notch at the median and whiskers extending to the lesser of the extreme point or 1.5 times the interquartile range.

C. Results

We assessed the performance of MPTCP-SBD in 5 different scenarios: 1) non-shared bottleneck, 2) shared bottleneck, 3) shifting bottleneck, 4) Active Queue Management (AQM), and 5) subflows with different base-RTTs.

1) Non-Shared Bottleneck (NSB): We evaluated the performance of MPTCP-SBD in terms of *SBD decision* accuracy and throughput gains. The *SBD decision* accuracy is the percentage of *correct* decisions over all decisions in a single experiment, and, subsequently, the average SBD decision accuracy is computed over all measurements. In Table II, we present the average *SBD decision* accuracy. Our results indicates that MPTCP-SBD can detect disjoint bottlenecks correctly in 75% of the cases for 2 subflows and 70% for 5 subflows. In Figure 5(a), we illustrate the throughput gains MPTCP-SBD can achieve by decoupling the subflows. We observe that MPTCP-SBD provides a throughput gain of up to 40% for the 2 subflows case and more than 100% for the 5 subflows case compared to MPTCP-OLIA.

2) Shared Bottleneck (SB): Similarly, for the shared bottleneck case, we evaluated *SBD decision* accuracy and MPTCP throughput. Table II shows that MPTCP-SBD can detect the shared bottleneck with an average accuracy of 97%. This results in MPTCP-SBD performing slightly more aggressive compared to MPTCP-OLIA regardless of the number of subflows (see Figure 5(c)). This aggressiveness is due to the SBD occasionally separating subflows that we expected to be in the same bottleneck group. This separation results in flows acting like independent TCP flows.

3) Shifting Bottleneck: To show how MPTCP-SBD adapts to shifting bottlenecks, we consider a scenario where MPTCP subflows share different bottlenecks at different times. This scenario is illustrated in Figure 6, where we shift the load from bottleneck 3 (e.g. shared bottleneck) to bottlenecks 1 and 2 (e.g. non-shared bottleneck) and then back to bottleneck 3.

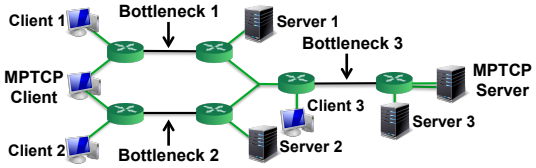


Figure 6. Shifting Bottleneck

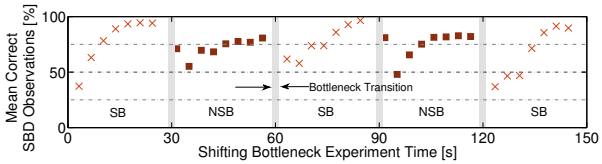


Figure 7. The average percentage of *correct* SBD observation within a SBD decision window in time for the shifting bottleneck scenario.

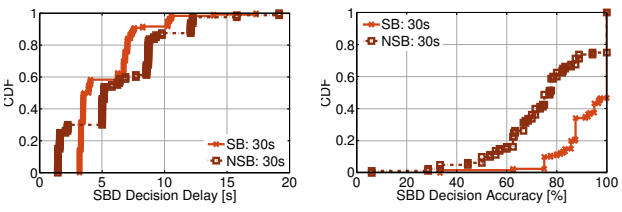
This way, MPTCP-SBD can detect the transition between non-shared and shared bottleneck phases. For the shifting bottleneck scenario, the bottlenecks are loaded for 30s, alternating between SB $\rightarrow$ NSB $\rightarrow$ SB $\rightarrow$ NSB $\rightarrow$ SB. With shifting bottlenecks we set the bottleneck interval to 30 s rather than 320 s as in [13] to show that our mechanism can cope to detect the changes even in shorter shifting periods.

Note that SBD needs $N \times T$ samples to build estimates (i.e. the mechanism requires *memory* to detect a shifting bottleneck) [15]. Figure 7 illustrates the average percentage of *correct* SBD observations within a SBD decision window versus time. Recall that the SBD decision threshold imposes that at least 50% of the SBD observations have to be consistent within one observation window in order to have a decision. This is shown in Figure 7, where the average percentage of correct observations are closer to 50% for the SBD decision windows that come immediately after a bottleneck transition and this percentage increases in time indicating that SBD decisions become more reliable and stable.

We define *transition delay* as the time between the first *expected* SBD decision after a bottleneck transition and illustrate the distribution of the transition delay in Figure 8(a). We observe that SBD had an average delay of 7 s (i.e. around 2 SBD decisions) to detect a transition. After the transition, MPTCP-SBD shows on average 90% accuracy in SB and over 60% accuracy in NSB scenarios as depicted in Figure 8(b).

4) Active Queue Management (AQM): This section shows MPTCP-SBD's performance when the bottleneck queue policy is changed from DropTail to Random Early Detection (RED)[5].

[5]Parameters set according to http://www.icir.org/floyd/REDparameters.txt



(a) Transition Delay (b) Accuracy within Bottleneck

Figure 8. Mean SBD Decision Accuracy and Delay in Shifting Bottleneck

Table III
SBD DECISION ACCURACY WITH RED FOR NON-SHARED BOTTLENECK AND SHARED BOTTLENECK SCENARIOS

Scenario	1 Group	2 Groups	3 Groups	4 Groups	5 Groups
NSB-2	0.321	0.679	-	-	-
NSB-3	0	0.324	0.675	-	-
NSB-4	0.041	0.057	0.267	0.658	-
NSB-5	0	0	0	0.366	0.633
SB-2	0.989	0.010	-	-	-
SB-3	0.973	0.026	0	-	-
SB-4	0.968	0.026	0.053	0	-
SB-5	0.962	0.026	0.016	0	0

RED marks and randomly drops packets when the average queue occupancy exceeds a certain threshold, i.e., not necessarily only when the queue is saturated. This changes the statistical characteristics of the OWD measurements, hence, SBD's input. Table III depicts that MPTCP-SBD is robust when the bottleneck queue is RED, for both shared and non-shared bottleneck scenarios. RED desynchronizes TCP flows making the bottleneck more stable. In the NSB scenario, we observe a slight drop in detection accuracy with a higher number of subflows. This is caused by our conservative configuration of SBD, which prefers grouping over splitting and, therefore, intermittently groups any two flows that are observed as being similar enough. This also means that we cannot always perfectly detect subflows belonging to distinct bottlenecks, however, in the NSB with 5 bottlenecks, MPTCP-SBD has five independent uncoupled flows 63.3% of the cases.

5) Subflows with Different Base-RTTs: So far, we have assumed that the subflows's base RTTs are very similar. In this section, we vary the subflows' base RTT and evaluate the performance of MPTCP-SBD. For both, shared and non-shared bottleneck scenarios, we kept one subflow's RTT fixed at 20ms and changed the other subflows' RTTs. The results and RTT settings are shown in Table IV.

For the shared bottleneck scenario, we observe that having different base RTTs slightly reduces the SBD accuracy (when compared to those with identical base RTTs), especially as the gap between shortest RTT and longest RTT grows. For the non-shared bottleneck scenario, the difference in base RTTs improves detection for NSB with 2 and 3 subflows compared to the scenario where the subflows have the same baseline RTT, see Table II. However, NSB with 4 and 5 subflows keeps similar detection values. This is due to our conservative configuration of SBD, however, further investigation in real non-shared bottleneck setups, is object of future work. This is a very promising result for the non-shared bottleneck scenario since in the real world the subflows' base RTTs are expected to vary due to link and queue perturbations.

V. REAL-NETWORK EXPERIMENTS

The experimental analysis of MPTCP-SBD in real networks is difficult, since the *ground truth* for bottlenecks is not known. We look at the performance of MPTCP-SBD within a topology constructed over NorNet[6] using Virtual Machines (VM) from

[6]NorNet: https://www.nntb.no.

Table IV
SBD DECISION ACCURACY WITH DIFFERENT RTTS FOR NON-SHARED BOTTLENECK AND SHARED BOTTLENECK SCENARIOS

Scenario	RTTs [ms]	1 Group	2 Groups	3 Groups	4 Groups	5 Groups
NSB-2	20, 40	0.153	0.846	-	-	-
NSB-3	20, 30, 40	0.014	0.187	0.798	-	-
NSB-4	20, 30, 40, 50	0.027	0.055	0.198	0.717	-
NSB-5	20, 30, 40, 50, 60	0.021	0.062	0.092	0.181	0.694
SB-2	20, 40	0.9615	0.0384	-	-	-
SB-3	20, 30, 40	0.961	0.033	0.055	-	-
SB-4	20, 30, 40, 50	0.955	0.044	0	0	-
SB-5	20, 30, 40, 50, 60	0.93	0.061	0.083	0	0

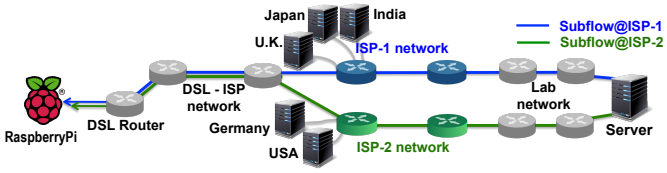


Figure 9. Real-network experimental setup

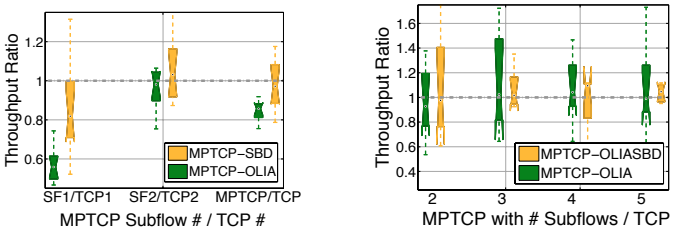
five commercial cloud service providers (2x in Europe, 1x in North America and 2x in Asia) that are connected via 100 Mbps links. Furthermore, we also used consumer hardware with a RaspberryPi connected to a home DSL provider whose connection is limited to asymmetric rates of 25 and 50 Mbps for uplink and downlink, respectively. The experimental setup is illustrated in Figure 9. In our experiments, we evaluated the performance under realistic network conditions with real-network experiments for shared-bottleneck, non-shared bottleneck and shifting bottleneck scenarios using the same parameters as in Section IV .

A. Non-shared Bottleneck

For the NSB scenarios, since the server is well-provisioned, the DSL provider on the client side would normally become the bottleneck (see Section V-B). In order to create server-side bottlenecks, we throttled the server links to 50 Mbps with `netem`, and used the dedicated VMs to receive background traffic from the server: via ISP-1 with an average rate of 40 Mbps, and via ISP-2 with 30 Mbps, resulting in two distinct and separate server-side bottlenecks of 20 Mbps and 10 Mbps, respectively. In Figure 9, the bottlenecks are created in the lab network, before the traffic enters both ISP networks.

For the NSB scenario, we observed a mean SBD accuracy of about 70%[7]. We observe that the two links were similar in terms of RTTs; therefore, NSB accuracy in real networks

[7]Based on what we expect to be a bottleneck with our experiment, remembering much of the network is not within our control.



(a) NSB Multi-Homed Server (b) SB at DSL connection

Figure 10. Throughput comparison of MPTCP and MPTCP-SBD. Boxes as in previous figures.

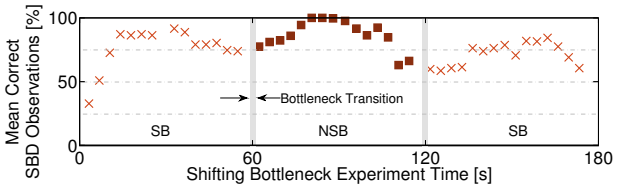


Figure 11. Mean SBD Observations for a Shifting Bottleneck {60,120} s

(70%) is comparable to the emulations (75%) reported in Section IV-C1. Furthermore, we observe an average throughput gain of 20% with MPTCP-SBD compared to MPTCP-OLIA as illustrated in Figure 10(a). This is lower than the results presented in Figure 5; the reason is that MPTCP-OLIA performs just as well as Reno with subflow SF2. Therefore, MPTCP-SBD can yield an improvement only on SF1, which reduces the overall benefit of MPTCP-SBD. Under the same conditions, Reno had approximately 25% more throughput than MPTCP-OLIA.

B. Shared Bottleneck

For the SB scenario, we used the same setup as above, but this time, we used the DSL connection as the bottleneck link. Apart from general background traffic due to it being a real network, we used the VMs to receive background traffic from the server via the two ISPs. The VMs in the U.K., Japan and India received data that were sent via ISP-1 at an average rate of 40 Mbps, and the VMs in Germany and the USA received data that were sent via ISP-2 at an average rate of 30 Mbps.

In this scenario the server sent traffic to the client with 2 subflows connected to the server's ISPs. We observed a mean SBD accuracy of 91% for two subflow case and around 85% for the five subflow case. As a result, Figure 10(b) shows that MPTCP-SBD has similar throughput values compared to MPTCP-OLIA for the two-subflow case and slightly more aggressive than MPTCP-OLIA for higher number of subflows. Compared to the emulation results of SB, we observed that SBD's accuracy in real network experiments, especially for many subflows, is marginally lower, as discussed in Section IV-C2. We justify this observation by the random varying nature of real systems compared to the emulation setup with synthetic background traffic. Also, note that having a high number of subflows in the network traversing the same bottleneck is less likely in real setups.

C. Shifting Bottleneck

For the shifting bottleneck scenario, first illustrated in Section IV-C3, we used the same network (Figure 9), with the client connected by both, ISP-1 and ISP-2, to the home DSL. The server is connected to each ISP via a 100 Mbps connection, whereas the client has asymmetric 25 Mbps in the uplink and 50 Mbps in the downlink. We create a shifting bottleneck every 60 s by changing the background traffic on each ISP from 70 Mbps to 20 Mbps, therefore first creating a shared bottleneck on the client side (SB) and then creating distinct bottlenecks on the server side (NSB).

Figure 11 shows that, although dealing with a real-network setup, the shifting shared bottleneck scenario also performs satisfactorily in the real network. The SBD accuracy we observed after the bottleneck transition is similar to that of SB and NSB scenarios, i.e., about two SBD decisions (7 seconds).

VI. Related Work

Congestion control for multipath transport [23] has evolved from algorithms behaving like regular TCP to *coupled* congestion control: Coupled MPTCP [4] applies *resource pooling* to shift away traffic to less-congested paths. Then, fully-coupled MPTCP [1] applies the idea that the total multipath `cwnd` increases and decreases in the same way as for TCP. Semi-coupled congestion control, or linked-increase (LIA) [5], [10] emerged, because of poor responsiveness and "window flappiness" (oscillatory behavior) of the fully-coupled algorithm. In semi-coupled congestion control the `cwnd` increase is coupled, whereas its decrease is not. MPTCP's default congestion control, Optimized Linked Increase Adaptation (OLIA) [6], [11], improves LIA's unfriendliness and congestion balance. A recent proposal, Balanced Link Adaptation (Balia) [7], improves OLIA's responsiveness. Note that all MPTCP congestion control algorithms are semi-coupled, hence they assume shared bottlenecks in the multipath connection.

The closest related work to this paper is Dynamic Window Coupling (DWC) [13] that studies the benefits of shared bottleneck detection in multipath transport. The DWC mechanism can be summarised as follows. If one of MPTCP's subflows has a loss, the flow manager sends an alert to the other flow controllers. Then, each subflow sets the smoothed RTT back, "undoing" the effect of the last `cwnd`/2, and continues to monitor the smoothed RTT for another `cwnd`/2. If any of the smoothed $RTTs > RTT_{th}$, the subflow is grouped with the subflow that had a loss. RTT_{th} is calculated in a similar way to TCP's smoothed RTT, but based on RTT_{max}. Correlating loss events is difficult since not all flows sharing a link will necessarily experience loss during the same congestion episode. Different paths may also have lags that differ by more than the `cwnd`/2 (about RTT/2) that DWC uses for correlation, making accurate correlation in these scenarios impossible (Note: some of our experiments in Section IV-C5 have much larger lags). RTT, even the smoothed RTT DWC uses, is a very noisy signal. For these and other reasons (see Section III-A) we took a different approach for MPTCP-SBD. It is difficult to directly compare our mechanism with DWC as it does not have a real stack implementation. Therefore, we repeat the same experiment scenarios to compare both mechanisms under similar conditions. By running similar experiments to those in [24, pp 127–130], we observe that our algorithm was able to detect shared bottlenecks with an accuracy of 97%, while DWC mentions an accuracy of approximately 75%.

VII. Conclusions and Future Work

In this work, we show the benefits of shared bottleneck detection for MPTCP. We argue that MPTCP's coupled congestion control is overly conservative when multiple paths do not share a bottleneck. With a shared bottleneck detection algorithm, MPTCP can decouple the subflows' congestion

window increase and still achieve its 3 main design goals: 1) improve throughput, 2) do not harm and 3) balance congestion. We designed and implemented MPTCP-SBD, a dynamic coupled congestion control mechanism for MPTCP that is aware of shared bottlenecks. We demonstrated the efficiency of the proposed MPTCP-SBD in a wide range of settings through extensive emulations, showing significant throughput gains in non-shared bottleneck scenarios without causing harm in shared bottlenecks. These results are further confirmed with real-network experiments.

There are many avenues for future work. One direction is to consider the robustness of the SBD algorithm against attacks, where a receiver could manipulate the SBD feedback to gain an unfair advantage. Therefore, ensuring the feedback mechanism deserves further attention. Moreover, we observed that while MPTCP-SBD tries to detect bottlenecks, all MPTCP's coupled congestion controls try to avoid them by shifting traffic away from congested paths. Also, MPTCP's lowest-RTT scheduler can cause subflows that are sharing a common bottleneck to oscillate in their relative share of the available bandwidth. These oscillations can make the SBD statistics noisier, affecting its performance. At the moment, the SBD algorithm is protocol agnostic. There may be advantages in tuning it specifically to MPTCP's congestion control, however, this could lead to difficulties every time a congestion control algorithm changes. Another possibility is to improve SBD's robustness against oscillations from the link, this could be explored with more complex grouping algorithms. Finally, investigating the performance of MPTPC-SBD for different use-cases is of great interest. One promising use-case is highlighted in [25] where single-homed devices may be able to take advantage of multipath protocols. Hosts with dual-stack IPv4/IPv6 [25] systems may be able to exploit concurrent IPv4 and IPv6 use when both paths are disjoint and the bottleneck lies in the network. This is an ideal application for MPTCP-SBD and should be further investigated.

VIII. Acknowledgements

This work was partially funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed are solely those of the authors.

References

[1] C. Raiciu, D. Wischik, and M. Handley, "Practical Congestion Control for Multipath Transport Protocols," University College London, London/United Kingdom, Tech. Rep., Nov. 2009.

[2] A. Ford, C. Raiciu, M. Handley, S. Barré, and J. R. Iyengar, "Architectural Guidelines for Multipath TCP Development," IETF, Informational RFC 6182, Mar. 2011, ISSN 2070-1721.

[3] D. Wischik, M. Handley, and M. B. Braun, "The Resource Pooling Principle," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 38, no. 5, pp. 47–52, Oct. 2008, ISSN 0146-4833.

[4] F. Kelly and T. Voice, "Stability of End-to-End Algorithms for Joint Routing and Rate Control," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 35, no. 2, pp. 5–12, Apr. 2005, ISSN 0146-4833.

[5] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Boston, U.S.A., Mar. 2011, pp. 99–112.

[6] R. Khalili, N. Gast, M. Popović, and J.-Y. L. Boudec, "MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1651–1665, Oct. 2013, ISSN 1063-6692.

[7] A. Walid, J. Hwang, Q. Peng, and S. H. Low, "Balia (Balanced Linked Adaptation) – A New MPTCP Congestion Control Algorithm," in *Proceedings of the 90th IETF Meeting*, Toronto, Canada, Jul. 2014.

[8] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," IETF, Standards Track RFC 5681, Sep. 2009, ISSN 2070-1721.

[9] K. Fall and S. Floyd, "Simulation-Based Comparisons of Tahoe, Reno and SACK TCP," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 26, no. 3, pp. 5–21, 1996, ISSN 0146-4833.

[10] C. Raiciu, M. Handley, and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols," IETF, RFC 6356, Oct. 2011, ISSN 2070-1721.

[11] R. Khalili, N. G. Gast, M. Popović, and J.-Y. L. Boudec, "Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP," IETF, Internet Draft draft-khalili-mptcp-congestion-control-05, Jul. 2014.

[12] A. Walid, Q. Peng, J. Hwang, and S. H. Low, "Balanced Linked Adaptation Congestion Control Algorithm for MPTCP," IETF, Internet Draft draft-walid-mptcp-congestion-control-02, Jan. 2015.

[13] S. Hassayoun, J. Iyengar, and D. Ros, "Dynamic Window Coupling for Multipath Congestion Control," in *Proceedings of the 19th IEEE International Conference on Network Protocols (ICNP)*, 2011, pp. 341–352, ISBN 978-1-4577-1392-7.

[14] D. A. Hayes, S. Ferlin, and M. Welzl, "Practical Passive Shared Bottleneck Detection using Shape Summary Statistics," in *Proceedings of the 39th IEEE Conference on Local Computer Networks (LCN)*, Edmonton, Alberta/Canada, Sep. 2014, pp. 150–158.

[15] D. Hayes, S. Ferlin, and M. Welzl, "Shared Bottleneck Detection for Coupled Congestion Control for RTP Media." IETF, Internet Draft draft-ietf-rmcat-sbd-01, Jul. 2015.

[16] V. Jacobson, R. Braden, and D. A. Borman, "TCP Extensions for High Performance," IETF, RFC 1323, May 1992, ISSN 2070-1721.

[17] O. Bonaventure, "Multipath TCP Timestamp Option," IETF, Internet Draft draft-bonaventure-mptcp-timestamp-01, Jul. 2015.

[18] J. Ahrenholz, "Comparison of CORE Network Emulation Platforms," in *Military Communications Conference (MILCOM)*, San Jose, California/U.S.A., Oct. 2010, pp. 166–171.

[19] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing Router Buffers," in *Proceedings of the ACM SIGCOMM Conference*, vol. 34, no. 4. New York/U.S.A.: ACM Press, Aug. 2004, pp. 281–292, ISSN 0146-4833.

[20] A. Vishwanath, V. Sivaraman, and M. Thottan, "Perspectives on Router Buffer Sizing: Recent Results and Open Problems," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 39, no. 2, pp. 34–39, Mar. 2009, ISSN 0146-4833.

[21] C. Paasch, "Improving Multipath TCP," Ph.D. dissertation, Université catholique de Louvain-la-Neuve, Nov. 2014.

[22] A. Botta, A. Dainotti, and A. Pescapé, "A Tool for the Generation of Realistic Network Workload for Emerging Networking Scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[23] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda, "Multipath Congestion Control for Shared Bottleneck," in *Proceedings of the 7th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*, Tokyo/Japan, May 2009.

[24] S. Hassayoun, "Synchronization of TCP Packet Losses: Evaluation and Application to Multipath Congestion Control," Ph.D. dissertation, Dépt. Réseaux, Sécurité et Multimédia, Université de Rennes 1, Université Européenne de Bretagne (UEB), Jun. 2011.

[25] I. A. Livadariu, S. Ferlin, Ö. Alay, T. Dreibholz, A. Dhamdhere, and A. M. Elmokashfi, "Leveraging the IPv4/IPv6 Identity Duality by using Multi-Path Transport," in *Proceedings of the 18th IEEE Global Internet Symposium (GI)*, Hong Kong/People's Republic of China, Apr. 2015.