IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received September 9, 2020, accepted September 23, 2020, date of publication October 27, 2020,
date of current version November 10, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3034129

Stateful-TCP–A New Approach to Accelerate TCP Slow-Start

LINGFENG GUO AND JACK Y. B. LEE, (Senior Member, IEEE)
Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong
Corresponding author: Jack Y. B. Lee (jacklee@computer.org)

ABSTRACT The Transmission Control Protocol (TCP) is one of the pillars of the Internet. Therefore extensive research has been conducted to improve its performance, primarily via optimizing TCP's congestion control algorithm. In this work, we show that besides congestion control, TCP's Slow-Start algorithm is increasingly becoming a bottleneck in modern high-speed networks. To tackle the problem, we propose a different approach called Stateful-TCP, where path bandwidth estimated in a previous flow is used to instantly ramp up the transmission rate of the subsequent flow to the same destination. This eliminates the need for bandwidth probing in conventional Slow-Start, enabling TCP to efficiently utilize the available path bandwidth right from the beginning. We applied Stateful-TCP to Linux's default TCP implementation – Cubic, to form S-Cubic and evaluated its performance via extensive emulations and Internet experiments. Results from independent Internet experiments using over 1,000 end-user clients showed that S-Cubic could reduce FCT by 37.5% and increase throughput by over 50% compared to Cubic. As opposed to an entirely new TCP design, Stateful-TCP is designed to complement congestion control and thus could potentially be applied to many of the existing TCP variants, as well as incorporated into future TCP designs.

INDEX TERMS TCP, slow-start, stateful, bandwidth estimation.

I. INTRODUCTION

The transmission Control Protocol (TCP) is one of the pillars of the Internet as most applications use it for data transport. For this reason, much research has been done to improve its performance in a wide range of network settings (e.g., [1]–[20]). Common to almost all TCP variants – a TCP flow begins in a Slow-Start phase [21] with a relatively low initial transmission rate (either limited by the initial congestion window size, i.e., CWnd, or transmission rate), which is then increased progressively as packets are correctly delivered. This conservative approach is necessary as today's networks span a wide range of bandwidth (e.g., from Kbps to Gbps), thus transmitting too aggressively at the beginning may cause severe congestion in networks with limited bandwidth.

Nevertheless, TCP's Slow-Start mechanism could become a significant bottleneck to its performance, more so in networks with large bandwidth-delay-product (BDP). This limitation is widely-known, and the initial CWnd of TCP has been raised from 2 MSS in TCP Reno [22] to 10 MSS in today's Linux kernel [1]. While increasing the initial CWnd can mitigate the limitation to some extent, it does not resolve the problem completely, as many Internet flows are relatively short [13], [20]. As a result, a short TCP flow may complete all data transfer before it could ramp up its transmission rate to fully utilize the bandwidth available. As an extreme example, if one transfers 100 KB data using Cubic – the current Linux default TCP variant, over a network with 100 Mbps bandwidth and 100 ms RTT, then the average throughput achieved is only 2.5% of the bandwidth available.

Note the naïve solution of increasing the CWnd further is not practical either, as too large an initial CWnd can also cause congestion in low-bandwidth network paths. Clearly, this problem is unsolvable without any information on the network path at the beginning of a TCP flow.

In contrast to TCP's original design, where each TCP flow is independent, most Internet applications initiate multiple successive TCP flows in a single application session. These TCP flows are likely to experience very similar network conditions, and thus much can be learned from a previous TCP flow to the same peer host. This motivates us to develop a novel Stateful-TCP mechanism where path bandwidth is estimated in a previous flow to instantly ramp up the transmission rate of the subsequent flow to the same destination. This eliminates the need for Slow-Start altogether, thus enabling TCP to efficiently utilize the available bandwidth right from the beginning.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Maaz Rehan.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

To explore Stateful-TCP's performance gains over conventional Slow-Start, we applied Stateful-TCP to Cubic to form a new S-Cubic implementation in Linux. We then conducted extensive emulated as well as real-world Internet experiments to evaluate its performance. Results from Internet experiments conducted using 24 client-server pairs in Google/Tencent cloud located in 11 cities around the world showed that compared to Cubic, S-Cubic could reduce flow completion time (FCT) by over 50% on average. A further benchmark test by an independent performance benchmarking company, using over 1,000 end-user clients spread across nine provinces in China over a period of one week, showed that S-Cubic could reduce FCT by 37.5% and increase throughput by over 50% compared to Cubic.

As opposed to an entirely new TCP design, Stateful-TCP is designed to complement the congestion control and error control algorithms, and thus could potentially be applied to many of the existing TCP variants, as well as incorporated into future TCP designs. This paper presents Stateful-TCP's principles, illustrates its application via S-Cubic, and evaluates its potential performance gains and tradeoffs.

The rest of the paper is organized as follows: Section II reviews some previous related works; Section III revisits TCP's Slow-Start mechanism and demonstrates the limitations of existing solutions; Section IV presents the Stateful-TCP mechanism and its application to Cubic; Section V presents experimental results for S-Cubic; Section VI summarizes the paper and outlines some future work.

II. BACKGROUND AND RELATED WORKS

The Transmission Control Protocol has evolved a long way since its introduction. For example, Linux kernel 4.20 implements a total of 17 TCP variants designed for different network environments, with Cubic [2] being the current default, replacing the early variant TCP Reno [22] due to Cubic's better performance over large-BDP networks [2]. Over the years, many new TCP variants had been developed. For example, Westwood [3], Veno [4], and BBR [5] were designed to mitigate the impact of random loss in mobile networks; TCP-Hybla [6] was developed for satellite network with long delay and high loss rate; DCTCP [7] was developed for datacenter networks; Sprout [8] was developed for delay-sensitive applications in mobile networks; PCC [9] was designed to adapt its congestion control behavior based on in-band network measurements, and Copa [10] was designed to achieve low delay while competing fairly in the presence of loss-based competing TCP flows.

Most previous works focused on TCP's congestion control algorithm. However, irrespective of the congestion control algorithm, TCP almost always begins a new flow with the conservative Slow-Start to progressively explore the path bandwidth. This is a necessity as path bandwidth is not known, but it can also become a significant bottleneck, as discussed earlier. This motivates researchers to develop new mechanisms to tackle the Slow-Start bottleneck.

For example, Hauger *et al.* [11] proposed Quick-Start, where TCP's sending rate is adjusted based on explicit feedbacks from network routers. Liu *et al.* [12] proposed Jump-Start, where the initial CWnd is set to the receiver's advertised window (AWnd) size and applied pacing to control the initial transmission rate. Li *et al.* [13] further extended Jump-Start to include proactive retransmission to reduce FCT in case of packet losses towards the end of a flow. In a more recent work, Nie *et al.* [14] proposed TCP-WISE, which employed proactive learning at the server-side to dynamically assign the initial CWnd for each new flow based on historical data. In a different approach, Winstein and Balakrishnan [15] proposed Remy without a Slow-Start phase at all by exploiting knowledge of the network to optimize its congestion-control algorithm.

Although Slow-Start begins with a small CWnd, it grows the CWnd exponentially fast, and in some network settings, it may also overshoot the link capacity, causing unnecessary congestion. To address this problem, researchers proposed additional conditions to modify the exit point of Slow-Start. For example, Floyd [23] proposed Limited Slow-Start, which introduced a new parameter called max_ssthresh (default to 100). Once the CWnd grows larger than max_ssthresh but smaller than ssthresh, CWnd will be increased by at most max_ssthresh/2 per RTT. Cavendish *et al.* [24] proposed Cap-Start to take the network interface card's (NIC) bandwidth into consideration and only activate Limited Slow-Start if the NIC bandwidth is higher than the estimated path bandwidth. Ha and Rhee [16] proposed Hystart to exit the Slow-Start phase early when CWnd has reached the estimated network BDP. Hystart has since been adopted in the Linux kernel and is enabled by default. In a more recent study, Huang [25] showed that Hystart's exit could sometimes be premature due to RTT fluctuations. They proposed Hystart++ to integrate Limited Slow-Start into Hystart. When it triggers, instead of entering the congestion avoidance phase, Hystart++ updates CWnd according to Limited Slow-Start until packet loss occurs or ssthresh is exceeded.

Paradoxically, on the one hand, Slow-Start is often a bottleneck due to the small initial CWnd. On the other hand, Slow-Start could also induce congestions if CWnd grows too much too fast. Therefore TCP's startup phase is still far from a solved problem. Without information on the network path, it is clear that the problem is unlikely to be even solvable. We argue that further optimization of TCP's startup phase must go beyond initial CWnd optimization. For instance, in 5G networks which promises bandwidth in excess of Gbps, even with a RTT of 20 ms, at 1 Gbps bandwidth, the CWnd required to fully utilize bandwidth will be 2.5 MB. Such a large initial CWnd could lead to significant congestion losses if left unchecked.

III. TCP SLOW-START REVISITED

In this section, we first analyze TCP Slow-Start's performance impact and then evaluate some existing as well as

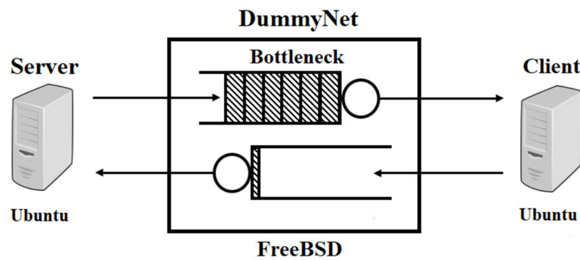


FIGURE 1. The experiment setup. The bottleneck link is emulated using DummyNet [26].

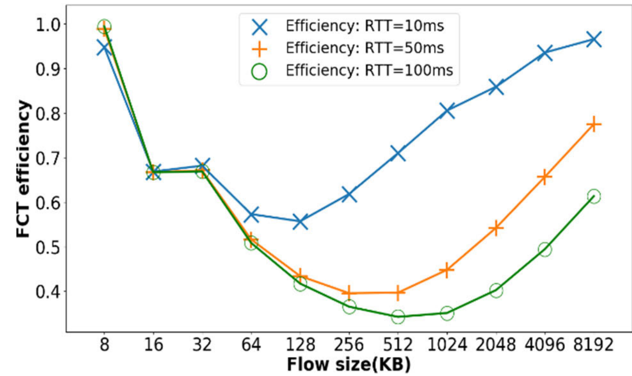


FIGURE 2. FCT efficiency of Cubic versus flow size.

intuitive solutions to tune the initial CWnd to mitigate Slow-Start's limitations.

A. PERFORMANCE IMPACT

In Slow-Start, TCP begins with a small initial CWnd (e.g., 10 MSS in current Linux kernel) and then doubles it every RTT in an exponential manner as ACKs are received. It exits the Slow-Start phase when the CWnd exceeds the Slow-Start threshold (ssthresh), Hystart [16] (if enabled) triggers, or a loss event occurs [21].

To illustrate the performance impact of TCP Slow-Start, we conducted experiments using the topology depicted in Fig. 1. We used Ubuntu with Linux kernel 5.3 as the server and client OS, and employed DummyNet [26] to emulate a bottleneck link with 100 Mbps bandwidth and RTT of 10 ms, 50 ms, and 100 ms [27]. There is no random loss, and the link buffer size was configured to one BDP. The flow size ranges from 8 KB to 8.2 MB, covering most of the range in the Internet's flow size distribution [20], [28].

For performance comparison, we first calculate the flow completion time (FCT), denoted by T, defined as the time from the client (receiver) sends the TCP SYN segment to the time all data are received, and then compare it against the minimum FCT, denoted by $T_{\min}$, defined as

$$T_{\min} = 1.5d_{\min} + \frac{S}{C} + 0.5d_{\min}, \tag{1}$$

where $d_{\min}$ is the two-way propagation delay (assumed to be symmetric), S is the flow size (inclusive of TCP/IP header overheads), and C is the bottleneck link bandwidth. The three terms on the R.H.S. represent the: (i) time spent in three-way handshaking; (ii) transmission time; and (iii) one-way propagation delay for the last TCP data segment to reach the receiver.

Thus $T_{\min}$ represents the minimum FCT achievable in the given network path for *any* TCP protocol. Using it, we then define a metric called *FCT efficiency*, denoted by Ω,

$$\Omega = \frac{T_{\min}}{T}, \tag{2}$$

where it can range from 0 to 1, with 1 representing optimal performance. A desirable property of FCT compared to throughput/utilization is that it accounts for propagation delay and overheads in TCP's three-way handshaking, which more accurately measures the performance perceived by the user/application.

Fig. 2 plots the FCT efficiency of Cubic versus flow size for three different RTT settings. The result for each setting is averaged over ten repeated experiment runs.

We observe that Cubic's FCT efficiency exhibited a U-shape characteristic with respect to flow size. Beginning with small flow size, as Linux TCP has a default initial CWnd of 10 MSS, flow size no larger than 10 MSS (e.g., 8 KB) can be transmitted entirely in the first RTT, thereby achieving close to optimal FCT efficiency. Note that Cubic did not achieve an FCT efficiency of exactly 1 in the experiment as the minimum FCT in (1) does not account for host processing delays.

As flow size increases, Cubic's efficiency drops as its CWnd is smaller than the path's BDP, preventing it from fully utilizing the available bandwidth. Its efficiency progressively improves as flow size increases further due to CWnd growth. Cubic's efficiency is also sensitive to propagation delay as one would expect, degrading more in the 50 ms and 100 ms cases. In these two cases, Cubic's efficiency is lower than 0.5 over a wide range of flow sizes (e.g., 128 KB to 1024 KB).

B. INITIAL CWND TUNING

To tackle TCP's Slow-Start limitation, researchers have developed novel ways to tune the initial CWnd [11]–[14] to increase the initial transmission rate. We evaluate some of the existing solutions in this section. We first implemented Jump-Start [12] into Linux as it is not currently supported. We did not implement Quick-Start [11] and TCP-WISE [14], as the former requires network router support, which is not generally available in the current Internet. For the latter, we did not have sufficient information to recreate the training algorithm or the experiment settings. Nonetheless, we created a similar scheme called TCP-Cache, which also exploits historical information for setting the initial CWnd to serve as a comparison.

We first evaluate Jump-Start, which sets the initial CWnd to the size of the initial AWnd reported by the peer and the initial pacing rate to AWnd/RTT, where both AWnd and RTT

TABLE 1. Summary of Linux Kernel Configurations.

Parameters	Jump Start	Cache	Cache+	Cubic	S-Cubic
Initial CWnd	AWnd	Cached CWnd	Cached CWnd	10	See text
TCB [17]	Yes	Yes	Yes	Yes	Yes
AWnd	Yes	Yes	See text	Yes	See text
Pacing	Yes	No	No	No	Yes
TFO [29]	No	No	No	No	No
SACK [30]	Yes	Yes	Yes	Yes	Yes
Hystart [16]	Yes	Yes	Yes	Yes	Yes
CC algorithm	Cubic	Cubic	Cubic	Cubic	Cubic

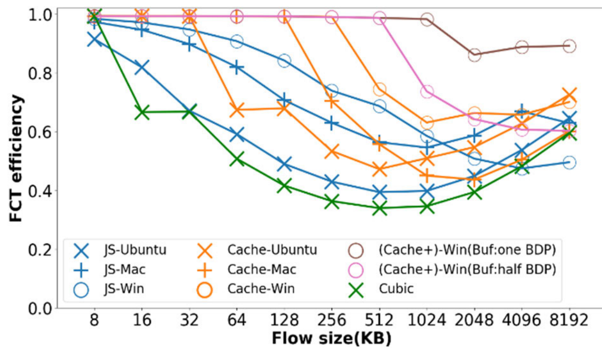


FIGURE 3. Comparison of FCT efficiencies for Jump-Start (JS), TCP-Cache (Cache), and TCP-Cache+ (Cache+) for three client operating system platforms.

are obtained during three-way handshake. The pacing ends once the first ACK is received.

We applied Jump-Start to Cubic and tested it in the topology in Fig. 1 with 100 Mbps bottleneck-link bandwidth, 100 ms RTT, and 1.25 MB (one BDP) bottleneck-link buffer size. Table 1 lists the key Linux kernel settings adopted by the various algorithms. Three operating system (OS) platforms were tested as the client receiver: Ubuntu Linux (kernel 5.3), Microsoft Windows 10 (Enterprise 2018), and Mac Mojave (10.14). The initial AWnd differs across different OS's: 20 MSS, 179 MSS, and 90 MSS for Ubuntu, Windows 10, and Mac, respectively. We also found that even for the same OS, the initial AWnd can also vary (e.g., two different Windows 10 hosts exhibited initial AWnd of 44 and 179 MSS, respectively. We adopted the larger one in our experiments).

Fig. 3 plots the FCT efficiency of Jump-Start versus flow size. Each data point is computed from the average of ten experiment runs. There are three observations. First, regular Slow-Start outperformed Jump-Start at 8 KB flow size. This is due to Jump-Start's initial pacing mechanism, which even in the best case (with a Windows 10 peer), was limited to 179 MSS / 100 ms $\approx$ 21.5 Mbps. In comparison, with an initial CWnd of 10 MSS, Cubic can send the entire 8 KB flow at the line rate (i.e., 1 Gbps).

Second, as flow size increases, Jump-Start's efficiency pulls ahead of Cubic, especially in the case with a Windows 10 client receiver. Nevertheless, its efficiency is still far from optimal over a broad range of flow sizes.

TABLE 2. TCP Parameters in the TCP Control Block (TCB) [17].

Variables	Description	Cached
RTT	Smoothed RTT	Yes
RTTvar	RTT variance	Yes
ssthresh	Slow-Start threshold	Yes
Reordering	The threshold of "holes" in SACK for triggering retransmission	Yes
CWnd	Congestion window	Yes

Lastly, we observe that Jump-Start with Windows 10 (JS-Win) behaves abnormally for flow sizes larger than 512 KB – its efficiency degrades (instead of improves) as flow size increases. Our investigation suggested that this may be due to interactions between Jump-Start and Hystart [16], which may have caused it to exit the Slow-Start phase prematurely.

In addition to matching the initial CWnd to AWnd, another logical approach is to *reuse* it from the *past* connection to the *same* peer. The idea is that the network path to the same peer is likely to remain the same, and so the previous connection would have already explored the path to arrive at a good CWnd. A similar idea has been exploited by the TCP Control Block (TCB) [17], which caches five internal TCP parameters that can be used in a new connection (Table 2).

For a new connection to the same peer (identified by IP address), current Linux already reuses: *RTT* to set the initial timeout value; *ssthresh* to control the exit point for Slow-Start; *Reordering* to set the threshold for triggering retransmission under SACK. Interestingly, *RTTvar* and *CWnd* are also cached. However, from our analysis of the Linux kernel source codes, neither has any effect on subsequent flows.

Nevertheless, given the availability of TCB, a logical extension would be to reuse the last recorded value of CWnd from the previous connection to set the initial CWnd for a new connection. We implemented this (henceforth called TCP-Cache) into the Linux kernel and repeated the previous experiments to evaluate its performance.

Specifically, we first ran a Cubic flow, followed by ten consecutive TCP-Cache flows. FCT is then computed from the average of the ten TCP-Cache flows. The results, also plotted in Fig. 3, shows that TCP-Cache generally achieved higher efficiency than its Jump-Start counterpart. However, similar to Jump-Start, its efficiency is highly client-OS-dependent. For example, while TCP-Cache can achieve high efficiency with Windows 10 client, its performance is only slightly better than Jump-Start (and Cubic) with Ubuntu client.

This is unexpected as the cached CWnd value should have no dependency on the client platform in a new connection. Further investigation revealed that it is not due to the cached CWnd, but due to the initial AWnd of the client. This is because the maximum number of packets inflight is limited not only by CWnd but also by AWnd. As the initial AWnd is platform-dependent, it not only renders

TCP-Cache's performance OS-dependent but also limits its achievable efficiency as the maximum initial transmission window can never exceed AWnd. For example, a TCP flow after transferring 1 MB data could grow CWnd to about 700 MSS. However, even with Windows 10 client, the initial AWnd is only 179 MSS, which severely limits the initial transmission rate.

C. THE CWND DILEMMA

One potential solution to the AWnd limitation is to *suppress* the receiver AWnd (except when it equals zero) in calculating the transmission window. This was first proposed by Liu *et al.* [18] in their TCP accelerator to speed-up TCP performance over mobile networks. The key idea is that flow-control (which AWnd is intended for) is rarely needed in today's computing devices as most can process incoming data packets much faster than the typical data rate. Therefore, even if the sender transmits more in-flight packets than the receiver's AWnd, buffer overflow is still unlikely to occur [18].

To apply this technique, we modified TCP-Cache into TCP-Cache+ such that the transmission window is now simply equal to CWnd (unless AWnd is zero, in which case transmission suspends as usual). The results for a Windows 10 client are also plotted in Fig. 3 with the curve named Cache+. We can observe that TCP-Cache+ outperforms TCP-Cache for longer flows. Its efficiency, however, degrades for flow sizes larger than 1024 KB. Moreover, in another experiment (not shown) with the link buffer size reduced to half BDP, its efficiency degrades earlier and more severely.

To investigate the cause, we analyze TCP-Cache+'s CWnd evolution and transmission rate for an 8.2 MB flow in Fig. 4. We observe that the CWnd had grown to a large value (>1200 MSS) by the time the first flow ended (red curve). This became the initial CWnd for the next TCP-Cache+ flow (green curve). The problem is that with the large initial CWnd, the new flow then transmitted one CWnd's worth of data at the *line rate* (i.e., 1 Gbps) right from the beginning. This resulted in a sharp spike in the transmission rate at around 1.6 s in Fig. 4. Not surprisingly, such a massive data burst resulted in buffer overflow at the bottleneck link. Consequently, TCP reduced the CWnd significantly, resulting in a much lower transmission rate. In retrospect, this is precisely why Slow-Start was introduced at the start of a new TCP flow. This presents a dilemma as setting the initial CWnd conservatively could severely limit performance while setting it more aggressively may induce unnecessary congestions.

IV. STATEFUL-TCP

In this section, we propose a new Stateful-TCP mechanism to resolve the initial CWnd dilemma and to bypass the Slow-Start phase altogether to ramp up a new TCP flow's initial transmission rate right from the beginning.

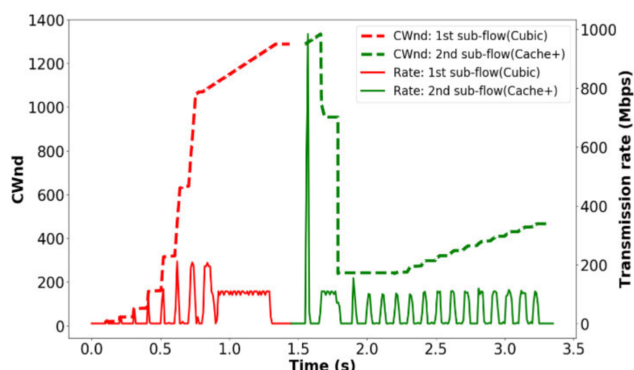


FIGURE 4. Congestion caused by a Cache+ flow (green) due to the large initial CWnd and massive initial transmission burst (at around 1.6 s).

A. PRINCIPLE

The fundamental dilemma in Slow-Start is that too small a CWnd will lead to bandwidth under-utilization. However, a sufficiently large CWnd may also lead to buffer overflow at the bottleneck link. A key insight to solving this dilemma is that the large CWnd is not the problem – it is the initial burst transmission that is causing the buffer overflow.

As a counter-example, if one runs an extended TCP flow, its CWnd can also grow to a large value, but that does not necessarily cause buffer overflow. This is because at steady-state, TCP's packet transmissions are *clocked* by the ACKs returned from the receiver, which in turn are clocked by the bottleneck link. Therefore, transmissions are *paced* in accordance with the bottleneck link bandwidth. In contrast, there is no such pacing at the beginning for TCP-Cache and TCP-Cache+, resulting in a massive transmission burst.

Therefore, the goal of Stateful-TCP is to enable a new TCP flow to operate in the same way as if it is a *continuation* of the previous flow to the same destination, i.e., bypass Slow-Start and transmit at the path bandwidth right from the beginning. The next section presents the system design to accomplish this goal.

B. SYSTEM DESIGN

Stateful-TCP operates in three phases: Startup phase, Estimation phase, and Termination phase, as depicted in Fig. 5. All three phases are implemented inside the TCP sender, so no change to the TCP receiver is needed. A hash table is used to cache states from a completed flow. Each table entry comprises three fields: peer IP address, estimated bandwidth, and minimum RTT.

1) STARTUP PHASE

After the three-way handshake is completed, the sender will first check if bandwidth estimated from a previous flow already exists for the peer. Specifically, the peer's IP address will be hashed into an H-bit index where H is determined by the hash table size (e.g., $H = 20$ bits in our experiments). The hash table entry is then looked up to determine one of three courses of actions:

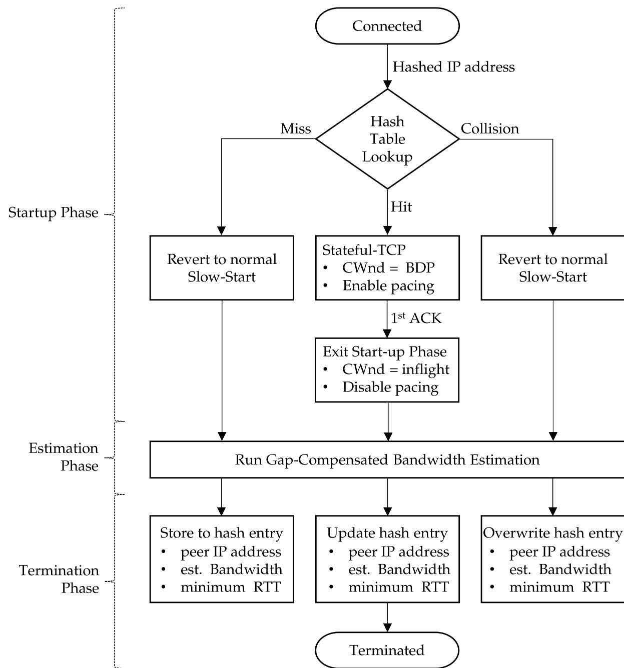


FIGURE 5. Stateful-TCP control flow.

Miss – The hash table entry is empty, i.e., no record of a previous flow, so it reverts to regular Slow-Start.

Hit – The hash table entry is non-empty and the previous flow's IP address matches the new one so that Stateful-TCP will be activated for the new flow by: (a) setting CWnd to the previous flow's BDP, i.e., estimated bandwidth × minimum RTT; (b) suppressing AWnd (except when 0); and (c) activating pacing for outgoing packets at a rate equal to the previous flow's estimated bandwidth.

Collision – The hash table entry is non-empty but the previous flow's IP address differs from the new one, suggesting hash collision. In this case, the new flow will revert to regular Slow-Start.

In both hash table miss and collision cases, states of the new TCP flow will be stored to the table entry when the flow terminates. One exception is when the estimated BDP is equal to or smaller than TCP's default initial CWnd, denoted by $CWnd_{init}$ (10 MSS in Linux), in which case the states will be discarded as there will be no performance gain over regular Slow-Start.

The Start-up phase ends when the first ACK is received, at which point pacing will be disabled, and CWnd will be set to

$$CWnd = \max\{\kappa, CWnd_{init}\}, \tag{3}$$

where κ is the current number of packets in-flight.

The idea is that the current number of packets inflight reflects the path's BDP, which is what CWnd should be in order to fully utilize the available bandwidth. We set a minimum of $CWnd_{init}$ for CWnd so that it will not be lower than regular Slow-Start.

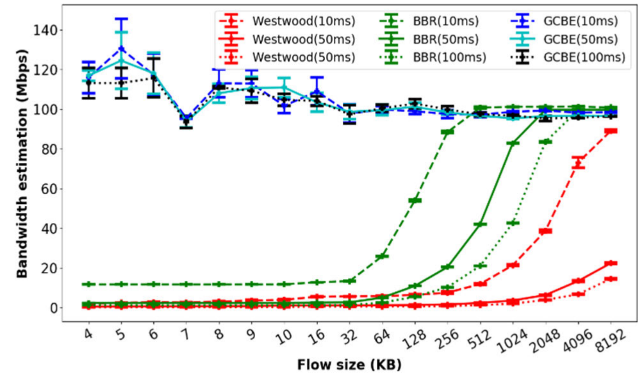


FIGURE 6. Comparison of bandwidth estimation versus flow size with bottleneck link bandwidth of 100 Mbps and link buffer size of one BDP.

2) ESTIMATION PHASE

This phase follows the Startup phase immediately and lasts until the TCP flow is terminated. During which time, the TCP sender continuously estimates the path bandwidth (c.f. Section IV-C). The estimation process operates independently from the congestion control algorithm so that the latter's operations are *not* changed except for one aspect – the AWnd is suppressed except for 0 so that it will not become the bottleneck (c.f. Section III-C).

3) TERMINATION PHASE

Upon connection shutdown, e.g., receiving a FIN segment from the client, Stateful-TCP will store the estimated bandwidth, the minimum RTT, and the IP address of the client into the hash table entry.

C. GAP-COMPENSATED BANDWIDTH ESTIMATION

A key component in Stateful-TCP is the bandwidth estimator, which runs throughout the Estimation phase. The problem of path bandwidth estimation is not new. In fact, bandwidth estimation is an integral part of some notable TCP variants, such as Westwood [3] and BBR [5]. The general principle is similar, i.e., estimate bandwidth according to the amount of data acknowledged within a certain period of time (e.g., one smoothed RTT). However, when applied to Stateful-TCP, these algorithms could suffer from underestimation.

To demonstrate the potential problem, we first investigate Westwood and BBR's bandwidth estimation algorithms by adding instrumentation codes into their Linux implementation to record the bandwidth estimated upon flow termination. We conducted experiments using the topology in Fig. 1 with bottleneck link bandwidth of 100 Mbps and bottleneck link buffer size of one BDP. Both client and server ran Ubuntu with Linux kernel 5.3.

Fig. 6 plots the bandwidth estimated by Westwood and BBR for flow sizes ranging from 4 KB to 8.2 MB. The first observation is that both algorithms significantly underestimate the link bandwidth (100 Mbps) for smaller flow sizes. For example, Westwood's estimated bandwidth begins to converge to the actual link bandwidth only at the largest

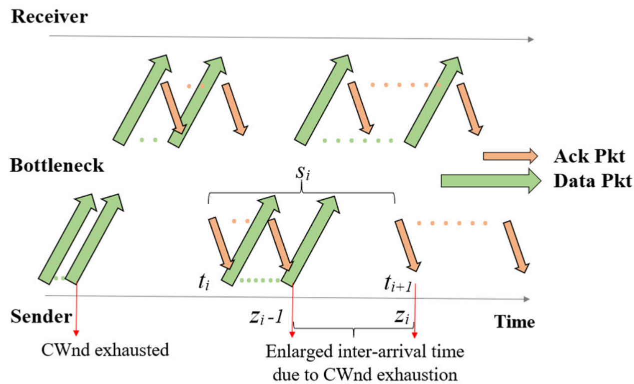


FIGURE 7. Illustration of bandwidth underestimation caused by CWnd-limited transmission.

flow size of 8 MB even under the shortest RTT of 10 ms. This is due to Westwood's use of low-pass filtering in smoothing the bandwidth estimates, which increased the convergence time.

In comparison, BBR's bandwidth estimates converge much faster, but even in the best case (with RTT = 10 ms) still requires a flow size of at least 512 KB to converge. This is undesirable as the majority of Internet flows are relatively short [20], [28]. Thus applying the underestimated bandwidth to initial pacing will limit the efficiency of the new flow.

To tackle this problem, we first need to identify the cause of the underestimation. Our investigations reveal that this is caused by *CWnd-limited* transmission, as illustrated in Fig. 7. Let t_i be the start time of estimation cycle i. Whenever an ACK is received, the sender will check if the elapsed time since t_i is equal to or longer than the current smoothed RTT, denoted by d,

$$t_{i+1} - t_i \geq d, \quad \forall i, \tag{4}$$

and if so, will mark the end of cycle i (and beginning of cycle $i+1$) to trigger bandwidth estimation. Bandwidth is then estimated from the average throughput in cycle i, i.e.,

$$c_i = \frac{s_i}{t_{i+1} - t_i}, \tag{5}$$

where s_i is the total number of bytes acknowledged by ACKs in the cycle, and c_i is the bandwidth estimate for cycle i.

An implicit assumption is that s_i accurately reflects the *maximum* amount of data that can pass through the bottleneck link during the estimation cycle. However, the sender may *suspend* transmission if it has exhausted the CWnd (or AWnd), as illustrated in Fig. 7, resulting in an extended gap between ACK arrival times. If this gap is not accounted for, then the estimated bandwidth will be lower than the actual one. Given that TCP begins with a small CWnd (10 MSS in Linux), it explains the underestimation observed in Westwood and BBR in Fig. 6.

To compensate for the transmission gap due to CWnd exhaustion, we propose a new gap-compensated bandwidth estimation (GCBE) method to exclude the gap induced by transmission suspension from the estimation. Specifically, let n_i be the number of ACK packets in estimation cycle i; $h_{i,j}$ and $a_{i,j}$ be the reception time and the number of bytes acknowledged by ACK j ($j = 0, 1, \ldots, n_i - 1$) in cycle i respectively.

We observe that transmission suspension will result in abnormally long inter-arrival time between two ACKs. Therefore we can exclude it from bandwidth estimation by first identifying the ACK with the longest inter-arrival time in cycle i, denoted by z_i, from

$$z_i = \arg\max_j \left\{ h_{i,j} - h_{i,j-1} \middle| j = 0, 1, \ldots n_i - 1 \right\}, \tag{6}$$

and then excluding both the bytes it acknowledges and the time it spans from the bandwidth estimation:

$$c_i = \frac{\sum_{k=1}^{n_i-1} a_{i,k} - a_{i,z_i}}{\left(h_{i,n_i-1} - h_{i,0}\right) - \left(h_{i,z_i} - h_{i,z_i-1}\right)}, \tag{7}$$

Note that the first ACK, i.e., $a_{i,0}$, is also excluded from (7) as its transmission time is not known. Similarly, when excluding the gap caused by ACK z_i, both the amount of data it acknowledged, i.e., a_{i,z_i}, and its inter-arrival time, i.e., $\left(h_{i,z_i} - h_{i,z_i-1}\right)$, are excluded.

However, it is also possible that there is no transmission gap in the measurement cycle, e.g., CWnd is larger than BDP. In that case, the estimation in (6) and (7) will still work as it merely removes one sample from the measurement cycle.

GCBE keeps the most recent L ($L = 5$ in our experiments) bandwidth estimates and then stores the maximum one for use in the next flow when the current flow terminates, i.e.,

$$c^* = \max \left\{ c_i | i = m - L, \ldots, m - 1 \right\}, \tag{8}$$

where m is the total number of measurement cycles. This is to prevent isolated loss events from skewing the bandwidth estimate.

We implemented GCBE inside the Cubic pluggable congestion control module in Linux. Fig. 6 compares its accuracy against the ones in Westwood and BBR. It is clear that GCBE does not suffer from underestimation due to CWnd-induced transmission suspension and can obtain a good estimate of the bottleneck link bandwidth for flow size as small as 4 KB. This is significant as in many applications (e.g., web), the initial TCP flow is often a short one (e.g., base HTML file). By employing GCBE in its estimation phase, Stateful-TCP can ramp-up the transmission rate of the second flow even if the first flow is very short.

D. LINUX IMPLEMENTATION

We applied Stateful-TCP to Cubic in Linux kernels from version 4.9 to 5.3, by modifying Cubic's pluggable congestion control module to form Stateful-Cubic (henceforth called S-Cubic). We chose this approach instead of modifying the kernel as Linux's pluggable congestion control module can be easily installed/switched at runtime without recompiling the kernel or even rebooting the OS.

The hash table for storing bandwidth estimates is dynamically allocated and initialized inside the congestion control module at the time the S-Cubic module is registered with the kernel. The hash table's memory requirement is relatively modest, considering that only three states {IP address, estimated bandwidth, and minimum RTT} are stored for each entry. In our experiments, we typically allocate one million entries for the hash table. Early results from deploying S-Cubic into a production server serving requests for an app-store in our collaborator's datacenter show that hash collision is reasonably low.

It is worth noting that another way to store the data would be to extend the TCP Control Block [17] with new data fields. However, this requires modification to the kernel codes and thus is less desirable from the deployment's point of view.

The S-Cubic module can co-exist with all other Linux TCP implementations, including the original Cubic. Switching between the different TCP implementations can be done simply by issuing a few Linux commands in a console. This greatly facilitates experimentation as one can easily alternate between, e.g., Cubic and S-Cubic, in a back-to-back manner using the same server to compare their performance within a short time to ensure that the network conditions are as close as practicable.

V. PERFORMANCE EVALUATION

In this section, we evaluate the performance impact of applying Stateful-TCP to Cubic via extensive emulated and real Internet experiments. The TCP configurations are summarized in Table 1 under the Cubic and S-Cubic columns.

A. FLOW COMPLETION TIME

We first evaluate and compare in Fig. 8 S-Cubic's FCT efficiency to Cubic, Jump-Start, and Cache+ using the same topology in Fig. 1 for three RTT settings (10 ms, 50 ms, 100 ms [27]), and two link buffer size (half BDP and one BDP) settings. Each data point is an average of 10 experiment runs. There are three observations.

First, S-Cubic performed consistently across different flow sizes, RTTs, and link buffer sizes, achieving close to optimal FCT efficiency. For example, S-Cubic's average FCT efficiencies are 96.3%, 98.8%, and 99.3% for RTTs of 10 ms, 50 ms, and 100 ms, respectively (with a link buffer size of one BDP). This is a remarkable result given that S-Cubic primarily modifies Cubic's behavior in the first RTT only and then reverts to the Cubic algorithm after that.

Second, we observe that S-Cubic's FCT efficiency is slightly lower in RTT of 10 ms compared to 50 ms and 100 ms. A careful analysis of the packet traces suggested that this is due to processing delay at the end systems (i.e., Linux hosts), which are not accounted for in computing the minimum FCT in (1). These become more significant (compared to the flow duration) when the RTT is short.

Third, comparing the results for the two link buffer sizes, we observe that Cache+ is relatively sensitive to the parameter. This is because Cache+ reuses the CWnd from the previous connection, and when the CWnd is sufficiently large, the large initial transmission burst could cause buffer overflow at the bottleneck link. S-Cubic avoids this problem by pacing transmission in the first RTT at the previously estimated bandwidth as if it is ACK-clocked.

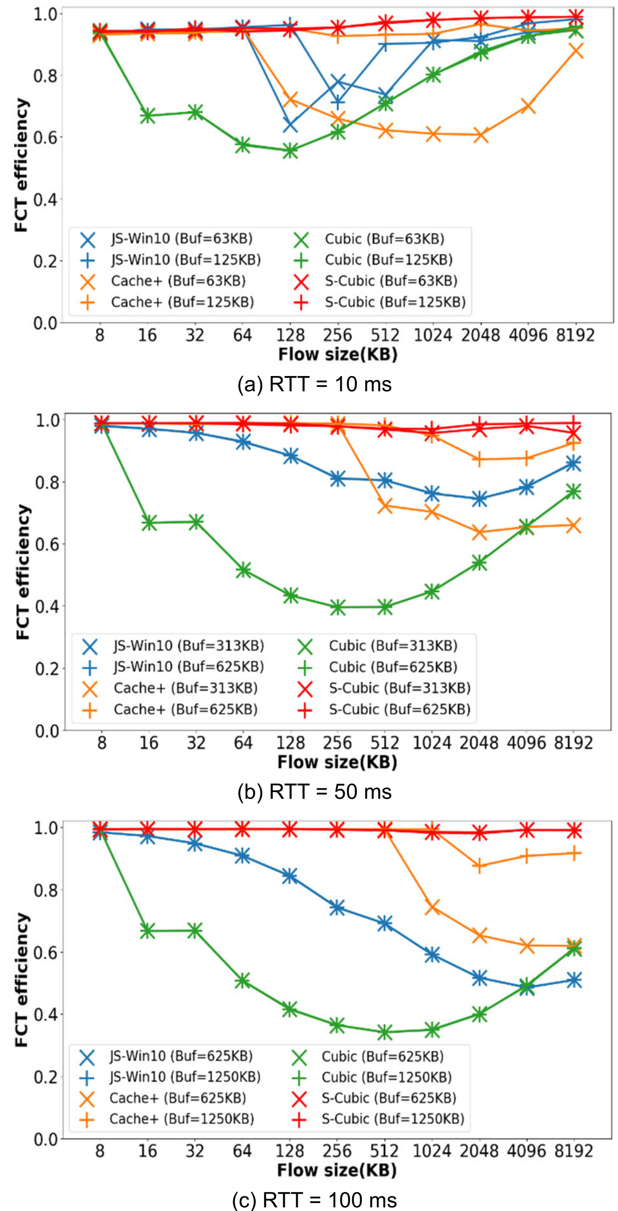


FIGURE 8. Comparison of FCT efficiencies versus flow size.

TABLE 3. Comparison of Per-flow Mean Queuing Delay (in ms).

Link buffer	Half BDP			One BDP		
RTT(ms)	10	50	100	10	50	100
Cubic	1.5	4.7	6.2	2.7	5.7	6.6
S-Cubic	1.2	1.8	3.7	2.0	5.5	6.6

B. QUEUING DELAY

In the same experiments in Section V-A, we also recorded the per-flow queueing delays. From that, we calculated the mean

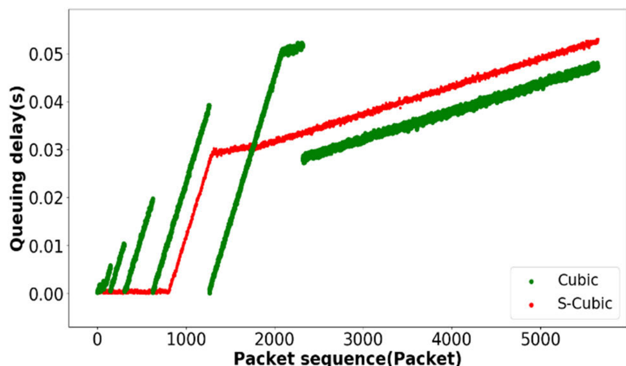


FIGURE 9. Comparison of the per-packet queueing delay evolution between a Cubic flow and an S-Cubic flow.

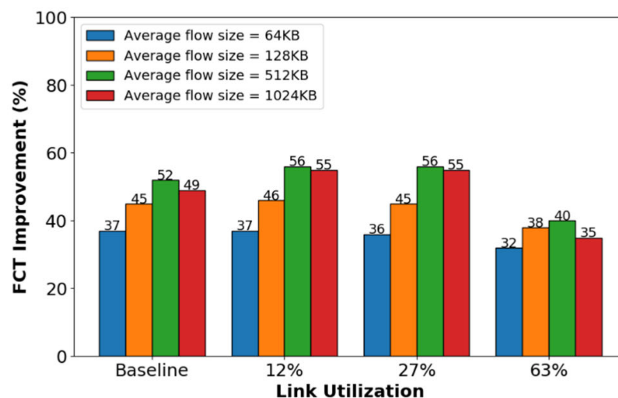


FIGURE 10. S-Cubic's FCT improvements over Cubic under three different link utilizations and four average flow sizes. Baseline refers to the case of no competing flows.

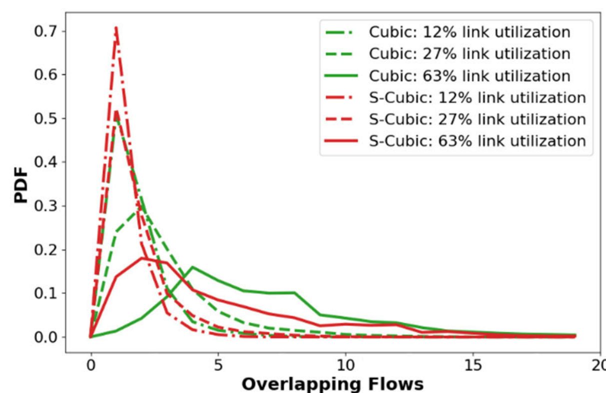


FIGURE 11. Probability density functions for the number of overlapping flows competing in the bottleneck link.

packet queueing delays and summarized them in Table 3 for the three RTT and two link buffer size settings. Surprisingly, although S-Cubic generally transmits at higher average datarate, its queueing delays turn out to be even shorter than Cubic in most cases. This counter-intuitive result is due to S-Cubic's transmission pacing.

Specifically, we plot in Fig. 9 the sequence of per-packet queueing delays for both S-Cubic and Cubic for an 8.2 MB flow in a link with 100 Mbps bandwidth, 100 ms RTT, and 1250 KB buffer size. This figure shows a desirable behavior of S-Cubic, i.e., in the first RTT of ~800 packets, S-Cubic maintained consistent and very short packet queueing delays due to its pacing mechanism. Consequently, for flows that are shorter than one BDP, the queueing delay experienced in S-Cubic will be significantly shorter than Cubic (e.g., 0.4 ms vs. 6.9 ms for S-Cubic and Cubic, respectively). Beyond the first RTT, S-Cubic reverts to Cubic so exhibited similar packet delay behavior.

Overall, S-Cubic, despite its far higher FCT efficiency, achieved similar packet delay as Cubic and significantly shorter packet delay for flows shorter than one BDP. It is worth noting that the overall delay performance is still dominated by Cubic's congestion control algorithm, especially for longer flows. The Stateful-TCP mechanism itself does not induce higher packet delays nor reduce it beyond the first RTT. Therefore a fruitful future research direction would be to integrate Stateful-TCP into other TCP variants that are optimized for delay-sensitive traffics [8] or for reducing buffer-bloat [5], [8], [10].

C. SHARED BOTTLENECK LINK

The previous experiments were based on a single TCP flow. That offers a controlled and predictable environment to compare and analyze the behavior of S-Cubic. In practice, a network bottleneck is likely to be shared by multiple TCP flows, and this presents a challenge - the presence of competing flows means that the actual bandwidth available will vary with time [32], [33].

Consequently, the path bandwidth estimated from a previous flow may no longer be the same by the time a new flow is initiated. To investigate the potential performance impact of competing flows, we conducted a second set of experiments with the topology in Fig. 1. Bottleneck link bandwidth, RTT, and buffer size were configured to 100 Mbps, 50 ms, and one BDP, respectively. This time, the client generates new TCP flows to the server according to a Poisson process. The flow sizes were drawn from a Pareto distribution (with $\alpha = 2.5$), which exhibited long-tail characteristics resembling Internet flow distribution [34]–[39].

We conducted experiments with mean flow sizes of 64 KB, 128 KB, 512 KB, and 1024 KB. Each experiment comprises 1,000 flows. To emulate different levels of bottleneck link utilization, we adjusted the Poisson process's mean arrival rate in accordance with the mean flow size to achieve three expected link utilization of 12%, 27%, and 63%, respectively [40], [41]. The same experiments were repeated for Cubic and S-Cubic.

Note that the performance metric FCT efficiency is not applicable in this case as multiple flows dynamically share the bottleneck link bandwidth, so the link capacity C in (1) is undefined. Therefore, we employ another performance metric - FCT improvement, defined as the percentage reduction in

FCT compared to Cubic, in the following experiments. The results are summarized in Fig. 10 for the three link utilizations, plus a baseline case where only one flow is allowed at a time (i.e., same as in Section V-A).

Remarkably, S-Cubic's FCT improvements under shared-bottleneck at 12% and 27% link utilizations are comparable to the baseline case even though multiple TCP flows may overlap in time, which can reduce the cached estimated bandwidth's accuracy. To verify the extent of overlapping flows, we recorded the number of overlapping flows traversing the bottleneck and plotted their probability density functions (PDFs) in Fig. 11.

Clearly, there are overlapping flows for both Cubic and S-Cubic, even at 12% link utilization. More importantly, the number of overlapping flows is much larger in Cubic than in S-Cubic. This is because S-Cubic flows generally complete much faster, thereby reducing the likelihood of flow overlaps.

S-Cubic's FCT improvements exhibited a concave shape across mean flow sizes from 64 KB to 1024 KB. This is due to Cubic's convex-shaped FCT efficiency with respect to flow size, as shown in Fig. 8 earlier. Finally, S-Cubic's FCT improvements did decrease at 63% link utilization. This is due to Cubic's improved efficiency at high link utilization. However, link utilization over 50% is uncommon in practice, e.g., about 70% of the links were never utilized over 50% during a day [40]. Even in this case, S-Cubic still achieved over 30% reduction in FCT compared to Cubic.

D. FAIRNESS AND FRIENDLINESS

We investigate S-Cubic's fairness to itself and friendliness towards Cubic in this section. We first investigate S-Cubic's aggregate behavior and then analyze its temporal dynamics. The experiment topology is similar to Section V-C, except that we added another pair of server-client so that two server-client pairs shared the same bottleneck link of 100 Mbps bandwidth, 50 ms RTT, and buffer size of one BDP.

We ran three sets of experiments with the number of clients receiving S-Cubic flows ranging from 0 (i.e., all Cubic) to 2 (i.e., all S-Cubic). We measured the mean FCT for each of the two clients for four mean flow sizes and plotted the results in Fig. 12(a) to 12(c) for three link utilizations. These results compare the aggregate behavior of Cubic and S-Cubic flows. Note that each bar has a top and a bottom part, representing the mean FCT of the two clients respectively.

We first consider fairness by comparing the two clients' mean FCT for the two cases of all-Cubic and all S-Cubic flows (i.e., first and third groups of bars in Fig. 12). It is evident that both Cubic and S-Cubic achieved a high degree of fairness towards their own competing flows. For example, even at the highest link utilization of 63% (Fig. 12(c)), Cubic's mean FCTs for 1 MB mean flow sizes were 0.73 s and 0.75 s across the two clients, while the same for both S-Cubic were 0.52 s and 0.55 s. The fairness performance at the two lower link utilization settings is also similar.

Next, we consider S-Cubic's friendliness to Cubic by comparing the FCT of the client receiving Cubic flows with the client receiving S-Cubic flows (i.e., the middle group of bars in Fig. 12). As expected, Cubic's mean FCTs increase when competing with S-Cubic flows, as S-Cubic flows can ramp up their transmission rates much more quickly than Cubic flows. Overall, Cubic flows' FCTs averaged over all four mean flow sizes are increased by 12.6%, 21.6%, and 23.1% for 12%, 27%, and 63% link utilizations, respectively.

In addition to aggregate fairness/friendliness, we also investigate the temporal dynamics within an S-Cubic flow

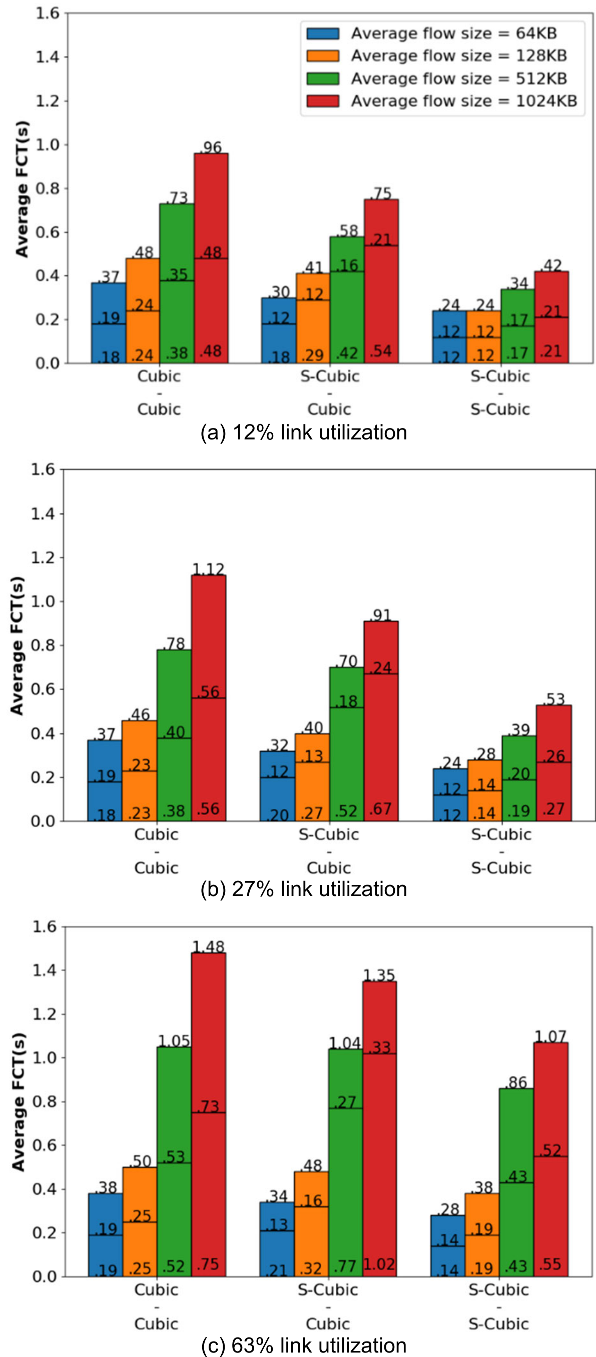


FIGURE 12. Evaluation of aggregate fairness and friendliness in three link utilization settings for Cubic and S-Cubic.

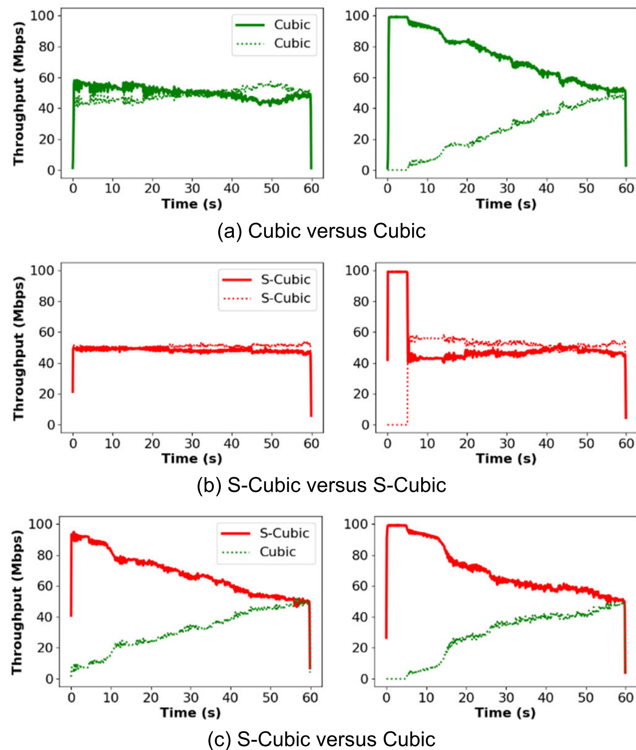


FIGURE 13. Temporal dynamics of two competing flows (left: started simultaneously; right: with 5 s interval). Link bandwidth, RTT, and link buffer size were configured to 100 Mbps, 50 ms, and one BDP (625 KB).

by analyzing its throughput over time. We initiated two TCP flows through the same bottleneck link of bandwidth 100 Mbps, RTT 50 ms, and buffer size of one BDP. For the S-Cubic flow, it used the estimated bandwidth from a previous 8 KB flow to begin its startup phase.

Fig. 13(a) plots the throughput time series for two experiments – the left one with two Cubic flows starting simultaneously and the right one with a Cubic flow starting 5 s after the first one. When started at the same time, the two Cubic flows share bandwidth fairly right from the beginning. However, if the second flow is started 5 s after the first flow, then it takes nearly one minute for the second flow to ramp up to share bandwidth equally with the first flow. This is because by the time the second flow joins, the on-going flow already fully utilizes the link bandwidth. The congestions caused by the competition caused the late-coming flow to exit Slow-Start prematurely, thus severely slowing its CWnd ramp-up.

We repeated the same experiments for S-Cubic and plotted the results in Fig.13 (b). The two S-Cubic flows share bandwidth fairly right from the beginning when they started at the same time. Unlike Cubic, even in the case where the second flow is started 5 s after the first flow, the two flows still converge to the fair-share throughput quickly. This is a desirable property as the late-comer is not deprived of bandwidth by the existing long flow.

For friendliness, we investigate the dynamics of a Cubic flow competing with an S-Cubic flow in Fig. 13(c). For the first case where two flows started at the same time (left figure), the throughput curves are remarkably similar to the case where the second flow starts 5 s after the first one, no matter if the first flow is Cubic (Fig. 13(a), right figure) or S-Cubic (Fig. 13(c), right figure).

This suggests that from the perspective of the second Cubic flow, the competing S-Cubic flow behaves just like an on-going TCP flow that has reached steady-state – a testament to the design principle presented in Section IV-A.

S-Cubic's fast convergence does have a tradeoff. We recorded the number of retransmissions per RTT in the previous experiments and plotted their time-series in Fig. 14 for the two cases of Cubic-vs-Cubic (c.f. Fig. 13(a), right figure), and S-Cubic-vs-S-Cubic (c.f. Fig. 13(b), right figure).

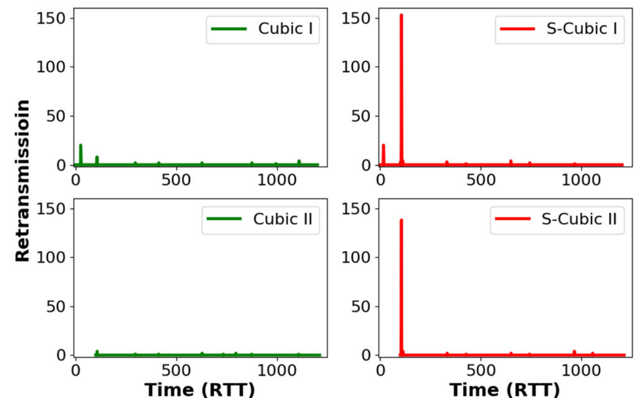


FIGURE 14. Number of retransmissions per RTT for two competing flows.

TABLE 4. Measured Mean RTT between the Servers (Google) and Clients (Tencent).

	S1: Belgium	S2: Oregon	S3: Tokyo
C1: California	135	18	108
C2: Frankfurt	8	142	229
C3: Hong Kong	264	131	50
C4: Moscow	44	178	270
C5: Mumbai	248	228	122
C6: Seoul	287	156	63
C7: Singapore	292	153	68
C8: Toronto	94	66	155

Not surprisingly, when the second flow joins 5 s after the first flow, congestion losses are induced, triggering both flows to enter congestion avoidance in order to adapt their CWnd to share bandwidth fairly. In comparison, S-Cubic induces more congestion losses than Cubic. For example, the second S-Cubic flow triggered 138 retransmissions versus Cubic's 4 retransmissions in the first RTT. The higher loss rate, however, only occurs in the first RTT though (i.e., during the Startup phase). After that, the retransmission rate is similar for both S-Cubic and Cubic, as both run the same Cubic congestion control algorithm.

E. INTERNET CLOUD EXPERIMENT

In this section, we turn our attention to experiments conducted in the open Internet. Specifically, we deployed S-Cubic to cloud instances around the world, with three

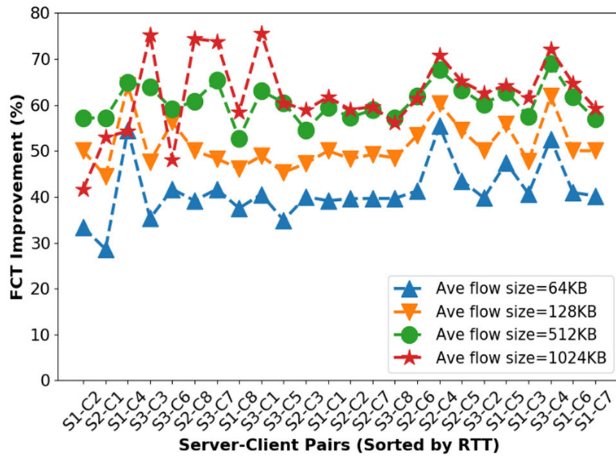


FIGURE 15. Average FCT improvement of S-Cubic over Cubic for the 24 client-server pairs.

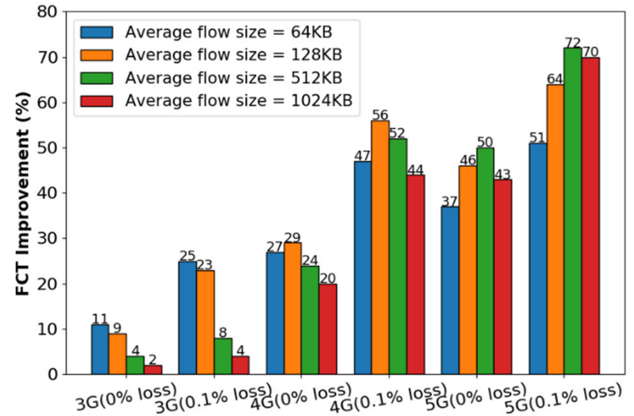


FIGURE 16. FCT improvements of S-Cubic over Cubic in 3G, 4G, and 5G mobile networks.

Google cloud [42] instances serving as servers and eight Tencent cloud [43] instances serving as clients. Table 4 summarizes the locations and the mean RTT measured between the 24 client-server location pairs, with client-server RTTs ranging from 8 ms (Frankfurt-to-Belgium) to 292 ms (Singapore-to-Belgium). The goal is to evaluate the performance of S-Cubic in real-world Internet environments.

For each client-server pair, we ran 1,000 flows for each flow size, alternating between Cubic and S-Cubic in an interleaved manner so that the network condition experienced by Cubic and S-Cubic is as close as practicable. As the link capacity is unknown in these experiments, FCT efficiency cannot be computed, so we employ S-Cubic's FCT improvement over Cubic as the performance metric. The results are summarized in Fig. 15 for the 24 client-server pairs across four average flow sizes. The x-axis is sorted in increasing order of client-to-server RTT (c.f. Table 4).

Generally speaking, the experimental results agreed with the previous results obtained from our test-bed setup (e.g., Fig. 10). In particular, S-Cubic achieves significant FCT improvements over Cubic in all test cases across all 24 client-server pairs, with an overall mean FCT improvement of 53.7%.

F. MOBILE NETWORK EXPERIMENTS

In this section, we investigate S-Cubic's performance in mobile networks via trace-driven emulated experiments. Mobile networks are known to exhibit rapid and significant bandwidth fluctuations over short timescales [18], [19], [44], so they may present a challenge to S-Cubic as it uses the estimated bandwidth from the previous flow as the initial sending rate.

To emulate a mobile network link, we capture actual bandwidth trace data from 3G, 4G, and 5G production mobile networks (by flooding it with UDP datagrams and then recording the reception rate at a stationary receiver), and then replicate them using a modified DummyNet [45] that can replay bandwidth trace data. Link buffer size was set to 1,280 KB and 5 MB in 3G and 4G networks, respectively [19], and RTT was set to 50 ms for both 3G/4G. For 5G, the RTT and link buffer size were set to 20 ms and 10 MB according to our measurement of the actual 5G network. Two random loss rates were tested: 0% and 0.1% [46]. Fig. 16 summarizes S-Cubic's FCT improvements over Cubic in mobile networks. There are two observations.

First, S-Cubic's performance gains increase when going from 3G to 4G, and increase further in 5G. This is due to differences in BDP – 3G's BDP is 22 MSS while they are 84 MSS and 305 MSS for 4G and 5G, respectively. This strongly suggests that in 5G and future high-speed mobile networks, TCP can become an increasingly significant bottleneck to fully realize the bandwidth offered by these high-speed networks. As further evidence, in a separate on-going measurement of a production mobile service by a tier-one service provider in China, the average TCP throughput for 1 MB data transfer is approximately 16 Mbps for customers connecting via 5G networks, which is unexpectedly low. However, our UDP-based measurement of a production 5G network in China shows an average bandwidth of 180 Mbps, indicating that the physical bandwidth is indeed very high, just that current TCP cannot fully utilize it.

Second, comparing the cases with and without random packet loss in Fig. 16, we see that S-Cubic's performance gains are higher when 0.1% random loss is introduced. This is somewhat counter-intuitive as except during the Startup phase, S-Cubic behaves in exactly the same way as regular Cubic. It turns out that the reason is precisely due to the difference in the Startup phase. In particular, random loss is known to disrupt Cubic's Slow-Start phase [3]–[5], thereby lengthening the time Cubic takes to ramp up its CWnd. By contrast, S-Cubic bypasses Slow-Start entirely, so it is far less impacted by random packet loss.

Next, we compare the delay performance of Cubic and S-Cubic in Table 5. The most significant differences are observed across different network types, with 3G exhibiting

TABLE 5. Comparison of Average Queuing Delay (in ms) between Cubic and S-Cubic in Mobile Networks.

Mobile Network	3G		4G		5G	
Random Loss	0%	0.1%	0%	0.1%	0%	0.1%
Cubic	194	47	30	6	3	1
S-Cubic	175	51	42	23	5	2

significantly longer delay than 4G, and in turn, 5G. This is a direct result of their bandwidth differences: ~5 Mbps for 3G, ~20 Mbps for 4G, and ~180 Mbps for 5G. On the other hand, random packet loss reduces the delay significantly in all three networks. This is because the congestion control algorithm in Cubic is sensitive to random loss [5], and thus its throughput is lowered by random loss, resulting in less queueing.

Finally, unlike in the case of the fixed-bandwidth network (c.f. Table 3), S-Cubic's queueing delay is higher than Cubic in almost all cases. This is due to mobile networks' bandwidth variations where queueing will occur whenever the bandwidth drops below the current transmission rate, even if the latter is lower than the actual mean bandwidth available. How to reduce queueing delay without sacrificing bandwidth efficiency in mobile networks is thus a challenging problem that warrants further investigation.

G. IMPACT OF AGED BANDWIDTH INFORMATION

A key piece of information to S-Cubic is the previous flow's estimated bandwidth to the same destination. Intuitively, the longer it takes for a new flow to start after the previous flow has terminated, the less correlated the estimated bandwidth will be when applied to the new flow. In the case of overestimation, the new flow will begin with a sending rate higher than the available bandwidth, potentially leading to congestion. On the other hand, underestimation will reduce the new flow's bandwidth utilization, resulting in longer FCT.

To investigate S-Cubic's sensitivity to bandwidth estimation errors, we conducted an experiment in a network with 100 Mbps bandwidth and 50 ms RTT, where a controlled percentage of bandwidth estimation error is artificially introduced to measure its impact on FCT and congestion. The experiment consists of 1,000 flows with an average flow size of 1024 KB following the Pareto distribution. In addition to Cubic and S-Cubic, we tested S-Cubic with Δ% of bandwidth estimation errors, ranging from −80% to +100%.

Fig. 17 plots the impact of bandwidth estimation error on FCT, queueing delay, and retransmission rate. For bandwidth underestimation, queuing delay is reduced while the FCT increases as expected. In the cases of bandwidth overestimation up to +60%, they increase the queuing delay due to the onset of congestion. Beyond that, retransmission kicks-in, rising to 2.3% at +100% overestimation.

Remarkably, FCT is relatively unaffected, e.g., 0.21 ms versus 0.19 ms at 100% versus 0% overestimation errors, respectively. The results show that although bandwidth estimation error does impact performance, its sensitivity and impact are relatively benign.

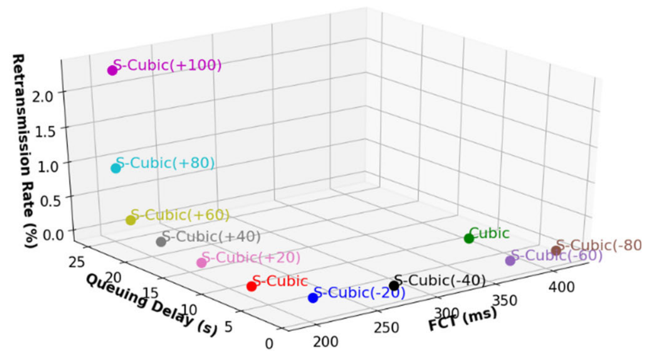


FIGURE 17. Impact of bandwidth estimation error on S-Cubic. Link bandwidth, RTT, and link buffer size were configured to 100 Mbps, 50 ms, and one BDP (625 KB).

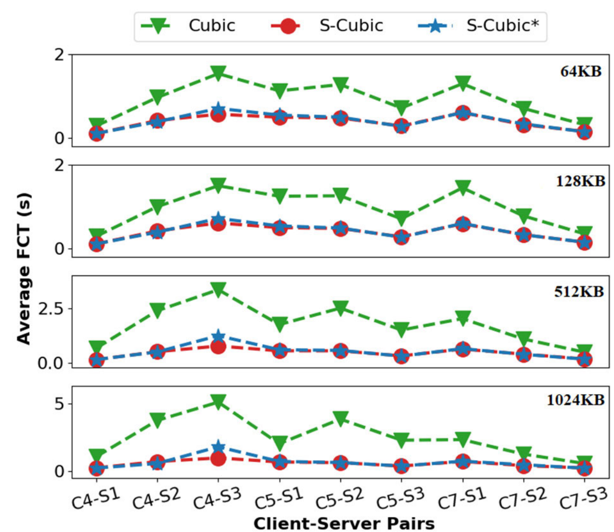


FIGURE 18. Comparison of FCTs for Cubic, S-Cubic, and S-Cubic* for nine client-server pairs. The experiment lasted for three days.

Taking the experiment one step further, we constructed a special experiment in the open Internet to assess the impact of real-world bandwidth estimation errors. Specifically, we created a special version of S-Cubic called S-Cubic*, where the estimated bandwidth is conducted only once at the beginning of the experiment (i.e., the first flow) and will not be updated by subsequent flows.

We conducted a new experiment using the same three Google servers and three Tencent clients - Moscow, Mumbai, and Singapore, covering paths with short to long RTTs. In each case, the client carried out a download job from the server once every twenty minutes. Each download job consists of three separate downloads of a fixed-size file: (i) download using Cubic; (ii) download using S-Cubic; and (iii) download using S-Cubic* using the fixed initial bandwidth estimate. The S-Cubic flow serves as a control,

producing the FCT where the bandwidth estimate is updated in each flow. The entire experiment lasted for three days.

The results are summarized in Fig. 18 for the nine client-server pairs. We observe that S-Cubic and S-Cubic* have similar FCTs across four flow sizes and client-server pairs. Except for C4-S3, the difference between S-Cubic and S-Cubic* is negligible even though all S-Cubic* flows in the three days were using the same single bandwidth estimate obtained at the beginning of the experiment. These results show that S-Cubic's sensitivity to the age of bandwidth estimate is relatively low even over a time horizon of three days in the network scenario tested.

H. INDEPENDENT BENCHMARKS

In this section, we report results provided by an independent performance benchmarking company Bonree [47] in China. This company specializes in conducting independent network performance benchmarks of which the results are primarily used by their customers to compare and select service providers for hosting Internet services.

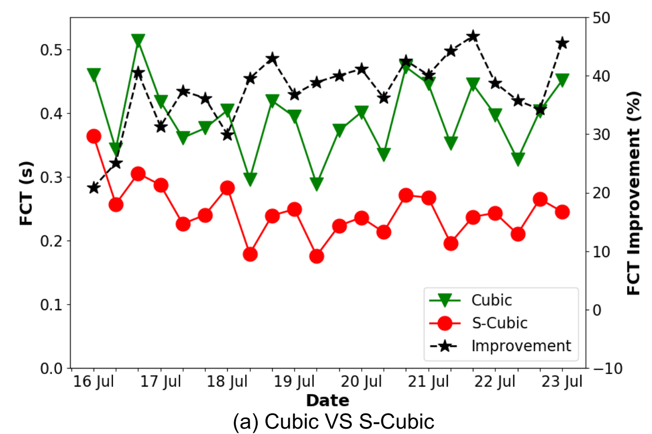
The benchmark setup (Table 6) comprises over 1,000 end-user clients spread across nine provinces in China. Clients are connected either via wired network or WiFi. Detail locations or network access information are not disclosed, presumably to prevent targeted optimization. The scale and geographical coverage of this benchmark offer a realistic look into the performance of Cubic and S-Cubic in real-world environments.

Fig. 19(a) summarizes the 8-hour mean FCT over the course of one week. We observe that S-Cubic consistently achieved substantially shorter mean FCT than Cubic. Overall, the average FCT for Cubic and S-Cubic was 0.40 s and 0.25 s, respectively, representing an FCT improvement of 37.5% by S-Cubic. Among the ~10,000 downloads, Cubic has very few cases with average throughput higher than 100 Mbps while in S-Cubic, around 10% downloads achieved throughput over 100 Mbps. Similarly, the overall average throughput achieved by Cubic and S-Cubic was 34.0 Mbps and 53.8 Mbps, representing a 58.2% gain in throughput by S-Cubic. Moreover, both Cubic and S-Cubic have similar download success rate of over 99.9%, demonstrating S-Cubic's compatibility with current network and client TCP implementation (Windows 10 in this case).

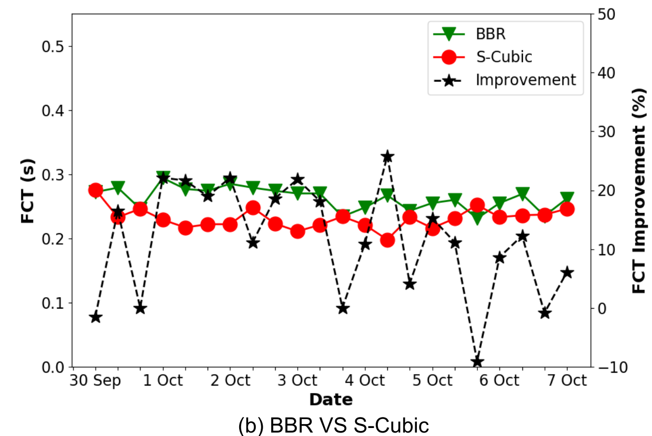
To offer a perspective on the performance of S-Cubic compared to more recent TCP designs, we conducted another set of experiments comparing the performance of BBR and S-Cubic in Bonree. Note that Bonree's platform supports comparison of a maximum of two TCP implementations at a time so Cubic was not included in this comparison. Also, the platform's Linux kernel version does not yet support BBRv2 so we tested BBRv1 instead. The results in Fig. 19(b) show that the performance gap between BBR and S-Cubic is much narrower, suggesting that BBR performed better than Cubic in this network environment. We conjecture that this is due to BBR's more robust performance in lossy networks (i.e., WiFi)

TABLE 6. Experiment Settings Adopted in the Independent Benchmarking platform Bonree [47].

Category	Setting	Value
Server	Number	2
	Locations	One in Guizhou province and one in Shandong province
	OS	Tencent Linux with Kernel 4.14
Client	Number	~1,000
	Locations	Distributed over nine cities over nine provinces in China
	OS	Windows 10
Network	Access Network	Wired or WiFi network
	ISP	China Mobile, China Unicom, and China Telecom.
Download	Flow size	1 MB
	Duration	One week
	Total downloads	~10,000



(a) Cubic VS S-Cubic



(b) BBR VS S-Cubic

FIGURE 19. Comparison of average FCT for Cubic, BBR and S-Cubic obtained from Bonree. Each data point is an average of 8-hour samples.

Overall, S-Cubic still managed to achieve 12% reduction in FCT compared to BBR. More importantly, as Stateful-TCP is designed to complement TCP's congestion control algorithm, this strongly suggests that applying Stateful-TCP to BBR will likely realize further performance gains.

VI. SUMMARY AND FUTURE WORK

This work introduces a new way to overcome the limitation of TCP Slow-Start, which is increasingly becoming the

bottleneck in modern high-speed networks. The emulation results, as well as experimental results obtained from real-world networks, show that the potential performance gains from applying Stateful-TCP to Cubic are not only substantial but could be essential to reap the benefits of emerging high-speed networks such as 5G and beyond.

This study is only the first step. There remain many open problems that warrant further investigation. For example, the application of Stateful-TCP to other TCP variants, especially those designed for different objectives, e.g., lower latency [8], [10], very large BDP [6], would be a fruitful area for future research. On the other hand, Stateful-TCP currently only accelerates the startup phase. Some of the techniques in Stateful-TCP, such as GCBE, could potentially be applied to congestion control as well so that performance beyond the startup phase could be improved further.

Finally, in spite of the positive results obtained in this study, broader experimentation and validation of Stateful-TCP in general, and S-Cubic in particular, is needed to better understand its performance in a broader range of environments and its potential impact to other traffics sharing the same bottleneck. The authors are collaborating with Tencent in this regard to scale-up the Internet experiments and also to expand it into real-world Internet services to measure the performance gains at the application level. In addition, we are releasing the S-Cubic implementation in Linux as open-source software[1] to enable the research community to scrutinize the implementation, to validate its performance independently, and to use it as a template for applying Stateful-TCP to other TCP designs.

ACKNOWLEDGMENT

The authors wish to thank the Associate Editor and the anonymous reviewers for their suggestions in improving this article to its final form, and Tencent for their generous support of this work by donating Tencent cloud instances for the experiments and providing access to the Bonree benchmarking platform.

REFERENCES

[1] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 3, pp. 26–33, 2010.

[2] I. Rhee and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[3] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proc. 7th Annu. Int. Conf. Mobile Comput. Netw. (MobiCom)*, Rome, Italy, 2001, pp. 287–297.

[4] C. P. Fu and S. C. Liew, "TCP veno: TCP enhancement for transmission over wireless access networks," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 2, pp. 216–228, Feb. 2003.

[5] N. Cardwell, Y. Cheng, C. S. Gunn, V. Jacobson, and S. Yeganeh, "BBR: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, Sep. 2016.

[6] C. Caini and R. Firrincieli, "TCP hybla: A TCP enhancement for heterogeneous networks," *Int. J. Satell. Commun. Netw.*, vol. 22, no. 5, pp. 547–566, Sep. 2004.

[1] S-Cubic is available at https://github.com/mclab-cuhk/Stateful-TCP

[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, New Delhi, India, Aug. 2010, pp. 63–74.

[8] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Proc. NSDI*, Lombard, IL, USA, Apr. 2013, pp. 459–471.

[9] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *Proc. NSDI*, Oakland, CA, USA, May 2015, pp. 395–408.

[10] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the Internet," in *Proc. NSDI*, Renton, WA, USA, 2018, pp. 329–342.

[11] S. Hauger, M. Scharf, J. Kogel, and C. Suriyajan, "Quick-start and XCP on a network processor: Implementation issues and performance evaluation," in *Proc. Int. Conf. High Perform. Switching Routing*, Shanghai, China, May 2008, pp. 703–714.

[12] D. Liu, M. Allman, S. Jiny, and L. Wang, "Congestion control without a startup phase," in *Proc. Int. Workshop PFLDnet*, Feb. 2007, pp. 61–66.

[13] Q. Li, M. Dong, and P. B. Godfrey, "Halfback: Running short flows quickly and safely," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Berlin, Germany, 2015, pp. 1–13.

[14] X. Nie, Y. Zhao, G. Chen, K. Sui, Y. Chen, D. Pei, M. Zhang, and J. Zhang, "TCP WISE: One initial congestion window is not enough," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, San Diego, CA, USA, Dec. 2017, pp. 1–8.

[15] K. Winstein and H. Balakrishnan, "TCP ex machina: Computer-generated congestion control," in *Proc. SIGCOMM*, HongKong, Aug. 2013, pp. 123–134.

[16] S. Ha and I. Rhee, "Taming the elephants: New TCP slow start," *Comput. Netw.*, vol. 55, no. 9, pp. 2092–2110, Jun. 2011.

[17] J. Touch *TCP Control Block Interdependence*, document RFC 2140, Apr. 1997.

[18] K. Liu and J. Y. B. Lee, "Mobile accelerator: A new approach to improve TCP performance in mobile data networks," in *Proc. 7th Int. Wireless Commun. Mobile Comput. Conf.*, Istanbul, Turkey, Jul. 2011, pp. 2174–2180.

[19] K. Liu and J. Y. B. Lee, "On improving TCP performance over mobile data networks," *IEEE Trans. Mobile Comput.*, vol. 15, no. 10, pp. 2522–2536, Oct. 2016.

[20] J. Zhou, Z. Li, Q. Wu, P. Steenkiste, S. Uhlig, J. Li, and G. Xie, "TCP stalls at the server side: Measurement and mitigation," *IEEE/ACM Trans. Netw.*, vol. 27, no. 1, pp. 272–287, Feb. 2019.

[21] M. Allman, V. Paxson, and E. Blanton, *TCP Congestion Control*, document RFC 5681, Sep. 2009.

[22] S. Floyd, T. Henderson, and A. Gurtov, *The NewReno Modification to TCP's Fast Recovery Algorithm*, document RFC 3782. Apr. 2004.

[23] S. Floyd, *Limited Slow-Start for TCP With Large Congestion Windows*, document RFC 3742, Mar. 2004.

[24] D. Cavendish, K. Kumazoe, M. Tsuru, Y. Oie, and M. Gerla, "CapStart: An adaptive TCP slow start for high speed networks," in *Proc. 1st Int. Conf. Evolving Internet*, Aug. 2009, pp. 15–20.

[25] Huang. *Hystart++: Modified Slow Start for TCP*. Accessed: Sep. 1, 2020. [Online]. Available: https://tools.ietf.org/id/draft-ietf-tcpm-hystartplusplus-00.html

[26] *DummyNet*. Accessed: Sep. 1, 2020. [Online]. Available: https://cs.baylor.edu/~donahoo/tools/dummy

[27] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, and J. Bailey, "The QUIC transport protocol: Design and Internet-scale deployment," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Los Angeles, CA, USA, Aug. 2017, pp. 183–196.

[28] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger, "TCP revisited: A fresh look at TCP in the wild," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf. (IMC)*, Chicago, IL, USA, 2009, pp. 76–89.

[29] Y. Cheug, J. Chu, S. Radhakrishnan, and A. Jain, *TCP Fast Open*, document RFC 7413, Dec. 2014.

[30] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, *TCP Selective Acknowledgment Options*, document RFC 2018, Oct. 1996.

[31] H. Balakrishnan, V. N. Padmanabhan, G. Fairhurst, and M. Sooriyabandara, *TCP Performance Implications of Network Path Asymmetry*, dcoument RFC 3449, Dec. 2002.

[32] T. Zhang, J. Wang, J. Huang, J. Chen, Y. Pan, and G. Min, "Tuning the aggressive TCP behavior for highly concurrent HTTP connections in intra-datacenter," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3808–3822, Dec. 2017.
[33] L. Qiu, Y. Zhang, and S. Keshav, "On individual and aggregate TCP performance," in *Proc. 7th Int. Conf. Netw. Protocols*, Oct. 1999, pp. 203–212.
[34] J. Rojas-Mora, T. Jimenez, and E. Altman, "Simulating flow level bandwidth sharing with Pareto distributed file sizes," in *Proc. 5th Int. ICST Conf. Perform. Eval. Methodologies Tools*, Brussels, Belgium, 2011, pp. 265–273.
[35] E. Chlebus and R. Ohri, "Estimating parameters of the Pareto distribution by means of Zipf's law: Application to Internet research," in *Proc. GLOBECOM. IEEE Global Telecommun. Conf.*, Nov. 2005, p. 5.
[36] P. Olivier, "Internet data flow characterization and bandwidth sharing modelling," in *Proc. Int. Teletraffic Congr.*, in (Lecture Notes in Computer Science), 2007, pp. 986–997.
[37] S. Floyd and V. Paxson, "Difficulties in simulating the Internet," *IEEE/ACM Trans. Netw.*, vol. 9, no. 4, pp. 392–403, Aug. 2001.
[38] K. Avrachenkovt, U. Ayesta, P. Brown, and E. Nyberg, "Differentiation between short and long TCP flows: Predictability of the response time," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 762–773.
[39] A. B. Downey, "Lognormal and Pareto distributions in the Internet," *Comput. Commun.*, vol. 28, no. 7, pp. 790–801, May 2005.
[40] A. Hassidim, D. Raz, M. Segalov, and A. Shaqed, "Network utilization: The flow view," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1429–1437.
[41] A. Odlyzko, "Data networks are lightly utilized, and will stay that way," *Rev. Netw. Econ.*, vol. 2, no. 3, pp. 210–237, Jan. 2003.
[42] *Google Cloud*. Accessed: Sep. 1, 2020. [Online]. Available: https://cloud.google.com
[43] *Tencent Cloud*. Accessed: Sep. 1, 2020. [Online]. Available: https://intl.cloud.tencent.com
[44] F. Ahmed, J. Erman, Z. Ge, A. X. Liu, J. Wang, and H. Yan, "Detecting and localizing end-to-end performance degradation for cellular data services," in *Proc. IEEE INFOCOM - 35th Annu. IEEE Int. Conf. Comput. Commun.*, Portland, OR, USA, Apr. 2016, pp. 459–460.
[45] *Modified DummyNet*. Accessed: Sep. 1, 2020. [Online]. Available: https://github.com/mclab-cuhk/netmap-ipfw
[46] Y.-C. Chen, E. M. Nahum, R. J. Gibbens, and D. Towsley, "Measuring cellular networks: Characterizing 3G, 4G, and path diversity," in *Proc. Annu. Conf. Int. Technol. Alliance*, Jun. 2012, pp. 1–8.
[47] *Bonree*. Accessed: Sep. 1, 2020. [Online]. Available: https://www.bonree.com
[48] G. C. Kessler. *An Overview of Cryptography*. Accessed: Sep. 1, 2020. [Online]. Available: http://www.garykessler.net/library/crypto.html

LINGFENG GUO received the B.Eng. degree in software engineering from Sun Yat-sen University, Guangdong, China, in 2016. He is currently pursuing the Ph.D. degree with the Department of Information Engineering, The Chinese University of Hong Kong. He has participated in research and development of internet protocols with The Chinese University of Hong Kong.

JACK Y. B. LEE (Senior Member, IEEE) received the B.Eng. and Ph.D. degrees in information engineering from The Chinese University of Hong Kong, Hong Kong, in 1993 and 1997, respectively. He is currently an Associate Professor with the Department of Information Engineering, The Chinese University of Hong Kong. He specializes in tackling research challenges arising from real-world systems. He also works with the industry to uncover new research challenges and opportunities for new services and applications. Several of the systems research from his laboratory has been adopted and deployed by the industry. His research interests include research in multimedia communications systems and mobile communications, protocols, and applications.