On Bufferbloat and Delay Analysis of MultiPath TCP in Wireless Networks

Yung-Chih Chen
School of Computer Science
University of Massachusetts Amherst
yungchih@cs.umass.edu

Don Towsley
School of Computer Science
University of Massachusetts Amherst
towsley@cs.umass.edu

Abstract—**With the rapid deployment of cellular networks, modern mobile devices are now equipped with at least two interfaces (WiFi and 3G/4G). As multi-path TCP (MPTCP) has been standardized by the IETF, mobile users running MPTCP can access the Internet via multiple interfaces simultaneously to provide robust data transport and better throughput. However, as cellular networks exhibit large RTTs compared to WiFi, for small data transfers, the delayed startup of additional flows in the current MPTCP design can limit the use of MPTCP. For large data transfers, when exploiting both the WiFi and cellular networks, the inflated and varying RTTs of the cellular flow together with the small and stable RTTs of the WiFi flow can lead to performance degradation. In this paper, we seek to investigate the causes of MPTCP performance issues in wireless environments and will provide analyses and a solution for better performance.**

I. Introduction

In recent years, demand to access the Internet by mobile users has soared dramatically. With the popularity of mobile devices and the ubiquitous deployment of cellular networks, modern mobile devices are now equipped with at least two wireless interfaces: WiFi and cellular. As multi-path TCP (MPTCP) has being standardized by the IETF [7], mobile users can now access the Internet using both wireless interfaces simultaneously to provide robust data transport. Although WiFi and cellular networks are pervasive and extensively used, we observe that cellular networks exhibit very different characteristics from WiFi networks: cellular networks usually show large and varying RTTs with low loss rates while WiFi networks normally exhibit higher loss rates but stable RTTs [5]. When leveraging these two networks simultaneously using MPTCP, this heterogeneity results in some performance issues, which eventually degrade MPTCP performance.

In this paper, we study two issues: the impact of the delay startup of additional flows in the current MPTCP design, and the effect of cellular bufferbloat on MPTCP performance. Since Internet traffic is mostly dominated by small downloads (although the tail distributions might be skewed), the delayed startup of additional flows in the current MPTCP implementation can limit the benefits of using MPTCP for small file transfers. To understand when one can start to utilize MPTCP's additional flows, we model the amount of data received from the first flow before the second flow starts and validate the model through measurements. Furthermore, as we observe large and varying RTTs in cellular networks, referred to as bufferbloat, we model and analyze the root cause of this phenomenon. We show how bufferbloat can affect the performance of MPTCP when using both WiFi and cellular networks. Last, we show that, on occasions when bufferbloat is prominent, MPTCP suffers even more because of flow starvation. We provide a solution that can effectively mitigate this performance degradation.

ISBN 978-3-901882-58-6 © 2014 IFIP

The remainder of this paper is organized as follows: Sec. II provides background about MPTCP and Sec. III describes our experimental setup. Sec. IV models the impact of the delayed startup of additional MPTCP flows. We investigate MPTCP performance issues related to cellular networks in Sec. V. Related works are discussed in Sec. VI and Sec. VII concludes this paper.

II. Background

Consider a scenario where a download proceeds between two multi-homed hosts using MPTCP. MPTCP establishes a connection that utilizes the paths defined by all end-to-end interface pairs. The traffic transferred over each path is referred to as a *flow* or a *subflow*. As a standard procedure in running MPTCP, a TCP 3-way handshake is initiated by the client over one path, with *MPTCP-CAPABLE* information placed in the option field of the SYN packet. If the server also runs MPTCP, it then returns corresponding information in the option field of SYN/ACK. The first MPTCP flow is established after the 3-way handshake completes. Information regarding additional interfaces at both hosts is then exchanged through this existing flow. Additional flows can be created afterwards via additional 3-way handshakes with *MP-JOIN* in the option field [7]. Fig. 1 illustrates the MPTCP flow setup and packet exchange diagram of a 2-flow MPTCP connection.

Each MPTCP flow maintains its own congestion window and retransmission scheme during data transfer, and begins with slow-start followed by congestion avoidance. We briefly describe the joint congestion control algorithm that has been proposed as the default congestion controller in MPTCP [18]. Let us denote the congestion window size and round trip time of flow i by w_i and R_i, and the aggregate window size over all the flows by w, where $w = \sum w_i$.

Coupled congestion algorithm was introduced in [21] and is the default congestion controller of MPTCP [18]. It couples the increase phase but does not change the behavior of TCP in the case of a loss.

- ACK on flow i: $w_i = w_i + \min(\alpha/w, 1/w_i)$
- Each loss on flow i: $w_i = \frac{w_i}{2}$

where α is an aggressiveness parameter that controls the speed of the increase phase to achieve fairness (details in Sec. V-B). Note that a revised version of the coupled algorithm was proposed in [12], which aims for better congestion balancing. In this paper, we will only focus on the coupled controller as it is the default congestion controller of the current MPTCP implementation [14].

III. Experimental Setup and Performance Metrics

In this paper, we evaluate MPTCP performance through measurements in WiFi and cellular networks. Since the current MPTCP implementation delays the startup of additional flows, this delay can limit MPTCP's performance when downloading small files. Moreover, as WiFi and cellular network exhibit different characteristics, when leveraging these two networks simultaneously using MPTCP, this heterogeneity can result in MPTCP performance degradation. Thus, we first describe our experimental setup followed by the performance metrics of interest.

Our experiment setup consists of an MPTCP-capable server with one Intel Gigabit Ethernet interface connecting the server to the UMass network. The mobile client is a Lenovo X220 laptop and has a built-in 802.11 a/b/g/n WiFi interface. Both the server and the client have 8 GB of memory. The WiFi network is accessed by associating the WiFi interface to a D-Link WBR-1310 wireless router connected to a private home network in a residential area. For different configurations, an additional cellular 3G/4G device from one of the three carriers (i.e., AT&T 4G LTE, Verizon 4G LTE, and Sprint 3G EVDO) can be connected to the laptop through a USB cable, and no more than two wireless interfaces (including the WiFi interface) are used simultaneously. Both the server and the client run Ubuntu Linux 12.10 with kernel version 3.11.3 using the MPTCP kernel implementation [14] version v0.88, and the default *coupled* congestion controller is used.

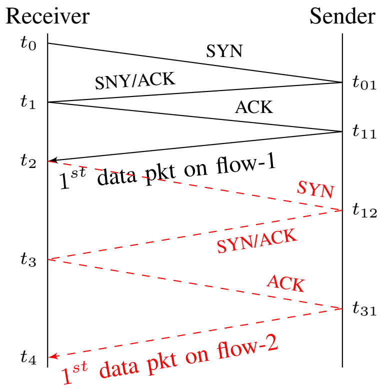

Fig. 1. MPTCP flow establishment diagram.

The UMass server is configured as an HTTP server running Apache2 on port 8080. The client uses *wget* to retrieve Web objects of different sizes (8 KB to 32 MB) via all available paths. We randomize the order of each configuration (i.e., single/multi-path, file sizes, and cellular carriers) when performing measurements. The measurements were conducted every hour over multiple days and the traces are collected at both the server and client side with *tcpdump*.

We focus on a simple 2-flow MPTCP scenario (i.e., the client has one WiFi and one cellular interface) and are interested particularly in the MPTCP *download times* for different file sizes, and the *round trip times (RTTs)* of each MPTCP flow. We define the download time as the duration from when the client sends out the *first SYN* to the server to the time it receives the *last data packet* from the server. RTTs are measured on a per-flow basis and are defined as the time differences between when packets are sent by the server and the reception times of the ACKs for those packets such that the ACK numbers are one larger than the last sequence numbers of the packets (i.e., not retransmitted packets).

IV. Modeling MPTCP Delayed Startup of Additional Flows

As Internet traffic is dominated by downloads of small files, the current MPTCP's delayed startup of additional flows can limit the benefits of using MPTCP. To understand when the second flow is utilized, we model the amount of data that a user receives from the

Parameter	Description
I_w	initial congestion window size
Δ	inter-arrival time of 1^{st} data pkts
R_i	packet RTT of flow i
d_{ss}	pkts sent in slow start
d_{ca}	pkts sent in congestion avoidance
b	delayed ACK parameter
ψ	exponential growth rate: $1 + 1/b$
γ	flow RTT ratio: R_2/R_1
β	congestion window ratio: w_2/w_1
α	MPTCP window increase parameter
F	network storage capacity
B	network buffer size
μ	network bandwidth
τ	minimum round trip latency

TABLE I. PARAMETER DESCRIPTION

first flow before the second flow starts based on the RTT ratio of the WiFi and the cellular networks.

As described in Sec. II, additional MPTCP flows can be created only after the first flow is established. In this paper, we focus on the case of 2-flow MPTCP and Fig. 1 illustrates the period of time of interest to us in the 2-flow MPTCP flow establishment diagram. It starts with a SYN packet sent over the first path (t_0) to the arrival of the first data packet received on the second path (t_4).

Let Δ denote the time between the arrivals of the first data packets in the two flows (i.e., $\Delta = t_4 - t_2$). Let R_1 and R_2 denote the RTTs of the first and the second flow, and $R_2 = \gamma R_1$, where $\gamma > 0$. Note that in the current MPTCP setting, the inter-arrival time of the first data packets in both flows is:

$$\Delta \quad = \quad t_4 - t_2 = 2 \cdot R_2 = 2\gamma \cdot R_1. \tag{1}$$

An MPTCP flow starts with a slow start phase where the sender sends as many packets as its congestion window ($cwnd$) allows, and Linux TCP uses delayed ACK [4] (i.e., the receiver sends one ACK to the sender for every b-th received data segment), during each packet round trip, the sender will receive approximately $cwnd/b$ ACKs, where w_i is the window size of flow i. We use ψ to denote the exponential growth rate of the congestion window during slow start such that $\psi = (1 + 1/b)$. The sender leaves slow start and enters congestion avoidance when a loss occurs. Last we denote the initial congestion window size by I_w.

We begin with the case where no packet loss occurs in $[t_2, t_4]$, henceforth referred to as Δ, and Table I lists the associated parameters. We first denote the number of packets received from the first flow during the i^{th} round trip in slow start by $d_{ss}(i)$,

$$d_{ss}(i) = I_w \cdot \psi^{i-1}. \tag{2}$$

When the slow start threshold is infinity[1], the first flow can send the following number of packets before the receiver begins to receive packets from the delayed second flow,

$$d = \sum_{i=1}^{\Delta} d_{ss}(i) = I_w \cdot \frac{\psi^{2\gamma} - 1}{\psi - 1}. \tag{3}$$

Fig. 2 shows measurement results for the number of packets received from the first flow during Δ as a function of RTT ratio[2]. Each dot represents an MPTCP measurement with WiFi and one cellular carrier. Note that in our measurement setting, WiFi is the primary flow, and hence $\gamma > 1$. The dashed line depicts the loss-free case presented in Eq. (3), and only a few samples match the dashed prediction line. The majority of the measurements are not captured by the loss-free model.

Since WiFi exhibits a higher loss rate $(0.9 \pm 0.2\%)$ than cellular ($< 0.03\%$ for all carriers) from our measurements, in the following we calculate the expected number of packets that an MPTCP user can receive from the first flow when *at most one loss* occurs during Δ. We look at the case of one packet loss mainly because the loss rate is generally smaller than 1% and there are only several packet round trips in Δ. Since SACK is used in MPTCP, multiple losses within the same round trip only account for *one loss event* that leads to only one congestion window reduction, we regard multiple losses in one round trip as a single loss event.

For simplicity, we assume each packet is dropped with probability p, independently of each other, and the receiver receives $d_{ss}(i)$ packets during the i^{th} round trip in slow start. We denote by $d_{ca}(j \mid k)$ the number of packets received during the j^{th} round trip of Δ (in congestion avoidance) given a loss event occurs during the k^{th} round trip,

$$d_{ca}(j \mid k) \quad = \quad \frac{d_{ss}(k)}{2} + j - (k+1), j > k. \tag{4}$$

Let $S(k)$ denote the probability that no packet loss occurs during *slow start* before the k^{th} round trip in Δ, and $C(k)$ denote the probability that no packet loss occurs during *congestion avoidance* to the end of Δ

[1] Current TCP does not have a default initial slow start threshold [6]. TCP enters congestion avoidance when a loss event occurs, and caches this updated slow start threshold for the subsequent connections to the same destination IP address.

[2] RTT ratio is calculated from the traces. R_2 is the RTT of the first data packet of flow-2, while R_1 is the average RTT of packets received during Δ. Since RTTs vary from time to time, the RTT ratios presented here are therefore estimates made from our measurements.

given a loss occurs at the k^{th} round trip; it is

$$S(k) = p(1-p)^{d_{ss}(k)-1} \prod_{i=1}^{k-1} (1-p)^{d_{ss}(i)}, \quad (5)$$

$$C(k) = \prod_{j=k+1}^{\Delta} (1-p)^{d_{ca}(j|k)}. \quad (6)$$

We define $C(0) = C(\Delta) = 1$, $S(0) = (1-p)^d$, and the conditional probability that a loss occurs at the k^{th} round trip to be:

$$\mathbb{P}(k) = \frac{S(k) \cdot C(k)}{Q}, k = 0, 1, 2, \ldots \Delta \quad (7)$$

where $Q = \sum_{i=0}^{\Delta} S(k) \cdot C(k)$, and $\mathbb{P}(0)$ represents the case of no loss event during Δ.

Denote by $d(k)$ the number of *total packets* received by the end of Δ from the first flow given a loss occurs at the k^{th} round trip; we have:

$$d(k) = \begin{cases} \sum_{i=1}^{\Delta} d_{ss}(i) & \text{, if } k = 0. \\ \sum_{i=1}^{k} d_{ss}(i) - 1 + \sum_{j=k+1}^{\Delta} d_{ca}(j \mid k) & \text{, otherwise.} \end{cases} \quad (8)$$

The expected number of packets received from the first flow before the *delayed* second flow starts is

$$\mathbb{E}[\text{ received packets }] = \sum_{k=0}^{\Delta} \mathbb{P}(k) \cdot d(k). \quad (9)$$

With Linux's initial window default setting $I_w = 10$ and delayed ACK parameter $b = 2$ (thus $\psi = 1.5$), we measure the average WiFi loss rate ($p = 0.009$) and the RTT ratio from each of the 2-flow MPTCP connections with all cellular carriers. The expected number of packets received *before* the delayed second flow starts is depicted as the solid line in Fig. 2. By fitting the empirical averages of different γ to the expected values derived from our model, the regression statistics show $R^2 = 0.8758$, indicating a good fit to our model.

When WiFi's loss rate is about 0.9%, before the delayed cellular flow starts, a user can receive an average number of 67, 88, and 130 packets respectively from the WiFi flow while the cellular flow is AT&T, Verizon, and Sprint with a median RTT ratio γ of 3.9, 4.4, and 6.0, respectively. That is, for small file transfers, the MPTCP's delayed startup of additional cellular flows results in low utilization of these flows. In the following section, we focus on larger file transfers and investigate MPTCP performance issues affected by cellular network's over-buffering.

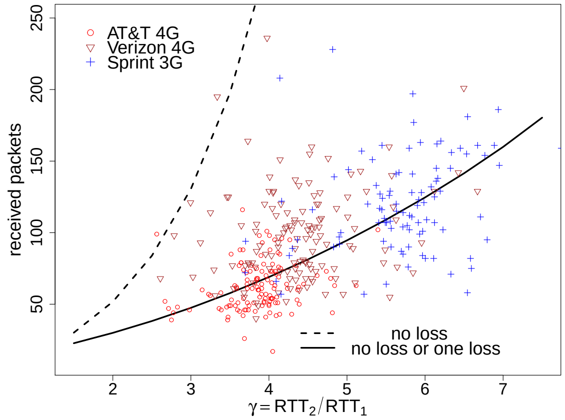

Fig. 2. Approximation of the expected number of received packets from the first flow as a function of RTT ratio. Samples are MPTCP measurements for different carriers of file sizes 1MB to 32MB.

V. MPTCP Performance Evaluation with Cellular Networks

We investigate the fact that cellular networks exhibit inflated and varying RTTs, also known as *bufferbloat*. As we have observed such phenomenon through measurements, we model and analyze *how* this phenomenon occurs as well as *what* is the outcome of this in terms of RTTs and loss rates. Last, we show that severe bufferbloat can lead to low flow utilization and eventually degrade MPTCP performance.

A. Understanding Bufferbloat and RTT Variation

The phenomenon of large and varying RTTs in cellular networks has been recently observed and termed *bufferbloat* [8], which occurs due to the existence of large buffers in the networks. Results from our earlier measurement study [5] are consistent with previous studies by Allman [1] and Jiang et al. [10], which show that bufferbloat is less prominent in most wired/WiFi networks (i.e., public/home WiFi networks), and can be severe in 3G/4G cellular networks. However, in addition to the severe RTT inflation of the cellular networks, we also observe that cellular networks exhibit loss-free environments. Our measurements of downloads of file sizes 8 KB to 32 MB among all the cellular carriers indicate loss rates less than 0.03%, which are much smaller than those of WiFi, approximately 0.9%.

Fig. 3 presents the measurement results of average connection RTT as a function of file size. Throughout the measurements, WiFi exhibits a stable average connection RTT of approximately 30 ms while AT&T LTE exhibits an average RTT that ranges from 60 ms to 180 ms as file size increases. Sprint 3G exhibits the

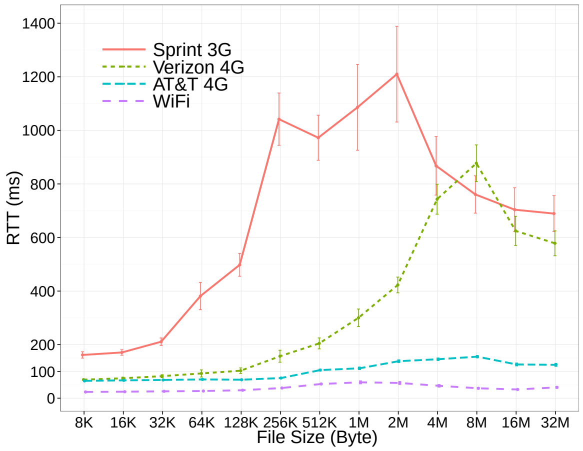

Fig. 3. Average connection RTT as a function of file sizes for different carriers (mean ± standard error).

greatest RTT variability among all the carriers, with averages ranging from 160 ms to 1.2 second. Verizon LTE, although using the same 4G technology as AT&T LTE, also exhibits high variability in its RTTs, and the averages range from 60 ms to 900 ms as the transferred file size increases. That is, γ can quickly rise from 2 to 40, and in some cases up to 80.

In the following we seek to understand how RTT inflation occurs due to network over-buffering. Let us denote by μ the network bandwidth, by τ the minimum packet RTT, and the minimum bandwidth-delay product (BDP) is thus denoted by $\mu\tau$. We denote the size of network buffer by B, and the network storage capacity by F, which is the maximum amount of in-flight packets that can be stored in the network, including one BDP and the size of the network buffer, and hence $F = \lceil B + \mu\tau \rceil$. Although early works suggested network buffer sizes to be much smaller than the average BDP [2] [3], recent studies on bufferbloat [8] [10] report the opposite in modern network systems. Therefore, to understand the root cause of bufferbloat in cellular networks, we assume the network buffer B to be larger than one BDP in the following analysis.

When more than $\mu\tau$ packets are in-flight, the network buffer gradually fills up. The queueing delay hence increases as does the RTT. Since the congestion window advances by one packet in the congestion avoidance phase, the RTT always increments by $1/\mu$ (i.e., additional queueing delay in the buffer) and, hence, is a stepwise increasing function that can be approximated by a linear function [20]. Fig. 4 depicts the RTT evolution in a complete congestion avoidance cycle ($0 < t < T_{fr}$). When a packet loss is detected, TCP enters *fast recovery*, and the congestion window is then halved. The sender resumes its transmission after the number of unACKed packets reduces to half of its previous size (the new window size). During this pause period (in Fig 4, where $T_{ca} < t < T_{fr}$), the buffer drains and since the sending rate is halved, the maximum RTT is also reduced by half. After fast recovery, a new congestion avoidance cycle starts.

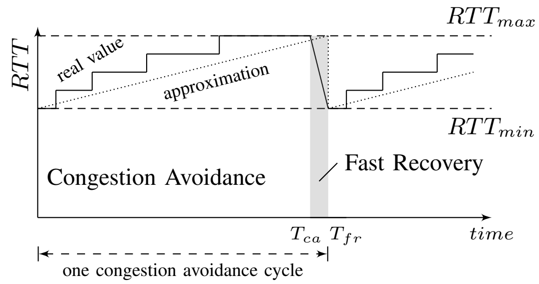

Fig. 4. RTT evolution: one congestion avoidance cycle

From [13], when the sender starts to fill the network buffer, its congestion window, $w(t)$, is

$$w(t) = \sqrt{\frac{1}{4}(F+1)^2 + \frac{2\mu t}{b}} \qquad (10)$$

where b is the delayed ACK parameter such that an ACK is generated on reception of every b packets. When $w(t)$ reaches the network storage capacity F at time T_{ca} (the end of the congestion avoidance phase), by solving Eq. (10) for $w(T_{ca}) = F$, we obtain $T_{ca} = \frac{3b}{8\mu}(F+1)^2$.

That is, during one congestion avoidance cycle, μT_{ca} packets are transmitted, and then one additional window of packets are sent before the lost packet is detected. Therefore, the number of packets sent during a congestion avoidance cycle is $N \approx \lceil \frac{3b}{8}(F+1)^2 + F \rceil$ (excluding the retransmitted packet). If we assume packets are dropped only due to the filled network buffer, then within one congestion avoidance cycle, the loss rate is $1/N$.

Fig. 5 depicts the network loss rate, $1/N$, as a function of network storage capacity F. Given a network bandwidth $\mu = 10$ Mbps and $\tau = 15$ ms, with the TCP delayed ACK parameter $b = 2$, the minimum BDP is roughly 12 packets. By setting a buffer size equal to the minimum BDP ($B = \mu\tau$), the loss rate drops from 0.7% to 0.2%. When the buffer size increases 8-fold, the loss rate drops to 0.01%. Hence, if we assume the minimum BDP does not change in each cycle, and packets are dropped only due to the filled network buffer, increasing the buffer size reduces the loss rate dramatically.

Since the network over-buffering issue has been recently reported by [8] and is termed as *bufferbloat*, our analyses above shed light on how bufferbloat can result in extremely small loss rates while exhibiting the

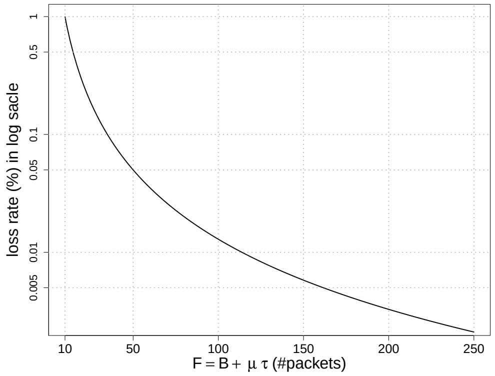

Fig. 5. Network loss rate as a function of network storage F.

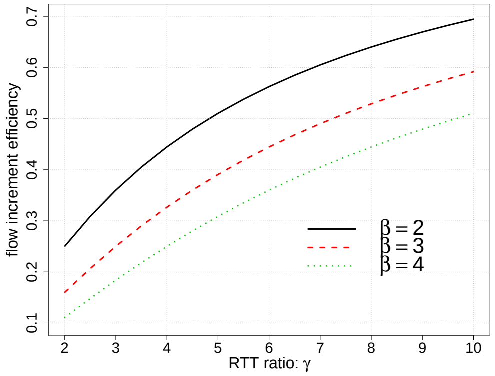

Fig. 6. MPTCP flow increment efficiency as a function of γ.

large RTT variations observed in our measurements. When flow RTTs are small and stable, ACKs return to the sender quickly and the RTT estimates are precise. However, when one of the MPTCP flows exhibits small and stable RTT values while the other experiences severe RTT inflation without packet losses, the joint congestion controller can be misguided by TCP's congestion inference from packet losses, and lose its ability to quickly balance congestion across the flows. In the following, we investigate how inflated cellular RTTs can affect MPTCP performance.

B. Idle Spins of the Joint Congestion Controller

The coupled congestion controller increases the congestion window of flow i upon reception of each ACK with $w_i = w_i + \min(\alpha/w, 1/w_i)$, where $w = \sum_i w_i$. Since this controller does not couple the decrease phase, it relies on α to respond to changes in flow windows and RTTs, and α is defined in [18] as:

$$\alpha \quad = \quad \frac{\max\{\frac{w_i}{R_i^2}\}}{(\sum_i \frac{w_i}{R_i})^2} \cdot w \tag{11}$$

As α is updated whenever there is *a packet drop* or *once per RTT* rather than once per ACK to reduce computational cost [18], this results in slow responsiveness of α. Moreover, since ACKs are used at the sender for RTT estimations and TCP uses delayed ACK, when the network is over-buffered, the sender fails to estimate the RTT in a timely manner. Also, the RTT values used in Eq. (11) are smoothed values (SRTT) with the consequence that they lag behind the true RTTs when they are rapidly increasing. As a result, the coupled congestion controller underestimates α, and hence the MPTCP increase rate.

For a simple 2-flow MPTCP with congestion window ratio $\beta = w_2/w_1$ and RTT ratio $\gamma = R_2/R_1$, if we assume in Eq. (11), the numerator has $w_1/R_1^2 \geq w_2/R_2^2$ (i.e., flow 1 is currently the better flow [21]), the MPTCP window increase rate can be rewritten as

$$\frac{\alpha}{w} = \frac{\frac{w_1}{R_1^2}}{(\frac{w_1}{R_1} + \frac{\beta w_1}{\gamma R_1})^2} = \frac{1}{(1+\beta/\gamma)^2} \cdot \frac{1}{w_1}. \tag{12}$$

Upon reception of each ACK, flow 1 increases w_1 by $\frac{1}{(1+\beta/\gamma)^2} \cdot \frac{1}{w_1}$. We define flow *increment efficiency* as the ratio of a flow's increase rate when running MPTCP to that of running single-path TCP (i.e., $1/(1+\beta/\gamma)^2$). Fig. 6 shows flow 1's increment efficiency as a function of RTT ratio γ for different window ratios β.

Since cellular networks exhibit loss rates typically lower than 0.03% [5], the cellular flow's window is often larger than that of the WiFi flow. For a 2-flow MPTCP connection (e.g., WiFi the first and cellular the second) with $\beta = 2$, when the cellular RTT inflates, γ can ramp up quickly from 2 to 8 as more packets are in-flight (as shown in Fig. 3). As depicted in Fig. 6, if we assume the window ratio β remains fixed at 2, and the inflated RTTs can be correctly estimated during the cellular RTT inflation period while γ increases from 2 to 8, the WiFi flow's increment efficiency should increase from 0.25 to 0.65. However, due to the slow responsiveness of α and the cellular flow's lagged RTT estimates, the WiFi flow's increment efficiency remains at 0.25 for at least one RTT. Thus, the WiFi flow's increase rate is underestimated by 61% during this cellular RTT inflation period.

This issue becomes more critical when the cellular flow, henceforth referred to as flow 2, fails to receive new ACKs from the client even after the sender performs fast retransmit (within R_2), and eventually its timeout timer expires after one retransmission timeout (RTO)[3]. The idle period that flow 2 does not send any

[3] $RTO = SRTT + \max\{G, 4 \times RTTVAR\}$, where $RTTVAR$ is the RTT variance, and the initial value of RTO is 1 sec [17]. Note that G is the clock granularity set to 0.2 sec in modern Linux systems.

packets, T_{idle}, is thus $T_{idle} \approx RTO - R_2$, and the period can be longer when bufferbloat is more prominent. During T_{idle}, the aggregate window, w, still remains large as flow 2's congestion window, w_2, will only be reset to two after the timeout event. The WiFi flow's (flow 1) increase rate, α/w, is therefore very small due to this large w. Moreover, during T_{idle}, packets are only delivered over flow 1. Flow 1's bandwidth, as we have observed, is severely underestimated and its increase rate should have been raised to $1/w_1$ as that of a single-path TCP.

Ideally when an MPTCP connection includes different flows characterized by *diverse but stable RTTs*, α can be set to respond network changes quickly and the coupled congestion controller should achieve MPTCP's desired throughput. However, since cellular networks exhibit bufferbloat, which in turn results in large congestion windows and unstable RTTs, these properties eventually lead to MPTCP performance issues.

C. Flow Starvation and TCP Idle Restart

MPTCP maintains a connection-level *shared send queue* for all the packets scheduled to be sent, while each flow manages its own subflow-level send buffer. When a flow has available space in its congestion window, the MPTCP packet scheduler clones the first segment at the head of the shared send queue into the flow send buffer[4].

When all previously sent packets over a particular flow are ACKed (subflow-level ACK), the data in the subflow-level send buffer can be removed. The original segment, however, will remain in the connection-level shared send queue until all older packets are correctly received and ACKed (connection-level ACK) via any of the available flows. That is, when a packet in the shared send queue is ACKed at the subflow level and the connection level, it can still be retained in the connection-level send queue if any older packets with smaller connection-level sequence numbers have not yet been reported as received. Once those older packets are received and ACKed, the connection-level ACKed packets are dequeued, and new packets from the application are appended to the tail of the connection-level send queue.

When one of the MPTCP flows suffers severe bufferbloat and the transmission latency quickly increases, packets may take unexpectedly longer to reach the receiver. Suppose the connection-level send queue has capacity M, and the first i packets are currently scheduled on the cellular flow, experiencing severe bufferbloat, while the $i+1^{th}$ to j^{th} packets are scheduled to the WiFi flow. Since WiFi has a much smaller RTT than cellular, the WiFi flow packets are quickly ACKed, and removed from their flow send buffer. The WiFi flow then has space in its congestion window and requests more packets from the connection-level send buffer (the $j+1^{th}$ to M^{th}). Note that at this point in time, packets traversing cellular are experiencing high latency due to bufferbloat, and the first i packets are still en-route while the WiFi flow has successfully received ACKs for the $i+1^{th}$ to M^{th} packets. Up until this point, those $M-i$ packets sent over WiFi are ACKed at the subflow level, and hence no data is retained in the WiFi flow send buffer. On the other hand, their original copies still remain in the connection-level send buffer, waiting for the first i packets sent over cellular to reach the receiver. Before the oldest i packets in the queue are correctly received and ACKed, the connection-level send queue fills up (the first i packets over the cellular flow, and $M-i$ ACKed packets waiting for the oldest i packets to be ACKed).

This leads to *flow starvation*. The WiFi flow has now removed all the ACKed data from its send buffer (subflow-level ACKed), and requested new packets from the shared send queue. The shared send queue, on the other hand, has no available packets to allocate to the WiFi flow. Moreover, it can not request any new packets from the application layer, as currently the shared send queue is full. This dilemma ends when *the oldest packets* in the queue are correctly received and ACKed, the application places new data in the connection-level send queue, and the WiFi flow resumes.

The consequence of an idle MPTCP flow has far more impact than above. When the WiFi flow's idle period is longer than the current estimated flow retranmission timeout (RTO) with window size w_1, the TCP's congestion window validation mechanism [9] is triggered and calculates a *restart window*, $w_r = \min(w_1, I_w)$, for the idle WiFi flow. For each RTO event, w_1 is halved until w_r is reached[5]. After the WiFi flow resumes and its window is reset to a new value, it is then forced to re-probe the network with slow start.

Fig. 7 illustrates a time series of the WiFi flow's congestion window and the cellular flow's RTT. Note that the cellular flow here suffers severe bufferbloat with periodic RTT inflation as illustrated Fig. 4. At the beginning of the connection, the cellular RTT inflates quickly from 80 ms to 800 ms, and hence produces the WiFi flow's first idle period soon after it enters congestion avoidance. The WiFi flow's congestion window is halved before the next slow start because it experiences an idle period of one RTO. Note that this behavior is not due to loss events, as the congestion window is often reset to w_r rather than two as in a timeout event.

[4] This is true when no packet is in the connection-level retranmission queue.

[5] Details please refer to the procedure `tcp_cwnd_restart()` in `tcp_output.c` in the Linux kernel.

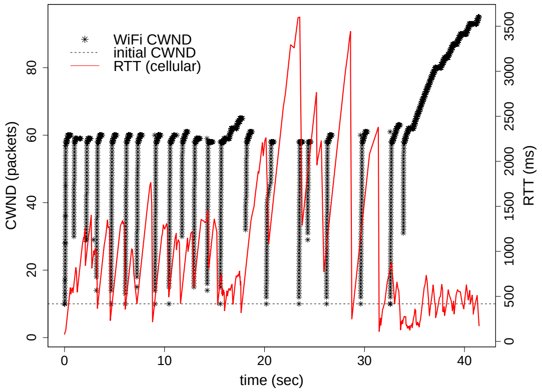

Fig. 7. Severe bufferbloat: periodic RTT inflation of the cellular flow results in idle restarts of the WiFi flow.

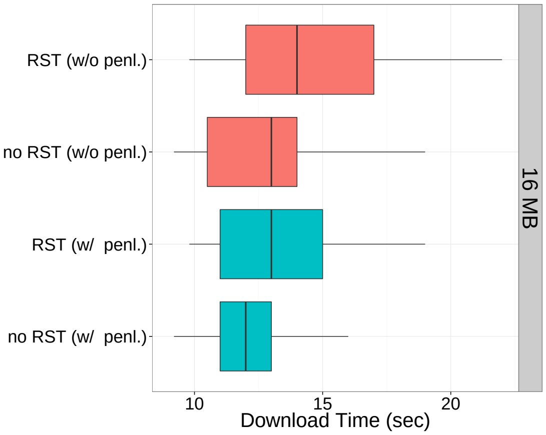

Fig. 8. Download time comparison: MPTCP with idle restart (RST) and penalization (penl.) enabled/disabled.

For the subsequent idle restarts in Fig. 7, WiFi's congestion window is often reset to I_w (i.e., the initial window of 10 packets). This phenomenon is very prominent during time interval 20 to 32 seconds, where the cellular RTTs inflate dramatically up to 3.5 seconds, and the WiFi idle period is much longer than its current RTO. The phenomenon of RTT inflation is less pronounced after 35 seconds when the RTT drops from 2 sec to 400 ms. After this point in time, the receiver receives packets from the cellular flow more quickly, and the corresponding ACKs to those packets arrive at the sender in a timely fashion without blocking the shared send queue. Hence, the WiFi flow successfully completes slow start and enter congestion avoidance. During these idle periods, not only does the WiFi flow starve, but the cellular flow exhibits a low *increment efficiency*. This occurs for the same reason described in Sec. V-B when one of the flows experiences a long idle period, the coupled controller underestimates the increase rate and eventually degrade MPTCP's performance.

To avoid unnecessary performance degradation due to bufferbloat, we propose to *disable* the default *idle restart* functionality [9] when using MPTCP with cellular. The benefit of doing so is two-fold. First, allowing an idle MPTCP flow to quickly restore its original congestion window reduces network resource waste and saves download time by not having to probe for the network capacity again. Second, as the coupled controller couples all flows at the increase phase, each flow's increase rate is much slower than its single-path counterpart. Therefore, after an idle restart, it takes much longer for the restarted flow to reach the same sending rate before the restart event.

To showcase how our proposed approach can effectively mitigate the impact of flow starvation, Fig. 8 showcases the results of MPTCP download times when TCP idle restart (RST) is enabled/disabled. Moreover, as the penalizing scheme proposed in [19] aims to optimize receive memory usage by reducing the window size of flows that contribute too many out-of-order packets, we also show the results of MPTCP with (w/ penl.) and without penalization (w/o penl.). Note that the MPTCP's receive buffer is set to the default maximum size of 6 MB, which is much larger than the targeted scenario in [19]. We do not disable TCP auto-tuning as proposed in [16] as our goal is to understand bufferbloat's impact on MPTCP shared send buffer rather than the efficiency of utilizing the receive buffer at the beginning of each connection.

We chose one cellular carrier that exhibits prominent bufferbloat during the day and performed file downloads of 16 MB files with 2-flow MPTCP connections. For each configuration, we performed 40 rounds of measurements and randomized the order of the configurations to reduce possible correlations during our measurements. When the cellular flow experiences bufferbloat and idle restarts occur frequently, we observe that MPTCP suffers severe performance degradation. The penalizing scheme helps in this case by throttling the cellular flow's sending rate and hence mitigates bufferbloat. Furthermore, it delays the occurrence of the idle restarts and provides an opportunity for those connection-level ACKs of the late received packets sent over cellular to arrive at the sender and unblock the shared send buffer.

When the TCP idle restart is *disabled*, the download time (both mean and variance) reduces for both vanilla MPTCP (no RST w/o penl.) and the MPTCP with penalization (no RST w/ penl.). We show that, when bufferbloat is evident, by disabling the TCP idle restart, on average the performance of MPTCP download time improves by 30% (no RST w/ penl.).

VI. Related Work

To the best of our knowledge, this is the first paper that models the impact of MPTCP's flow delayed startup to understand when a user can start to leverage the additional flows. It is also the first work that investigates the impact of bufferbloat on MPTCP performance. Since the root cause of MPTCP performance problems of *flow starvation* and *idle restart* is *bufferbloat* in the cellular networks, if cellular operators can size their router buffers properly as suggested in [2] [3], the bufferbloat issues can be mitigated. The associated MPTCP performance issues can hence be resolved. Several works have recently aimed to tackle this issue based on existing infrastructure. Jiang et al. [10] proposed a receiver-based rate limiting approach to mitigate the RTT inflation by tracking down the RTT evolution. Nichols and Jacobson proposed a scheduling algorithm, CoDel [15], to control network delay through managing router buffers. These approaches require additional changes and management at the receivers and at the buffers within the network, and might directly affect the performance of MPTCP from different perspectives. If MPTCP can wisely select available paths and flows to leverage [11] without being hampered by bufferbloat, and the joint congestion controller can be more responsive to the rapid variation of RTTs, the benefits of MPTCP will be more pronounced. As these require further study and more careful examination in the networks, we leave these as future works.

VII. Conclusion

In this paper, we study the performance of a simple scenario of 2-flow MPTCP with WiFi and cellular networks. We show that for small downloads, the current MPTCP's delayed startup of additional flows limits MPTCP's performance. Based on the RTT ratio of the WiFi and cellular networks, we demonstrate that the additional flows can be underutilized for small file transfers by modeling the number of packets received before the second flow starts. Second, as we have observed bufferbloat in the cellular networks, we investigate the root cause of large and varying cellular RTTs by modeling and analyzing bufferbloat. Furthermore, we show how MPTCP might suffer from cellular bufferbloat when coupling with another WiFi flow for large file transfers. Last, we show how flow starvation occurs when bufferbloat is prominent and can eventually harm MPTCP's performance. By disabling the TCP idle restart for congestion window validation, we show that this is an efficient approach to mitigate the MPTCP performance degradation.

Acknowledgments

This research was sponsored by US Army Research laboratory and the UK Ministry of Defense under Agreement Number W911NF-06-3-0001. This material is also based upon work supported by the National Science Foundation under Grant No. CNS-1040781.

References

[1] M. Allman. Comments on bufferbloat. *ACM SIGCOMM CCR*, 43(1):30–37, 2012.

[2] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *ACM SIGCOMM*, 2004.

[3] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental study of router buffer sizing. In *ACM IMC*, 2008.

[4] R. Braden. RFC 1122: Requirements for internet hosts – communication layers, 1989.

[5] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. A measurement-based study of MultiPath TCP performance over wireless networks. In *ACM IMC*, 2013.

[6] S. Floyd. RFC 3472: Limited slow-start for TCP with large congestion windows, 2004.

[7] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. RFC 6824: TCP extensions for multipath operation with multiple addresses.

[8] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the Internet. *Communications of the ACM*, 55(1), 2012.

[9] M. Handley, J. Padhye, and S. Floyd. RFC 2861: TCP congestion window validation, 2000.

[10] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G networks. In *ACM IMC*, 2012.

[11] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. *Communications of the ACM*, 54:109–116, Jan. 2011.

[12] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. MPTCP is not pareto-optimal: Performance issues and a possible solution. In *ACM CoNEXT*, 2012.

[13] T. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *Networking, IEEE/ACM Transactions on*, 5(3):336–350, 1997.

[14] MultiPath TCP Linux Kernel implementation. http://mptcp.info.ucl.ac.be/.

[15] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[16] C. Paasch, R. Khalili, and O. Bonaventure. On the benefits of applying experimental design to improve multipath TCP. In *ACM CoNEXT*, 2013.

[17] V. Paxson and M. Allman. RFC 2988: Computing TCP's retransmission timer, 2000.

[18] C. Raiciu, M. Handly, and D. Wischik. RFC 6356: Coupled congestion control for multipath transport protocols, 2011.

[19] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable multipath TCP. In *USENIX NSDI*, 2012.

[20] M. Scharf, M. Necker, and B. Gloss. The sensitivity of TCP to sudden delay variations in mobile networks. In *IFIP Networking*, 2004.

[21] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *USENIX NSDI*, 2011.