SUSS: Improving TCP Performance by Speeding Up Slow-Start

Mahdi Arghavani
University of Otago
Otago, New Zealand
argma678@student.otago.ac.nz

Haibo Zhang
University of Otago
Otago, New Zealand
haibo.zhang@otago.ac.nz

David Eyers
University of Otago
Otago, New Zealand
david.eyers@otago.ac.nz

Abbas Arghavani
Mälardalen University
Västerås, Sweden
abbas.arghavani@mdu.se

ABSTRACT

The traditional slow-start mechanism in TCP can result in slow ramping-up of the data delivery rate, inefficient bandwidth utilization, and prolonged completion time for small-size flows, especially in networks with a large bandwidth-delay product (*BDP*). Existing solutions either only work in specific situations, or require network assistance, making them challenging (if even possible) to deploy. This paper presents SUSS (Speeding Up Slow Start): a lightweight, sender-side add-on to the traditional slow-start mechanism, that aims to safely expedite the growth of the congestion window when a flow is significantly below its optimal fair share of the available bandwidth. SUSS achieves this by accelerating the growth in *cwnd* when exponential growth is predicted to continue in the next round. SUSS employs a novel combination of ACK clocking and packet pacing to effectively mitigate traffic burstiness caused by accelerated increases in *cwnd*. We have implemented SUSS in the Linux kernel, integrated into the CUBIC congestion control algorithm. Our real-world experiments span many device types and Internet locations, demonstrating that SUSS consistently outperforms traditional slow-start with no measured negative impacts. SUSS achieves over 20% improvement in flow completion time in all experiments with flow sizes less than 5 MB and *RTT* larger than 50 ms.

CCS CONCEPTS

• **Networks** → **Transport protocols**.

KEYWORDS

Congestion Control, slow-start, TCP optimization, CUBIC

ACM Reference Format:
Mahdi Arghavani, Haibo Zhang, David Eyers, and Abbas Arghavani. 2024. SUSS: Improving TCP Performance by Speeding Up Slow-Start. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3651890.3672234

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672234

1 INTRODUCTION

To ensure high but sustainable network utilization, TCP uses a congestion control algorithm (CCA) to regulate the number of inflight packets (those sent but not yet acknowledged) per TCP connection by dynamically adjusting the size of its congestion window (*cwnd*). To seek out the optimal *cwnd*, denoted *cwnd**, each TCP connection starts with a slow-start phase, in which the volume of inflight data per round-trip time (*RTT*) is doubled until congestion is detected. This conservative approach helps prevent severe congestion during the initial phase of a TCP connection, but it also results in the underutilization of network bandwidth, leading to *increased flow completion time (FCT)* [28]. Extended FCT can significantly diminish the Quality of Experience (QoE) [10], especially for flows less than a few megabytes in size, which we refer to as 'small TCP flows' in this paper. These small flows constitute a substantial portion of today's TCP traffic on the Internet [19], driven by the widespread usage of web browsing and popular social media platforms such as Facebook, WhatsApp, and TikTok.

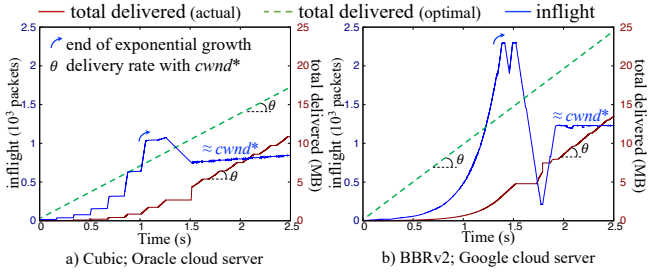


Figure 1: File download measurements from a US cloud server to a PC in New Zealand using two different CCAs.

Fig. 1 shows our measurements for downloading a file from a cloud server in the United States to a PC in New Zealand using two CCAs (Cubic [15, 33] and BBRv2 [7]). θ represents the delivery rate achieved with *cwnd**, calculated as the ratio of *cwnd** to *RTT*. To demonstrate the inefficiency of slow-start, we used the value of *cwnd* during the steady-state phase as an estimate for *cwnd** to calculate θ. Then, we added a dashed green line to each plot, hypothesizing a scenario where the data rate is optimal from the outset. This visualization highlights the capability for transferring a larger amount of data during the initial phase, which includes the

majority of small TCP flows for delivering small data objects such as web pages, photos, and short videos.

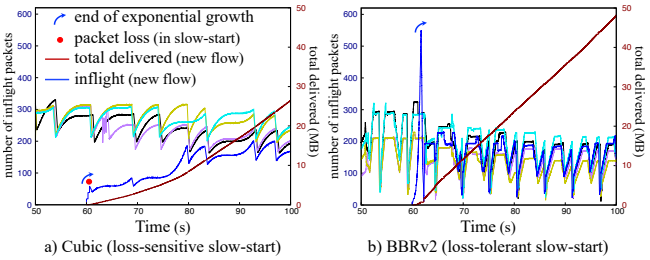


Figure 2: Local testbed measurements: A new TCP flow competes for bottleneck bandwidth against four existing flows.

Fig. 2 depicts another measurement for downloading a file in our local testbed, which consists of five pairs of client-server machines interconnected via two Linux routers in a dumbbell topology. This example highlights the inefficiency of CUBIC's slow-start when operating within a congested network path. As shown in Fig. 2(a), a new TCP flow faces challenges in rapidly attaining a fair share of network resources when competing with established larger flows, resulting in prolonged unfairness. This issue is due to CUBIC's sensitivity to packet loss, potentially causing a new flow to exit the slow-start phase prematurely, before reaching its $cwnd^*$. In contrast, as shown in Fig. 2(b), BBR demonstrates greater loss tolerance, facilitating the new TCP flow's achievement of its $cwnd^*$ after experiencing a delay caused by the slow-start under-utilization—a factor that is negligible for large TCP flows.

Despite the extensive study of CCAs, bandwidth under-utilization during slow-start has not been given much consideration. Several network-assisted approaches [4, 9, 10, 16, 30, 31] have been proposed to improve data delivery rate during slow-start. However, these approaches require feedback from routers and switches, making them challenging for deployment and testing. End-to-end solutions [1, 22, 34] have also been designed to estimate $cwnd^*$ based on bandwidth measurement and optimal rate estimation. However, it is hard to achieve accurate estimations in early *RTT*s due to the lack of sufficient and reliable information about the network path.

In this paper, we present SUSS, a lightweight, sender-side add-on to CUBIC's slow-start mechanism. It aims to safely accelerate *cwnd* growth in large-BDP networks and enhance the delivery rate during the early *RTT*s of slow-start. The **key idea** of SUSS is to predict the continuation of exponential growth of *cwnd* in the subsequent *RTT*, based on a well-established slow-start exit mechanism. Specifically, if *cwnd* is extrapolated to continue to increase in the next *RTT*, the growth of *cwnd* in the current *RTT* can be proportionally accelerated. This allows *cwnd* to quickly ramp up in the early *RTT*s of slow-start when its current value is still far below $cwnd^*$. To effectively implement SUSS, **two primary challenges** must be addressed: firstly, determining the method by which SUSS predicts the likelihood of exponential *cwnd* growth in the next round, and secondly, devising a method for the controlled transmission of additional data packets to prevent negative consequences such as short-term bursts of packets and packet loss.

To address the first challenge, we develop a theoretical framework to estimate whether *cwnd* growth continues in the next round, based on HyStart [14]—the slow-start exit mechanism used in CUBIC. Our approach leverages measurements obtained from ACKs received in the current *RTT*, with a particular emphasis on the minimum *RTT* and the time interval between the first and the last ACKs received in the current *RTT*. These observations are critical for assessing the conditions of the network path. To address the second challenge we propose a novel combination of ACK clocking and packet pacing. ACK clocking is used to protect the functionality of HyStart and its estimation of the growth factor, whereas packet pacing is used to avoid the bursts of traffic that might be caused by surges in *cwnd*.

We have implemented SUSS within version 5.19.10 of the Linux kernel. The source code and an installation guide are available on GitHub at https://github.com/SUSSdeveloper/SUSS. We conducted extensive experiments for performance evaluation by deploying SUSS on various cloud and on-premises servers and testing it with different communication links for the last hop (e.g. wired, Wi-Fi, 4G/5G). Experimental results demonstrate that SUSS consistently outperforms slow-start with no negative impact on crucial aspects of TCP performance such as fairness.

The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 introduces the theory behind SUSS. We present the design of SUSS in Section 4 and its implementation in Section 5. Experimental results are discussed in Section 6, and Section 7 concludes the paper and suggests future work.

2 RELATED WORK

Existing studies on addressing bandwidth underestimation in slow-start can be categorised into two groups: network-assisted approaches and end-to-end approaches.

Network-assisted approaches rely on explicit and timely feedback from network devices such as routers and switches to improve delivery rate in slow-start. For example, Quick-Start [30] includes the desired initial sending rate in the SYN packet during TCP handshake, and routers can further reduce this rate to a value acceptable to them. By utilizing markings on packets to obtain the longest instantaneous queue length, the scheme presented in [4] increases the flow rate up to the available capacity, instead of relying on slow-start for exploration. In RCP [9, 10], each router assigns a single fair-share rate for all passing flows, and each flow is sent at the lowest rate offered by the routers along the path. Additionally, there are several Active Queue Management (AQM) algorithms, such as [16], designed to manage queue delays in routers while also helping TCP slow-start converge to $cwnd^*$ more quickly and efficiently. By leveraging the capabilities of switches with P4-programmable data planes, P4air [31] enables each TCP connection to start with a fair share of the available bandwidth to enforce fairness among flows. Since network-assisted approaches require knowledge of the network conditions to provide feedback and control, they are typically complex and not scalable, making deployment and testing difficult, if not entirely impractical.

End-to-end approaches, which modify either the sender, receiver or both, offer significant flexibility and scalability for real-world deployment. Google's BBR [6] is a sender-side approach that

continuously explores available bandwidth based on the increase in delivery rate instead of using packet loss as a congestion signal. By doing so, BBR enhances the delivery rate during slow-start and avoids drastic reductions caused by occasional packet loss. However, BBR does not address bandwidth under-utilization, as it retains the growth dynamics of traditional slow-start (see Fig. 1.b).

It is intuitive to use a larger initial window size (*iw*) to overcome bandwidth under-utilization and enhance the delivery rate from the outset. RFC3390 [27] increased *iw* from 1–2 segments to around 4KB (2–4 segments). In 2013, RFC6928 [8] raised *iw* to 10 segments (approximately 15KB), which is still in use today due to the need to fit all networks and applications. Typically, TCP is unaware of the flow length when it begins transmitting data packets and lacks prior knowledge of the network path conditions to appropriately select the *iw* size. Some studies have encouraged enlarging *iw* while many researchers are concerned about the consequences of releasing a large burst of data packets in early rounds [28, 29]. Initial spreading [29] was proposed to add space between packets to reduce packet loss during *iw* enlargement.

PCP [1], AFStart [34], JumpStart [22], and Paced Chirping [24] rely on measuring the bandwidth of the bottleneck link or testing the feasibility of a sending rate. PCP [1] sends a short sequence of probe packets at a specific rate to detect whether the current network environment is capable of supporting this testing rate. In JumpStart [22], the sending rate of the initial round is calculated as the minimum value between the receiver's advertised window and the amount of locally queued data, divided by the *RTT* obtained from the three-way handshake. It terminates slow-start when either all locally queued data is transmitted or an ACK arrives. Similar to JumpStart, Halfback [21] begins with an aggressive initial packet pacing phase. However, in the subsequent *RTT*, it proactively secures itself against potential packet losses. Although its loss recovery strategy enables Halfback to swiftly recover from loss events, it has to re-transmit nearly 50% of the packets during the slow-start phase, which will introduce considerable overhead [20].

On the other hand, Stateful-TCP [12, 13], TCP WISE [25], and TCP-RL [26] rely on historical experience to estimate the initial rate. Stateful-TCP leverages the sending rate of previous flows with the same destination, which is beneficial when a user sends consecutive requests to a server. TCP WISE dynamically sets *iw* for different user clusters. TCP-RL uses reinforcement learning to choose a large enough *iw* for connections grouped based on ISP or region.

Most of the above proposals assume uncontrolled exponential growth in *cwnd* without any proactive slowdown, which can disrupt the functions designed to terminate the exponential growth in *cwnd* such as HyStart [14]. Moreover, accurate bandwidth measurement and rate estimation are very challenging, especially in early rounds, due to the lack of sufficient and reliable information on the network path condition. Over-estimation of either bandwidth or delivery rate can cause packet loss and result in the early exit of slow-start.

3 THEORY BEHIND SUSS

In this section, we present the core concept of SUSS. To this end, as shown in Fig. 3, we consider a TCP connection being established to transfer a volume of data from a sender to a receiver along a set of links and routers, where one of the links has the minimum

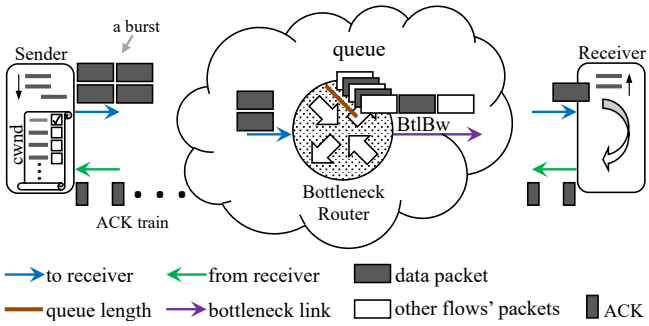


Figure 3: TCP abstract system model.

capacity (a.k.a., the bottleneck link). The key parameters of our abstract model include:

- $round(i)$: the time interval $[t_i^s, t_{i+1}^s)$, where t_i^s is the time when the ACK of the first data packet sent in $round(i-1)$ is received for $i > 1$. For $i = 1$, $round(1)$ initiates at the beginning of the data transfer.
- $cwnd_i$: maximum congestion window size in $round(i)$.
- *iw*: volume of data sent in $round(1)$, i.e., $cwnd_1 = iw$.
- *Data train*: sequence of data packets sent in a round.
- *ACK train*: sequence of ACKs for a data train.
- Δt_i^{at}: time taken to receive the ACK train in $round(i)$.
- R_i: the transmission rate of the data train in $round(i)$.
- *minRTT*: measured minimum round-trip time since the initiation of the TCP connection.
- $moRTT_i$: the minimum observed *RTT* in $round(i)$.
- *BtlBw*: bandwidth of the bottleneck link.
- G_i: growth factor for *cwnd* in $round(i)$.
- $cwnd^*$: the optimal congestion window size when a fair share of *BtlBw* is achieved.

Generally, in the slow-start phase $cwnd_i$ is updated in each round i as below:

$$cwnd_i = G_i \times cwnd_{i-1} \tag{1}$$

where G_i is always 2 in the traditional slow-start. This leads to a conservative increment in *cwnd*, which typically results in substantial bandwidth under-utilization at the phase's onset, when *cwnd* is far below $cwnd^*$. The primary aim of SUSS is to expedite the exponential increase of *cwnd* when it is believed to be significantly less than $cwnd^*$. To achieve this, SUSS recommends that in the current round i, the sender adheres to the slow-start protocol first. After fully receiving the ACK train, the sender evaluates, based on information gained from the current round, whether the exponential growth of *cwnd* is extrapolated to persist for the next round. If this is the case, the sender does not delay until the onset of the next round (i.e., t_{i+1}^s) and continues sending an additional $2 \times cwnd_{i-1}$ of data at the same rate R_i (see Fig. 4). This method effectively increases the growth factor of $cwnd_i$ to $G_i = 4$, facilitating a more rapid expansion of $cwnd_i$ when it is far below its optimal level. A question may arise as to why SUSS does not factor in potential *cwnd* growth in later rounds instead of just the next round to further amplify *cwnd* expansion, especially if an increase is foreseen in future rounds. While it is feasible to extend SUSS

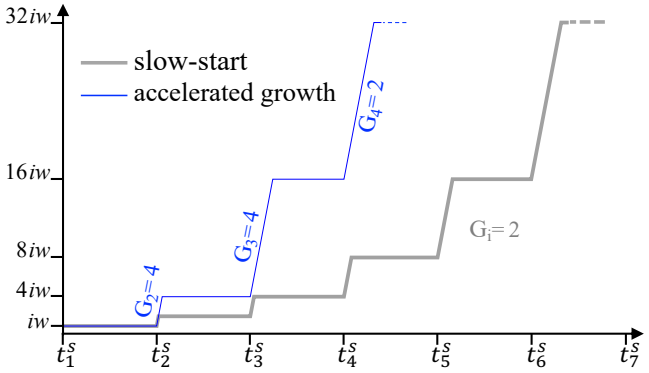


Figure 4: *cwnd* evolution in slow-start ($G_i = 2$) versus an example of accelerated growth.

to consider *cwnd* growth beyond the next round, our experimental analysis indicates that the network condition may vary over several upcoming rounds, potentially leading to the risk of rapid network congestion. Nonetheless, Appendix A outlines the theory for situations where stable network conditions are maintained over multiple rounds.

The **key challenge** for SUSS is to determine if the exponential growth of *cwnd* is expected to continue in the next round. To address this, SUSS relies on criteria designed for stopping *cwnd*'s exponential growth. Among the several criteria suggested for preventing *cwnd* overshoot during the slow-start [2, 3, 14], we have chosen to implement the approach from HyStart [14]. This method is noteworthy because it is part of CUBIC, the default CCA used in the network protocol stacks of Linux, Windows, and macOS. In HyStart, the exponential growth of *cwnd* is allowed if the following conditions are fulfilled: (1) *Upon receiving an ACK, the amount of time elapsed from the start of the current round does not exceed half of the minRTT*; (2) *The minimum observed RTT in the current round ($moRTT_i$) must not be greater than* $1.125 \times minRTT$. The thresholds used by HyStart, which are also incorporated into the Linux implementation of CUBIC, are based on empirical observations and rational considerations, ensuring efficient and stable network performance. Notably, while SUSS uses these thresholds, it is not strictly bound to them and can function effectively even if the thresholds are adjusted. In what follows we detail how SUSS formulates and assesses these conditions.

Formulating Condition 1 for SUSS: To predict whether Condition 1 is met in the next round, all SUSS has to do is investigate whether the following inequality holds:

$$\Delta t_{i+1}^{at} \leq \frac{minRTT}{2}. \tag{2}$$

Since Δt_{i+1}^{at} is not known in $round(i)$, it necessitates an estimation. At time t_i^s, $round(i > 1)$ begins upon receiving an ACK for the first packet sent in the preceding round. Subsequently, upon receiving every ACK, SUSS suggests transmitting twice the amount of acknowledged data, exactly similar to the slow-start. Hence, the ACK train is completely received by the time $t_i^s + \Delta t_i^{at}$ while $2 \times cwnd_{i-1}$ of data have been sent during Δt_i^{at} seconds at rate $R_i = (2 \times cwnd_{i-1})/\Delta t_i^{at}$. At this time (i.e., at $t_i^s + \Delta t_i^{at}$), the sender has enough information to reliably predict whether exponential growth will continue in the next round. When the sent data train in the current $round(i)$ (with the size $2 \times cwnd_{i-1}$) enters the bottleneck buffer, it takes $\frac{2 \times cwnd_{i-1}}{BtlBw}$ seconds for the router to send out the data train. In other words, the tail of the data train follows its head after $\frac{2 \times cwnd_{i-1}}{BtlBw}$ seconds. This means that the corresponding ACK train will be received in $round(i+1)$ during

$$\Delta t_{i+1}^{at} \approx \frac{2 \times cwnd_{i-1}}{BtlBw} \tag{3}$$

seconds. Applying the same logic to the data train dispatched in $round(i-1)$ yields:

$$\Delta t_i^{at} \approx \frac{cwnd_{i-1}}{BtlBw}. \tag{4}$$

Consequently, it can be inferred that:

$$\Delta t_{i+1}^{at} \approx 2 \times \Delta t_i^{at}. \tag{5}$$

This implies that Eq. (2) is satisfied, and exponential growth persists into $round(i+1)$ if:

$$\Delta t_i^{at} \leq \frac{minRTT}{4}. \tag{6}$$

Formulating Condition 2 for SUSS: The next step involves the sender determining if *moRTT* for the next round ($moRTT_{i+1}$) will be less than $1.125 \times minRTT$. In CUBIC HyStart, $1.125 \times minRTT$ acts as a threshold to identify initial signs of queueing delay, signaling the onset of network congestion. Direct calculation of $moRTT_{i+1}$ in $round(i)$ is not feasible; hence, we resort to estimation. Let $moRTT_i$ represent the current round's minimum observed *RTT*. Given that the *minRTT* was last updated r rounds ago, we infer that the average increase in queuing delay since then is approximately $(moRTT_i - minRTT)/r$ seconds per round.[1] Consequently, $moRTT_{i+1}$ can be estimated as follows:

$$moRTT_{i+1} = moRTT_i + \frac{(moRTT_i - minRTT)}{r}. \tag{7}$$

Therefore, to validate Condition 2, the sender should verify the following inequality:

$$moRTT_i + \frac{(moRTT_i - minRTT)}{r} \leq 1.125 \times minRTT. \tag{8}$$

We discuss three of the potential concerns that the method proposed for accelerating *cwnd* may raise:

(1) Intensifying the traffic burstiness: A rapid surge in *cwnd* can result in a short-term burst in data rate, potentially leading to adverse effects on TCP performance. To prevent further extending a high density data train, we will introduce an innovative technique in Section 4 to effectively mitigate the exacerbation of burstiness that employs a combination of packet pacing and ACK clocking. Our proposed technique can ensure a more controlled and balanced transmission pattern while maintaining the network path conditions that HyStart depends on.

(2) Impact on other concurrent flows: Another concern may arise regarding the potential impact of using a larger G on other flows sharing the same bottleneck link. While it is true that transmitting data at a higher rate can lead to congestion and fairness-related

[1] Since quadrupling *cwnd* happens in the early *RTT*s, r is typically small, 1 or 2.

issues, our experimental evaluations in Section 6 confirm that the proposed idea does not impact fairness in the numerous scenarios we explore, and even improves fairness when a new flow is initiated in a congested environment, similar to the example shown in Fig. 2. This is due to the following reasons:

- A larger G is used only when *cwnd* is still significantly smaller than its fair share of the network path capacity. As a result, the acceleration of *cwnd* growth is likely to remain within the flow's fair share and will not cause unfairness.
- The utilization of a larger G is transient, primarily limited to the early rounds when the increase in *cwnd* is small compared to its *BDP* and the volume of cross traffic.
- The additional packets sent due to the acceleration of *cwnd* growth are paced to minimize the chance of queuing pressure and spikes. In other words, additional data packets are spread throughout the network path.
- An earlier increase in *cwnd* helps the new flow to obtain a higher share of *BtlBw* in scenarios where an early packet loss prematurely ends the slow-start phase.

(3) Impact of variation in *BtlBw*: According to Eq. 5 and Eq. 7, Δt_{i+1}^{at} and $moRTT_{i+1}$ are estimated based on the measurements in $round(i)$. Hence, the accuracy of these estimations can suffer from the variations in *BtlBw*. Clearly, any increase in *BtlBw* will not have a negative impact on SUSS. However, if *BtlBw* drops, both Δt_{i+1}^{at} and $moRTT_{i+1}$ will be under-estimated. If *BtlBw* drops when *cwnd* is far below $cwnd^*$, the bottleneck link is still far from saturation. In such a case, quadrupling the *cwnd* is less likely to overflow the bottleneck buffer, as the potential side effect may be mitigated by the buffer's capacity. If *BtlBw* drops when *cwnd* is close to $cwnd^*$, it is less likely that SUSS will apply a growth factor of 4 because instability in *BtlBw* enlarges the length of the ACK train and queuing delay so that both conditions cannot be satisfied. In such a case, even if *BtlBw* drops, SUSS will perform in the same way as traditional slow-start since both double *cwnd*. A more detailed analysis of the impact of *BtlBw* variation is given in Appendix B.

4 DESIGN OF SPEED-UP SLOW-START (SUSS)

Here we describe the design of SUSS, our proposed add-on to TCP slow-start to safely accelerate *cwnd* growth, based on the ideas and work presented in the previous section.

ACK clocking is a widely employed mechanism in TCP to smooth the transmission of traffic by dynamically adjusting the sending rate of data packets in response to the receipt of ACK packets. Packet pacing is another widely adopted mechanism in TCP to mitigate burstiness by regulating the timing and spacing between consecutive data packets. Neither clocking nor pacing is adequate on its own to effectively tackle the challenges posed by applying a large growth factor (G_i). When relying solely on ACK clocking, a large G_i extends the transmission of a burst of packets upon receiving each ACK train. By exclusively using packet pacing, the measurement of Δt_i^{at} will no longer accurately reflect the network path condition in the subsequent rounds, thereby disrupting the operation of the HyStart mechanism and hampering the estimation of G_i.

To address the aforementioned challenges, SUSS employs a novel combination of ACK clocking and packet pacing to send the data train in each round when $G_i > 2$, as illustrated in Fig. 5. In the *clocking period*, upon receiving an ACK, SUSS always sends twice the amount of data acknowledged by that ACK. The purpose of this setup is to facilitate the measurement of Δt_i^{at}, which is crucial for HyStart functionality and the estimation of G_i. Note that reducing the amount of data sent during the clocking period would not accurately mimic the traditional slow-start, resulting in less precise measurements of Δt_i^{at}. When $G_i > 2$, the *clocking period* cannot send the whole data train in the current round. There will be a pacing period in which the remaining data in the data train will be sent at a predetermined pace to mitigate burstiness. There is a *guard interval* before and after the pacing period to avoid interference with the clocking period in both the current and the next round. It is worth noting that, when the measured growth factor is no greater than 2, SUSS operates similarly to the traditional slow-start and only uses the clocking mode to send the data train in the current round. In the following, we present the detailed design for the two periods when $G_i > 2$.

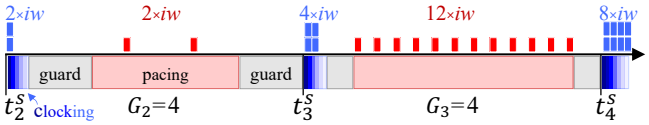


Figure 5: Illustration for two rounds with $G_2 = G_3 = 4$.

Clocking Period: In this period, SUSS sends data packets based on ACK clocking. It starts at time t_i^s upon receiving the first ACK in $round(i)$. The amount of data sent during this period for $round(i)$ is always twice the amount of data sent during the clocking period of $round(i-1)$. Let S_i^{Bdt} denote the amount of data sent in the clocking period for $round(i)$. As illustrated in Fig. 5 where blue boxes represent the data sent during the clocking period, S_i^{Bdt} is doubled in each round, and $S_i^{Bdt} = iw \times 2^{i-1}$ where iw is the initial setting for *cwnd*. Let Δt_i^{Bat} be the amount of time taken to receive the ACKs for the blue packets send in $round(i-1)$. Then Δt_i^{at} can be estimated as follows:

$$\Delta t_i^{at} = \frac{cwnd_{i-1}}{S_{i-1}^{Bdt}} \times \Delta t_i^{Bat}. \tag{9}$$

Pacing Period: The remaining data packets in the data train will be sent at a predetermined pace during this period, as illustrated in Fig. 5 where the red boxes represent the data sent with pacing. Unlike the clocking period that starts at t_i^s upon receiving the first ACK in $round(i)$ and lasts for Δt_i^{Bat}, the challenge for the pacing period is to determine its starting point and duration. Addressing this challenge is crucial because an early start of the pacing period can intensify the burstiness in the current round, while a late start and/or improper setting for its length can affect the clocking period in the next round.

Let S_i^{Rdt} denote the amount of data to be sent during the pacing period of $round(i)$. Since $cwnd_i = S_i^{Bdt} + S_i^{Rdt}$,

$$\begin{aligned} S_i^{Rdt} &= cwnd_i - S_i^{Bdt} \\ &= G_i \times (S_{i-1}^{Bdt} + S_{i-1}^{Rdt}) - S_i^{Bdt}. \end{aligned}$$

Given that $S_i^{Bdt} = iw \times 2^{i-1}$ and $S_{i-1}^{Bdt} = iw \times 2^{i-2}$, we have

$$S_i^{Rdt} = G_i \times S_{i-1}^{Rdt} + (G_i - 2) \times 2^{i-2} \times iw. \tag{10}$$

From Eq. 10, the growth rate of S_i^{Rdt} is higher than that of S_i^{Bdt} when $G_i > 2$, and the proportion of S_i^{Rdt} to $cwnd_i$ varies in different rounds (see Fig. 5). Therefore, the duration and starting time of the pacing period in each round need to be dynamic. In SUSS, we carefully determine the starting time and duration for each pacing period as follows: based on the proportion of red packets in $round(i)$, SUSS allocates $\frac{S_i^{Rdt}}{cwnd_i} \times minRTT$ seconds to the pacing period if $G_i > 2$. Hence, the sending rate during the pacing period can be determined by

$$sendingRate_i = \frac{S_i^{Rdt}}{\frac{S_i^{Rdt}}{cwnd_i} \times minRTT} = \frac{cwnd_i}{minRTT}. \tag{11}$$

Since the duration of the clocking and pacing periods are known, it can be inferred that there is no packet transmission for $minRTT - (\Delta t_i^{Bat} + \frac{S_i^{Rdt}}{cwnd_i} \times minRTT)$ seconds in $round(i)$. Hence, each guard interval in $round(i)$ has a length of $guard_i$ that can be computed as follows:

$$\begin{aligned} guard_i &= \frac{minRTT - (\Delta t_i^{Bat} + \frac{S_i^{Rdt}}{cwnd_i} \times minRTT)}{2} \\ &= \frac{cwnd_i - S_i^{Rdt}}{2 \times cwnd_i} \times minRTT - \frac{\Delta t_i^{Bat}}{2} \\ &= \frac{S_i^{Bdt}}{2 \times cwnd_i} \times minRTT - \frac{\Delta t_i^{Bat}}{2} \end{aligned} \tag{12}$$

Therefore, the pacing period in $round(i)$ can start at $t_i^s + \Delta t_i^{Bat} + guard_i$. The following lemma shows that $guard_i$ always has a positive value when $round(i)$ contains a pacing period.

Lemma 1. *If $round(i)$ contains a pacing period,*

$$guard_i \geq \frac{S_i^{Bdt}}{4 \times cwnd_i} \times minRTT.$$

Proof. If $round(i)$ contains a pacing period, $G_i > 2$. According to the analysis presented in Section 3, a growth factor larger than 2 can be applied in $round(i)$ only if the exponential growth can continue in $round(i+1)$, that is,

$$\Delta t_{i+1}^{at} \leq \frac{minRTT}{2} \tag{13}$$

On the other hand, based on Eq. (9) we have $\Delta t_{i+1}^{at} = \frac{cwnd_i}{S_i^{Bdt}} \times \Delta t_{i+1}^{Bat}$. Therefore, $\Delta t_{i+1}^{Bat} \leq \frac{S_i^{Bdt}}{cwnd_i} \times \frac{minRTT}{2}$. Since $\Delta t_i^{Bat} \leq \Delta t_{i+1}^{Bat}$, we can write

$$\Delta t_i^{Bat} \leq \frac{S_i^{Bdt}}{cwnd_i} \times \frac{minRTT}{2} \tag{14}$$

Based on Eq. (12) and Inequality (14), we get

$$\begin{aligned} guard_i &\geq \frac{S_i^{Bdt}}{2 \times cwnd_i} \times minRTT - \frac{S_i^{Bdt}}{cwnd_i} \times \frac{minRTT}{4} \\ &\geq \frac{S_i^{Bdt}}{4 \times cwnd_i} \times minRTT \end{aligned}$$

□

Fig. 6 illustrates the operation of SUSS based on the examples given in Fig. 4 and Fig. 5. In $round(i)$, G_i is measured at time $t_i^s + \Delta t_i^{Bat}$. Since $G_2 = G_3 = 4$, in $round(2)$ and $round(3)$ additional data packets are sent during pacing periods. At the beginning of $round(2)$, receiving ACKs for iw bits of data results in sending $2iw$ in the clocking period to increase $cwnd$ from iw to $2iw$. Since $G_2 = 4$, $cwnd_2 = 2^{(1+1)} \times cwnd_1 = 4iw$. After passing $guard_2$ seconds, $2iw$ bits of additional packets are paced to increase $cwnd$ to $4iw$. Since these additional packets constitute half of $cwnd_2$, the pacing period in $round(2)$ lasts for half of $minRTT$. $round(3)$ starts by receiving ACKs for $2iw$ bits of data which results in sending $4iw$ and increasing $cwnd$ to $6iw$. Since $G_3 = 4$, we will have $cwnd_3 = 2^{(1+1)} \times cwnd_2 = 16iw$, which means pacing $12iw$ bits of data during three fourths of $minRTT$. Since $G_4 = 2$, data transfer reverts to the traditional slow-start in $round(4)$. The sending for the first $8iw$ is triggered by receiving the ACKs for data sent in the clocking period in $round(3)$, and the sending for the remaining $24iw$ is triggered by receiving the ACKs for data sent in the pacing period in $round(3)$.

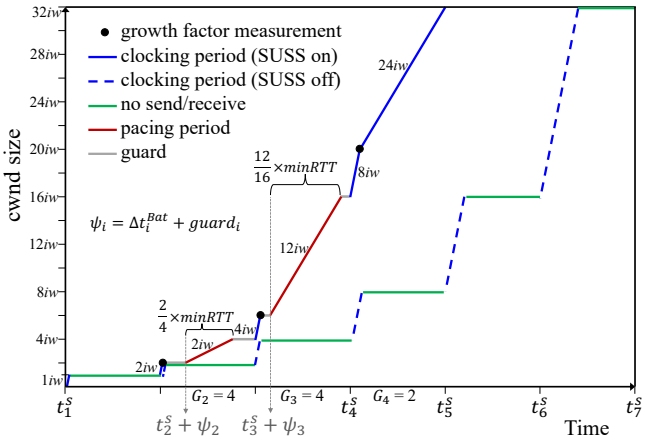


Figure 6: Operation of SUSS for the example shown in Fig. 5.

5 IMPLEMENTATION OF SUSS

We implemented SUSS in Linux kernel version 5.19.10. In our implementation, SUSS extends the memory usage for controlling each TCP connection by only 40 bytes: less than 1% of the total. Furthermore, SUSS induces very marginal CPU overhead, primarily due to the infrequent measurements of the growth factor. Fig. 7 shows the flowchart of SUSS. The major modifications of the Linux kernel are summarized under the following three topics:

Measurement of G_i: As shown in Fig. 7, G_i is calculated in $round(i)$ when the last blue ACK is received. In CUBIC, the slow-start algorithm uses the packet sequence number to identify the start of each round. Similarly, SUSS uses the sequence numbers for the first and last packets sent in the clocking period of the previous round to determine the head and tail of the blue part of the received ACK train in each round. Based on the measured Δt_i^{Bat}, SUSS estimates Δt_i^{at} based on Eq. (9) and determines G_i by choosing the largest k that meets **Condition 1** and **Condition 2**.

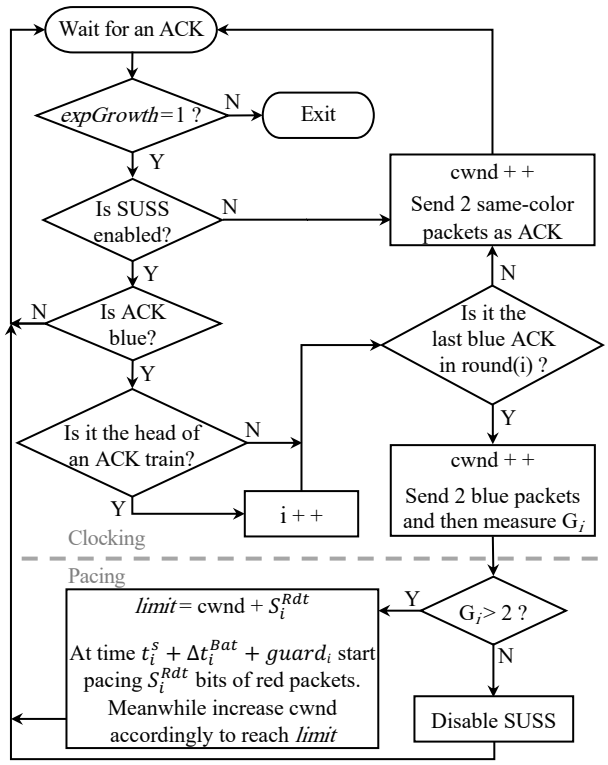


Figure 7: SUSS flowchart.

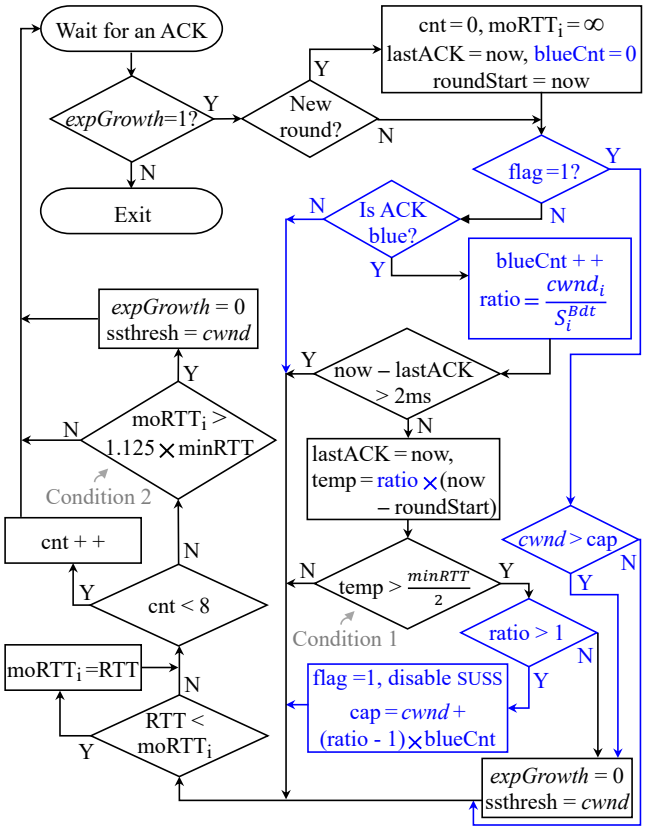


Figure 8: Modified HyStart flowchart (changes shown in blue).

Data transmission in the pacing period: SUSS will switch to the pacing mode only when $G_i > 2$. Since G_i is calculated at time $t_i^s + \Delta t_i^{Bat}$, pacing the additional packets (red packets) starts $guard_i$ seconds later. The value of *cwnd* grows gradually as packets are paced. This helps avoid unnecessary *cwnd* enlargement in the rare case when the pacing period is not completed (e.g., the exponential growth stops due to packet loss). Once $G_i \leq 2$, SUSS is disabled and the process of sending packets will revert to the traditional slow-start. **Modification to HyStart:** Due to packet pacing, the red part of the ACK train cannot accurately represent the network path condition. Hence, HyStart is modified accordingly as shown in Fig. 8, where modifications are shown in blue. At the beginning of each round, a set of variables including *cnt* (number of received ACKs) and *blueCnt* (number of received blue ACKs) are initialized. Then HyStart evaluates the two criteria to determine if it is time to stop the exponential growth. Since **Condition 1** relies on analyzing the proximity between successive ACKs and Δt_i^{at}, we modify HyStart to measure Δt_i^{at} just based on the blue packets by scaling the elapsed time since the start of the current round (Eq. 9). To this end, we use a local variable *ratio* to represent the ratio of the size of the data train to the size of its blue part in $round(i)$. Once the estimated length of the ACK train exceeds half of *minRTT*, it is imperative to stop the exponential growth (i.e., *expGrowth* = 0). However, in the scenario where the elapsed time has already been scaled, a cap must be defined to postpone stopping the growth. A flag is set to indicate that the cap has already been initialized and *expGrowth* = 0 when *cwnd* exceeds the cap.

6 PERFORMANCE EVALUATION

In this section, we conduct experiments to evaluate the practicality and effectiveness of SUSS in real-world scenarios. Our experiments employ a diverse set of off-the-shelf hardware and cloud servers connected through different types of communication links. For testing under controlled settings as well as determined extreme conditions, we also conduct experiments in a local testbed. We examine the performance of SUSS primarily in terms of delivery rate and FCT, while also investigating *RTT*, loss rate, fairness, and system stability to ensure that the proposed enhancements do not adversely affect these crucial aspects of TCP performance [11].

6.1 Experiment Setup and Evaluation Methodology

For the internet-scale testbed, we set up a web server from which a client will download a file. We vary the location of the web server, the type of clients, and the communication link for the last hop.
Server deployment: We recompile the Linux kernel with SUSS and deploy one stand-alone server connected to a university campus network in NZ and multiple servers in the cloud data centers

provided by Google, Oracle and Microsoft. Since Google uses BBR as the default CCA whereas Oracle and Microsoft Azure both use CUBIC, we only present the test results for Google and Oracle data centers due to space constraints, but we did observe similar results with Microsoft Azure. We have positioned the cloud servers in the following data center locations: three Google data centers in the eastern United States, Tokyo, and Singapore; and three Oracle data centers in the western United States, Sydney, and London. Such a setup enables us to test SUSS under different RTTs, BDPs, and cross traffic patterns.

Client deployment: Since SUSS is a server-side approach, no changes need to be applied at the client side. We have used various end-user devices located in Sweden and NZ as clients, including desktop computers (Windows 10, wired Ethernet), laptops (Linux 5.19.10, WiFi), and mobile phones (Android 12 / iOS 17.3, 4G/5G).

TCP traffic generation: `Apache2` is used as the web server. We also tested alternative tools such as `netcat` and `iperf`, and did not detect any disparities on assessing the performance of SUSS. To exclude any upper layer optimizations within web browsers, each client downloads the file by executing either `wget` or `curl` from a command line interface.

In our local testbed, we have instead established a network consisting of five separate Linux-based client-server pairs. These pairs are interconnected through a dumbbell network topology, utilizing two routers. The central element of this setup is the bottleneck router, which is also a Linux machine. On this router, we employ `netem`, a component of the Traffic Control utility in Linux, to accurately simulate network conditions. It allows us to configure parameters such as the speed of the bottleneck link and the size of the bottleneck buffer, enabling precise control over the network environment for our experiments.

Evaluation Methodology: To ensure accurate measurements, we modify the Linux kernel of the servers to effectively enable sending messages to the kernel log. This allows us to collect TCP state variables, including the number of inflight packets, *cwnd*, *RTT*, total amount of delivered data, and a few customized metrics defined to monitor and assess SUSS. This kernel-level approach can provide much more precise measurements than methods based on traditional packet analyzers. For each testing scenario, we download a file from the server, alternating SUSS between on and off with each test repeated for 50 iterations.

6.2 Throughput, RTT, and FCT

In our first experiment, we used a 4G Android smartphone in NZ to download a file from a server deployed in Google's data center located in the eastern United States, utilizing CUBIC. Fig. 9 shows the dynamics of *cwnd* and *RTT* with and without SUSS. When the *cwnd* reaches about 1300 packets, both methods transit from exponential growth to linear growth, triggered by HyStart. CUBIC with SUSS exhibits a faster and smoother increase in *cwnd*, taking roughly half the time to reach a congestion window size of 1300 packets compared to CUBIC without SUSS. However, the accelerated increase in *cwnd* does not incur extra end-to-end delay, as *RTT* in SUSS remains almost unchanged in the early rounds before stopping exponential growth. This is because SUSS accelerates the

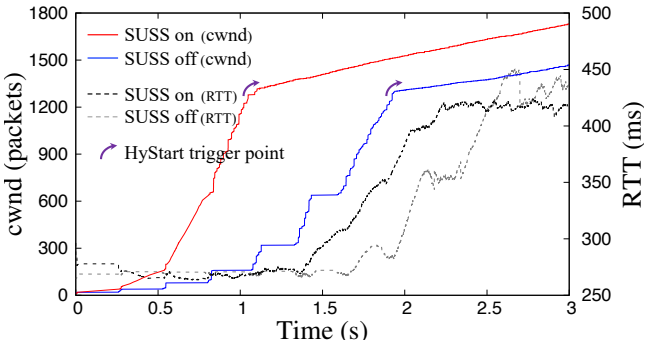


Figure 9: SUSS changes the *cwnd* and *RTT* growth dynamic.

growth of *cwnd* only when it is significantly below *cwnd**, and pacing the additional packets caused by *cwnd* acceleration effectively mitigates burstiness to avoid instantaneous queuing delay and even packet loss.

Fig. 10 shows the total amount of data delivered with and without SUSS over time when CUBIC is in use. It can be seen that SUSS has brought substantial improvements. For example, two seconds after the connection is established, CUBIC without SUSS delivers 2 MB, whereas CUBIC with SUSS delivers three times more, achieving over a 200% improvement in total delivered data. This figure also shows that SUSS can reach *cwnd** more quickly than the traditional slow-start, as indicated by the black rectangles. Furthermore, SUSS avoids overshooting *cwnd* and maintains the consistent delivery rate θ, similar to slow-start (see parallel dashed green lines in Fig. 10).

To further evaluate the robustness of SUSS, we repeat the experiment by varying the locations of the server and the client and the type of communication link for the last hop. Given our internet-scale testbed includes seven servers and four types of communication links, this setup enables us to conduct the experiment under 28 different scenarios. In each testing scenario, we measure the Flow Completion Time (FCT) achieved by three schemes — CUBIC with SUSS on, CUBIC with SUSS off, and BBR — under different settings on flow size. Fig. 11 shows the results for four of these scenarios where the server is located in a Google data center in Tokyo. In all

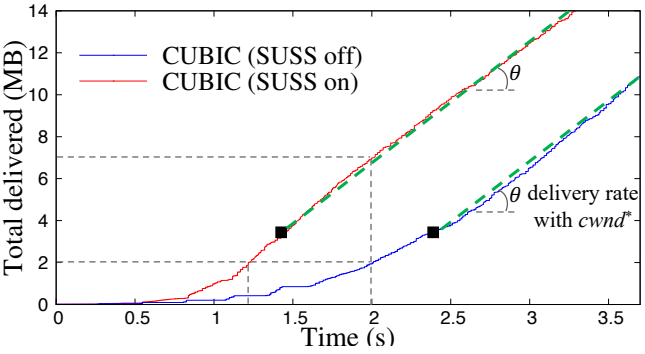
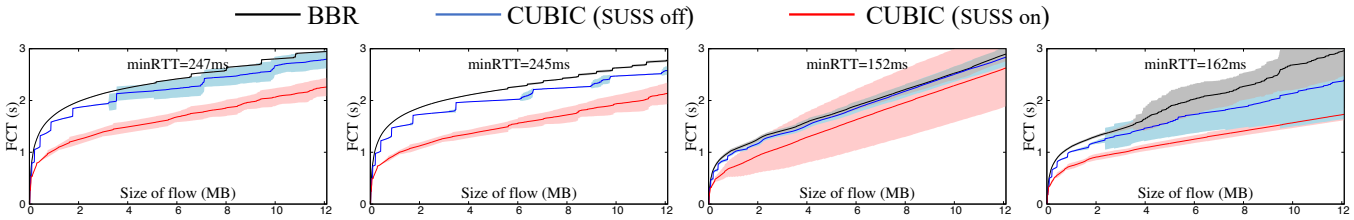
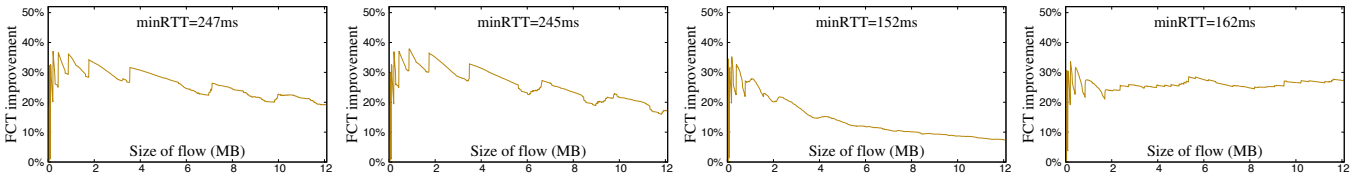


Figure 10: SUSS improves data delivery.



(a) Google; Tokyo; 5G link (b) Google; Tokyo; Wired link (c) Google; Tokyo; WiFi link (d) Google; Tokyo; 4G link

Figure 11: FCT in BBR, CUBIC with SUSS on, and CUBIC with SUSS off. The server is located in a Google data center in Tokyo. The client is in Sweden when the last hop is 5G/Wired (a and b) and in New Zealand when it is WiFi/4G (c and d).



(a) Google; Tokyo; 5G link (b) Google; Tokyo; Wired link (c) Google; Tokyo; WiFi link (d) Google; Tokyo; 4G link

Figure 12: FCT improvement achieved by SUSS for scenarios illustrated in Fig. 11.

sub-figures, the results represent the average value over 50 iterations, with the standard deviation depicted using a shaded area. It can be seen that CUBIC with SUSS on consistently outperforms the other two schemes with no negative impacts. The wireless scenarios experience more deviations than the wired scenarios due to the bandwidth variations of wireless links.

Fig. 12 shows the average improvement in FCT achieved by CUBIC when SUSS is enabled for the four scenarios presented in Fig. 11. It can be seen that accelerating slow-start can yield significant gain for small flows, as many small flows predominantly reside within the slow-start phase. For example, when the flow size is not larger than 2 MB, more than 20% improvement in FCT has been achieved in all these four testing scenarios. As the flow size further increases, the achieved improvement diminishes, except in Fig. 12(d). Therefore, even if the exponential growth is accelerated, it will not result in a significant improvement for large flows. For the scenario in Fig. 12(d), CUBIC's HyStart stops the exponential growth earlier and transits to linear growth before reaching $cwnd^*$, due to a high level of jitter [5]. However, even in such scenarios, SUSS enhances data delivery rate, achieving better performance than its competitors.

The results for all the 28 testing scenarios are given in Appendix C. Overall, CUBIC with SUSS outperforms CUBIC without SUSS in all the 28 testing scenarios, and performs slightly worse than BBR in only one of the 28 testing scenarios (Fig. 18 (d7)) due to the early stop of exponential growth caused by HyStart's stopping criteria. This demonstrates the robustness and superiority of SUSS across diverse testing scenarios.

As discussed earlier, SUSS enhances the FCT of small flows, which are prevalent on the Internet. To investigate the behavior of SUSS when the TCP flow is large, we conducted an experiment where a client in Sydney downloads a 100MB file from a server located in the eastern United States. Both TCP endpoints are located in Google's data centers. As shown in Fig. 13, there is a significant improvement during the early megabytes of the transfer, which subsequently tapers off to negligible levels. This suggests that while SUSS enhances the performance of small-sized flows, it does not increase *cwnd* beyond its optimal value, nor does it impact the FCT of larger flows. These findings corroborate the experiments presented in Section 6.4 and Section 6.5.

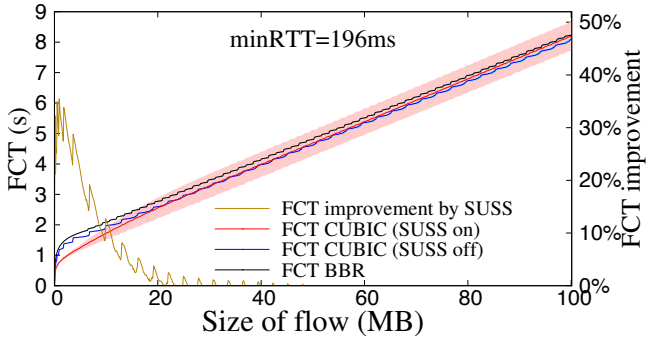


Figure 13: SUSS has no impact on large TCP flows.

6.3 Packet loss

High-density bursts cause TCP packet loss when the bottleneck buffer is under pressure [32]. We have conducted experiments to evaluate the impact of SUSS's acceleration in *cwnd* on packet loss rate. Fig. 14 shows the packet loss rates for a scenario where the

server is deployed in an Oracle data center in London, the client is a 5G device in Sweden, the CCA is CUBIC (with and without SUSS), and the flow size is varied from 2 MB to 40 MB. It can be seen that CUBIC has experienced much less packet loss when SUSS is enabled. This is because SUSS employs a pacing mechanism to reduce packet burstiness. This helps to spread data packets instead of sending them in bursts, resulting in a lower loss rate. On the other hand, when SUSS is disabled, CUBIC sends a new data packet upon receiving an ACK. During early RTTs, data packets travel close to each other when CUBIC is in use. This bursty transmission and the exponential growth lead to a higher packet density, which increases the loss rate when buffer overflow occurs. The above explanation can be well justified using Fig. 6. For CUBIC without SUSS, the blue dashed lines in Fig. 6 represent bursty transmissions. It can be seen that $4iw$, $8iw$, and $16iw$ were sent in bursts at the beginning of round 4, 5 and 6, respectively. This results in very high packet density, which increases the loss rate when buffer overflow occurs. For CUBIC with SUSS, the blue solid lines in Fig. 6 represent bursty transmissions in the clocking periods, and the red solid lines represent packet transmissions in pacing periods. It can be seen that the number of packets transmitted in burst (i.e., transmitted in the clocking periods) is much less than that in CUBIC without SUSS. A large portion of the packets are sent in the pacing periods, e.g., $2iw$in round 2 and 16 iw in round 3. This can significantly reduce packet density and thereby reduce the packet loss rate.

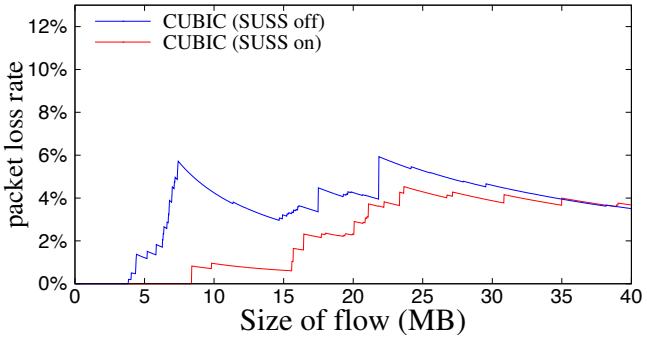


Figure 14: Comparing packet loss when a 5G client in Sweden downloads a file from an Oracle server in London.

It can also be observed from Fig. 14 that the packet loss rates for CUBIC with and without SUSS converge when the flow size increases. This is because SUSS only affects the slow-start phase. As the flow size increases, the proportion of time a flow spends in the slow-start phase decreases, causing the packet loss rate to be dominated by losses occurring during the steady-state phase.

The packet loss rates for all testing scenarios and the comparison with BBR can be found in Fig. 17 in Appendix C. Since SUSS affects only the slow-start phase, we just show the packet loss rate for flow size up to 12 MB in Fig. 17. Overall, CUBIC with SUSS performs better than CUBIC without SUSS, since SUSS uses packet pacing to effectively mitigate traffic burstiness caused by the enlarging of *cwnd*. It is worth noting that BBR demonstrates negligible packet loss, attributed to its efficient pacing mechanism in both the slow-start phase and the steady-state phase. As shown in Fig. 17, BBR experiences packet loss in only one scenario (i.e., c2) because it does not stop the exponential growth when encountering a few packet losses. This behavior contrasts with loss-based CCAs such as CUBIC. We also observed that the packet loss rate is noticeable in testing scenarios using Oracle servers and high-speed links (i.e., Fig. 18. a1, a3, b1, b3, b6), but negligible in other tested scenarios. This is due to high-density bursts lasting longer in high-speed links, while network paths with smaller *BtlBw*s lead to reduced density in data trains due to extended ACK trains.

6.4 Fairness

Accelerating the growth of the congestion window could possibly lead to unfairness in bandwidth competition. However, our experimental evidence confirms that SUSS not only avoids causing unfairness but also actively enhances fairness in scenarios with congested network paths. As shown in Fig. 9, the exponential growth ends at almost the same *cwnd* for both scenarios, whether SUSS is on or off. Fig. 10 shows that SUSS achieves a rapid increase in the delivery rate in the early rounds due to the accelerated *cwnd* growth, but once the TCP connection approaches its fair share, the delivery rate plateaus around its optimal value (θ). When reaching their fair share, both methods exhibit parallel growth, as shown in Fig. 10. This means that, following the rapid rate increase, SUSS constrains the connection from demanding more bandwidth than its share, thereby preserving the achieved fairness.

To further evaluate fairness, we follow the recommendation of RFC 5166 [11] to use the Jain's Index [17] as the fairness coefficient (F), based on the goodput to indicate how fair the bandwidth is shared between concurrent flows.

$$F = \frac{(\sum x_i)^2}{n \times \sum (x_i^2)}$$

where n is the number of flows and x_i is the goodput of the i-th flow. The closer F is to 1, the better the level of fairness is achieved. To this end, in our local testbed featuring a 50 Mbps bottleneck bandwidth and CUBIC congestion control, four clients initiate file downloads from their respective servers at 2-second intervals. After 60 seconds when a level of fairness is achieved among the four flows, the fifth flow starts downloading (similar to the test shown in Fig. 2(a)). We repeated the test for different *minRTT*s and bottleneck buffer sizes. As shown in Fig. 15, the initiation of the fifth flow results in an immediate decrease in the value of F, exhibiting a prolonged delay in approaching 1 when SUSS is off. This effect is more pronounced when the preceding four flows have a higher number of in-flight packets, a direct consequence of longer *RTT*s and larger buffer sizes. Under these conditions, the new flow experiences a longer time to increase its *cwnd* from *iw* to $cwnd^*$, particularly if an early buffer overflow stops the exponential growth.

6.5 Stability

To assess the stability of a large TCP flow when competing with multiple small flows for bottleneck bandwidth, we devised a specific testing scenario in our local testbed: a single large flow transfers data while twelve 2 MB TCP flows with different *minRTT*s are initiated sequentially at 2-second intervals, as illustrated in Fig. 16.

We conducted the experiments with different configurations of the CCA algorithm, bottleneck buffer size and *RTT*. Specifically, we

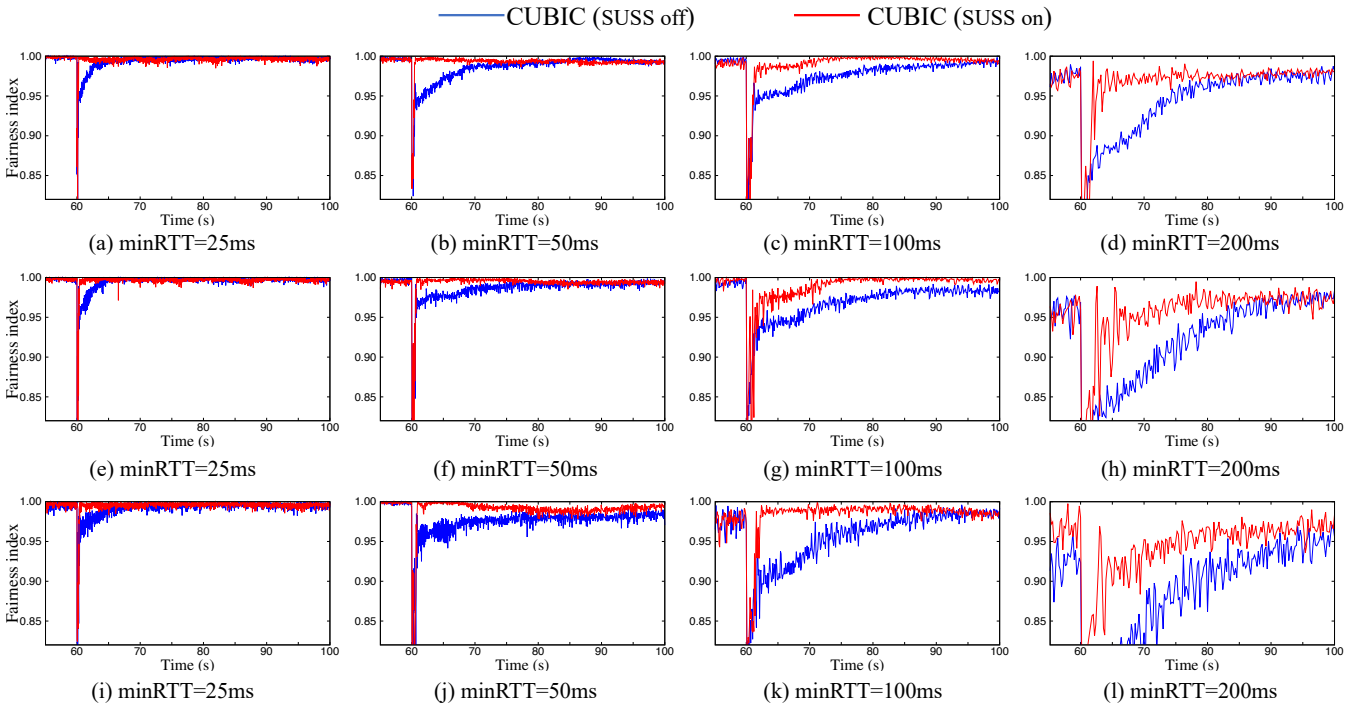


Figure 15: Comparative fairness analysis under varying *minRTT* and bottleneck buffer sizes (1BDP for sub-figures a–d, 1.5BDP for sub-figures e–h, and 2BDP for sub-figures i–l), in our local testbed.

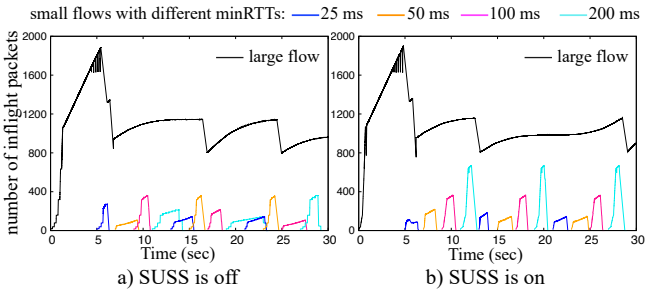


Figure 16: A large flow facing the initiation of multiple concurrent small flows. In this example, the large flow's *minRTT* is 200 ms, CUBIC is in use and the buffer size is one BDP.

tested three CCAs for the large flow (CUBIC, BBRv1, and BBRv2), and two CCAs for the small flows (CUBIC with SUSS on, CUBIC with SUSS off). For each configuration on CCA, we further tested two distinct bottleneck buffer sizes (1 *BDP* and 2 *BDP*). For each configuration on CCA and bottleneck buffer size, we also tested over four different *RTT*s (25 ms, 50 ms, 100 ms and 200 ms) for the large flow. In total, there are $3 \times 2 \times 4$ tests conducted in this experiment, with each one consisting of 50 iterations. Adjustments to the bottleneck buffer size and the flows' *RTT*s were implemented using the `netem` tool.

The experimental results are summarized in Table 1. It can be seen that, on average, the improvement in the FCT of small flows is 32% when the large flow utilizes CUBIC, 28% with BBRv1, and 26% with BBRv2. Notably, these enhancements are achieved without a noticeable negative impact on the FCT of the large flow, as highlighted by the numbers in red color, thereby underscoring the stability of large flow performance in the tested scenarios.

This experiment further reveals the importance of resilience for a CCA to effectively handle scenarios where the large flow releases bandwidth during congestion and subsequently reclaims it once the congestion has been resolved. For instance, BBRv1 demonstrates resilience in releasing bandwidth and swiftly reoccupying the available bandwidth as soon as a small flow ends. This phenomenon is stronger in scenarios with smaller *minRTT*s. In contrast, CUBIC tends to reduce its transmission rate more rapidly upon packet loss (which is prevalent in scenarios with shallow buffers) and exhibits a comparatively slower response in reoccupying the freed bandwidth. This difference in behavior leads to higher FCT values, as indicated by the blue text within Table 1.

7 CONCLUSION AND FUTURE WORK

This paper presents SUSS, a model that safely mitigates the network under-utilization that typically occurs during TCP slow-start. SUSS helps speed up the increasing of the congestion window (*cwnd*) toward its optimal size within each TCP connection. SUSS combines packet clocking and pacing techniques in order to control the pattern of packets sent and to ensure appropriate termination of

Buffer size	minRTT of large flow	SUSS off		SUSS on		
		FCT of large flow	FCT of small flows	FCT of large flow	FCT of small flows	Improvement for small flows
1 BDP	25 ms	30.4	1.33	31.2	0.94	29%
	50 ms	37.9	1.18	35.6	0.98	17%
	100 ms	29.0	1.42	28.5	1.22	14%
	200 ms	27.2	2.69	26.9	2.03	25%
2 BDP	25 ms	27.7	1.17	25.7	1.05	10%
	50 ms	26.3	1.54	25.7	1.30	16%
	100 ms	25.8	3.30	25.7	2.09	37%
	200 ms	26.2	5.63	26.1	2.82	50%
Average		28.8	2.28	28.2	1.55	**32%**

(a) The large flow employs CUBIC

Buffer size	minRTT of large flow	SUSS off		SUSS on		
		FCT of large flow	FCT of small flows	FCT of large flow	FCT of small flows	Improvement for small flows
1 BDP	25 ms	24.1	20.07	24.3	17.54	29%
	50 ms	24.6	13.64	25.2	8.28	17%
	100 ms	25.7	8.11	26.2	4.30	14%
	200 ms	27.6	4.43	28.2	2.52	25%
2 BDP	25 ms	25.5	2.24	25.7	1.45	10%
	50 ms	25.6	2.29	25.8	1.66	16%
	100 ms	25.9	2.75	26.1	1.90	37%
	200 ms	26.3	5.70	26.5	4.92	50%
Average		25.7	7.40	26.0	5.32	**28%**

(b) The large flow employs BBRv1

Buffer size	minRTT of large flow	SUSS off		SUSS on		
		FCT of large flow	FCT of small flows	FCT of large flow	FCT of small flows	Improvement for small flows
1 BDP	25 ms	28.5	1.44	29.2	1.04	29%
	50 ms	27.3	1.49	28.4	1.16	17%
	100 ms	27.0	1.69	27.2	1.34	14%
	200 ms	27.5	3.13	28.0	1.83	25%
2 BDP	25 ms	27.0	1.81	28.0	1.23	10%
	50 ms	26.1	1.90	26.6	1.38	16%
	100 ms	26.2	2.29	26.4	1.72	37%
	200 ms	26.4	5.01	26.6	4.17	50%
Average		27.0	2.35	27.6	1.73	**26%**

(c) The large flow employs BBRv2

Table 1: SUSS improves FCT of small CUBIC flows without compromising the stability of a concurrent, large flow.

the exponential growth phase. This helps ensure that the growth of *cwnd* can be accelerated without increasing disruptive packet loss. SUSS is integrated into the CUBIC implementation in the Linux kernel, and is released as open source software. A promising future research direction is integrating SUSS with BBR. Like CUBIC, BBR adheres to the exponential growth dynamics of traditional slow-start and under-utilizes bottleneck bandwidth in early *RTT*s. Integrating SUSS with BBR could optimize bandwidth utilization and improve FCT of small BBR flows.

SUSS is tested on a wide range of real-world network scenarios, and the experimental results demonstrate its efficacy in improving the TCP performance at the start of each flow. Thus SUSS helps improve the performance of small data transfers, such as web browsing activities as well as the images and short-form videos commonly encountered in today's popular social media platforms.

This work does not raise any ethical issues.

REFERENCES

[1] Thomas E Anderson, Andy Collins, Arvind Krishnamurthy, and John Zahorjan. 2006. PCP: Efficient Endpoint Congestion Control. In *Proceedings of the 3rd conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, San Jose, CA.

[2] Mahdi Arghavani, Haibo Zhang, David Eyers, and Abbas Arghavani. 2020. StopEG: Detecting when to stop exponential growth in TCP slow-start. In *Proceedings of the IEEE 45th Conference on Local Computer Networks (LCN)*. IEEE, 77–87.

[3] Praveen Balasubramanian, Yi Huang, and Matt Olson. 2023. HyStart++: Modified Slow Start for TCP. RFC 9406. https://doi.org/10.17487/RFC9406

[4] Robert John Briscoe. 2018. Fast Friendly Start for a Data Flow. US Patent 9,860,184.

[5] Philipp Bruhn, Mirja Kuehlewind, and Maciej Muehleisen. 2023. Performance and improvements of TCP CUBIC in low-delay cellular networks. *Computer Networks* 224 (2023), 109609.

[6] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *Queue* 14, 5 (2016), 20–53.

[7] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Kevin Yang, Ian Swett, Victor Vasiliev, Bin Wu, Luke Hsiao, et al. 2019. BBRv2: A model-based congestion control performance optimization. In *Proceedings of IETF 106th Meeting*. 1–32.

[8] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. 2013. Increasing TCP's Initial Window. RFC 6928. https://doi.org/10.17487/RFC6928

[9] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. 2005. Processor Sharing Flows in the Internet. In *Proceedings of the IEEE/ACM International Symposium on Quality of Service (IWQoS)*. 271–285.

[10] Nandita Dukkipati and Nick McKeown. 2006. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review* 36, 1 (2006), 59–62.

[11] Sally Floyd. 2008. Metrics for the Evaluation of Congestion Control Mechanisms. RFC 5166. https://doi.org/10.17487/RFC5166

[12] Lingfeng Guo and Jack YB Lee. 2020. Stateful-TCP—A New Approach to Accelerate TCP Slow-Start. *IEEE Access* 8 (2020), 195955–195970.

[13] Lingfeng Guo, Yan Liu, Wenzheng Yang, Yuming Zhang, and Jack YB Lee. 2021. Stateful-BBR–An Enhanced TCP for Emerging High-Bandwidth Mobile Networks. In *Proceedings of the IEEE/ACM 29th International Symposium on Quality of Service (IWQoS)*. 1–9.

[14] Sangtae Ha and Injong Rhee. 2011. Taming the elephants: New TCP slow start. *Computer Networks* 55, 9 (2011), 2092–2110.

[15] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

[16] Toke Høiland-Jørgensen, Paul McKenney, Dave Taht, Jim Gettys, and Eric Dumazet. 2018. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290. https://doi.org/10.17487/RFC8290

[17] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* 21 (Sept. 1984).

[18] Haiqing Jiang, Zeyu Liu, Yaogong Wang, Kyunghan Lee, and Injong Rhee. 2012. Understanding bufferbloat in cellular networks. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*. 1–6.

[19] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. 2021. Flow length and size distributions in campus Internet traffic. *Computer Communications* 167 (2021), 15–30.

[20] Lingang Li, Yongrui Chen, and Zhijun Li. 2023. Small Chunks can Talk: Fast Bandwidth Estimation without Filling up the Bottleneck Link. In *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[21] Qingxi Li, Mo Dong, and P Brighten Godfrey. 2015. Halfback: Running Short Flows Quickly and Safely. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–13.

[22] Dan Liu, Mark Allman, Shudong Jin, and Limin Wang. 2007. Congestion Control Without a Startup Phase. In *Proc. PFLDnet*. 61–66.

[23] Matt Mathis and Jamshid Mahdavi. 2019. Deprecating the TCP Macroscopic Model. *ACM SIGCOMM Computer Communication Review* 49, 5 (2019), 63–68.

[24] Joakim Misund and Bob Briscoe. 2019. Paced Chirping: Rapid flow start with very low queuing delay. In *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 798–804.

[25] Xiaohui Nie, Youjian Zhao, Guo Chen, Kaixin Sui, Yazheng Chen, Dan Pei, Miao Zhang, and Jiyang Zhang. 2017. TCP Wise: One Initial Congestion Window Is Not Enough. In *Proceedings of the IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. 1–8.

[26] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. 2019. Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1231–1247.

[27] Dr. Craig Partridge, Mark Allman, and Sally Floyd. 2002. Increasing TCP's Initial Window. RFC 3390. https://doi.org/10.17487/RFC3390

[28] Jan Rüth, Ike Kunze, and Oliver Hohlfeld. 2019. TCP's initial window—deployment in the wild and its impact on performance. *IEEE Transactions on Network and Service Management* 16, 2 (2019), 389–402.

[29] Renaud Sallantin, Cédric Baudoin, Emmanuel Chaput, Fabrice Arnal, Emmanuel Dubois, and André-Luc Beylot. 2014. A TCP model for short-lived flows to validate initial spreading. In *39th Annual IEEE Conference on Local Computer Networks*. 177–184. https://doi.org/10.1109/LCN.2014.6925770

[30] Pasi Sarolahti, Amit Jain, Sally Floyd, and Mark Allman. 2007. Quick-Start for TCP and IP. RFC 4782. https://doi.org/10.17487/RFC4782

[31] Belma Turkovic and Fernando Kuipers. 2020. P4air: Increasing Fairness Among Competing Congestion Control Algorithms. In *Proceedings of the IEEE 28th International Conference on Network Protocols (ICNP)*. 1–12.
[32] David Wei, Pei Cao, Steven Low, and Caltech EAS. 2006. TCP Pacing Revisited. In *Proceedings of IEEE INFOCOM*, Vol. 2. Citeseer, 3.
[33] Lisong Xu, Sangtae Ha, Injong Rhee, Vidhi Goel, and Lars Eggert. 2023. CUBIC for Fast and Long-Distance Networks. RFC 9438. https://doi.org/10.17487/RFC9438
[34] Yan Zhang, Nirwan Ansari, Mingquan Wu, and Heather Yu. 2012. AFStart: An adaptive fast TCP slow start for wide area networks. In *2012 IEEE International Conference on Communications (ICC)*. IEEE, 1260–1264.

APPENDICES

Appendices are supporting material that has not been peer-reviewed.

A GENERALIZED SUSS

As previously discussed, SUSS aims to accelerate the growth of *cwnd* in the current round, provided that an increase in *cwnd* is anticipated in the subsequent round. This appendix extends SUSS to account for potential *cwnd* growth beyond the next round, aiming to enhance the expansion of *cwnd* in the current round based on expected increases in future rounds. The primary challenge for SUSS involves determining the probable number of rounds, k, over which the exponential growth of *cwnd* can be projected to continue. To tackle this, SUSS utilizes the conditions outlined in Section 3, which we will now restate and elaborate upon.

Formulating Condition 1 for SUSS: To assess if Condition 1 is satisfied over the forthcoming k rounds, SUSS examines the following inequality:

$$\Delta t^{at}_{i+k} \leq \frac{minRTT}{2},\ \forall k \in \{1, 2, 3, \ldots, k_{max}\} \tag{15}$$

where k_{max} denotes a user-defined parameter. Given that Δt^{at}_{i+k} is unknown in round i, it requires estimation. As described in Section 3, we approximate that $\Delta t^{at}_{i+1} \approx 2 \times \Delta t^{at}_{i}$. By applying this logic iteratively and assuming stable network conditions in the upcoming k rounds, we estimate Δt^{at}_{i+k} as:

$$\Delta t^{at}_{i+k} \approx 2^k \times \Delta t^{at}_{i}. \tag{16}$$

This suggests Eq. 15 is met, and exponential growth continues through $round(i+k)$ if:

$$\Delta t^{at}_{i} \leq \frac{minRTT}{2^{k+1}}. \tag{17}$$

Formulating Condition 2 for SUSS: As detailed in Section 3, considering that *minRTT* was last updated r rounds ago, we estimate the average increase in queuing delay since then as $(moRTT_i - minRTT)/r$ seconds per round. Hence, $moRTT_{i+k}$ is approximated by:

$$moRTT_{i+k} = moRTT_i + k \times \frac{(moRTT_i - minRTT)}{r}. \tag{18}$$

To confirm Condition 2, the sender should verify:

$$moRTT_i + k \times \frac{(moRTT_i - minRTT)}{r} \leq 1.125 \times minRTT. \tag{19}$$

To identify k (and thus compute G_i), SUSS implements an iterative approach upon the completion of the ACK train in $round(i)$, as detailed in Algorithm 1. This process starts with initialization,

Algorithm 1 G_i calculation

```
1:  k ← 0
2:  while k ≤ k_max do
3:      if Δt_i^at ≤ minRTT/2^(k+1) and (r == 0 or moRTT_i + k × (moRTT_i − minRTT)/r ≤ 1.125 × minRTT) then
4:          k ← k + 1
5:      else
6:          break
7:      end if
8:  end while
9:  G_i ← 2^(k+1)
10: return G_i
```

followed by a loop that executes up to k_{max} iterations, reflecting the rounds for which SUSS evaluates the conditions. If the conditions are satisfied within each iteration (line 3), k is incremented—otherwise, the loop terminates. If the loop persists, the sender assesses the conditions for an additional round. Upon loop completion or termination, the algorithm calculates and returns G_i (lines 9–10).

B IMPACT OF VARIATION IN *BtlBw*

To investigate the impact of *BtlBw* variation on SUSS, 1400 iterations of experiments were conducted across seven servers in various geographical locations, using four types of end devices: 4G, 5G, WiFi, and wired links. Section 6 provides detailed descriptions of this experimental campaign. Fig. 18 show the standard deviations of variations in *BtlBw* using a shaded area. It can be seen that wireless testing scenarios, particularly 4G and WiFi (the sub-figures in the third and fourth columns of Fig. 18), experience larger deviations compared to wired scenarios due to higher variability in bandwidth. However, these experiments confirm that the performance of CUBIC with SUSS on is not significantly affected by even large variations in *BtlBw*. For example, in Fig. 18.c2, despite large variations in *BtlBw* (indicated by the wide standard deviation of FCT), CUBIC with SUSS on outperforms both BBR and standard CUBIC. Particularly, it improves the performance of standard CUBIC by 18% to 40%, depending on the flow size. This trend is consistent across other scenarios, with SUSS improvements rarely falling below 20%, especially when the flow size is below 4 MB. From our experiments, we have two key observations:

Observation 1: In most scenarios, the variation in *BtlBw* does not result in *cwnd* exceeding *cwnd**. This can be explained as follows: (1) If *BtlBw* drops when *cwnd* is far below *cwnd**, the bottleneck link is not saturated, indicating an early phase of slow-start with a small *cwnd*. In this case, doubling or quadrupling *cwnd* is less likely to overflow the buffer, so the average performance of SUSS remains relatively unaffected. (2) If *BtlBw* drops when *cwnd* is close to *cwnd**, a sharp drop in *BtlBw* increases the ACK train length and queueing delay. Consequently, the conditions for quadrupling the growth factor (as per Eqs. 6 and 8) are not met, making quadrupling unlikely. Thus, SUSS will perform similarly to traditional slow-start, with both methods doubling *cwnd*.

Observation 2: In rare wireless scenarios, variations in *BtlBw* cause *cwnd* to exceed *cwnd**. However, since the last hop wireless

link is usually the bottleneck, the last hop router commonly employs deep buffers to mitigate the side effects caused by bandwidth variation in wireless networks [18, 23]. Hence, any excess of $cwnd_i$ beyond $cwnd^*$ due to variations in $BtlBw$ is still manageable, as the bottleneck router's buffer can often accommodate the extra data packets.

C EXPERIMENTAL RESULTS FOR ALL 28 TESTING SCENARIOS

Fig. 17 shows the packet loss rates of the three compared schemes corresponding to the test scenarios in Fig. 18. Some test scenarios did not experience packet loss and are therefore not included in this figure.

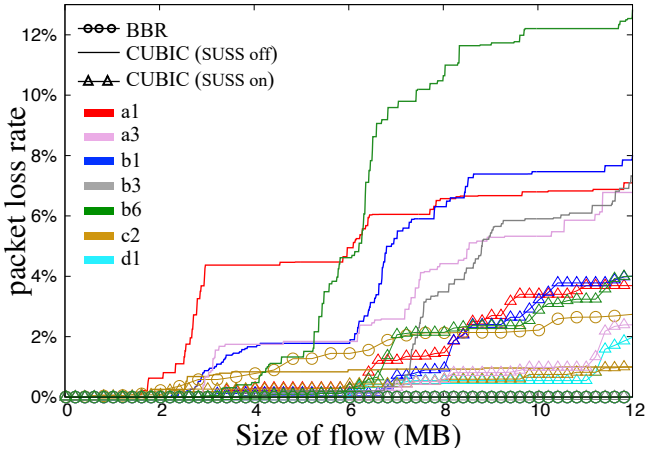


Figure 17: Comparing packet loss for all the scenarios shown in Fig. 18.

As detailed in Section 6, our internet-scale testbed consists of seven different servers and four types of last-hop communication links. Accordingly, Fig. 18 presents the complete set of experimental results, arranged in a matrix of seven rows and four columns. In this figure, the left y-axis represents FCT for CUBIC and BBR, while the right y-axis indicates the FCT improvement in CUBIC achieved by SUSS.

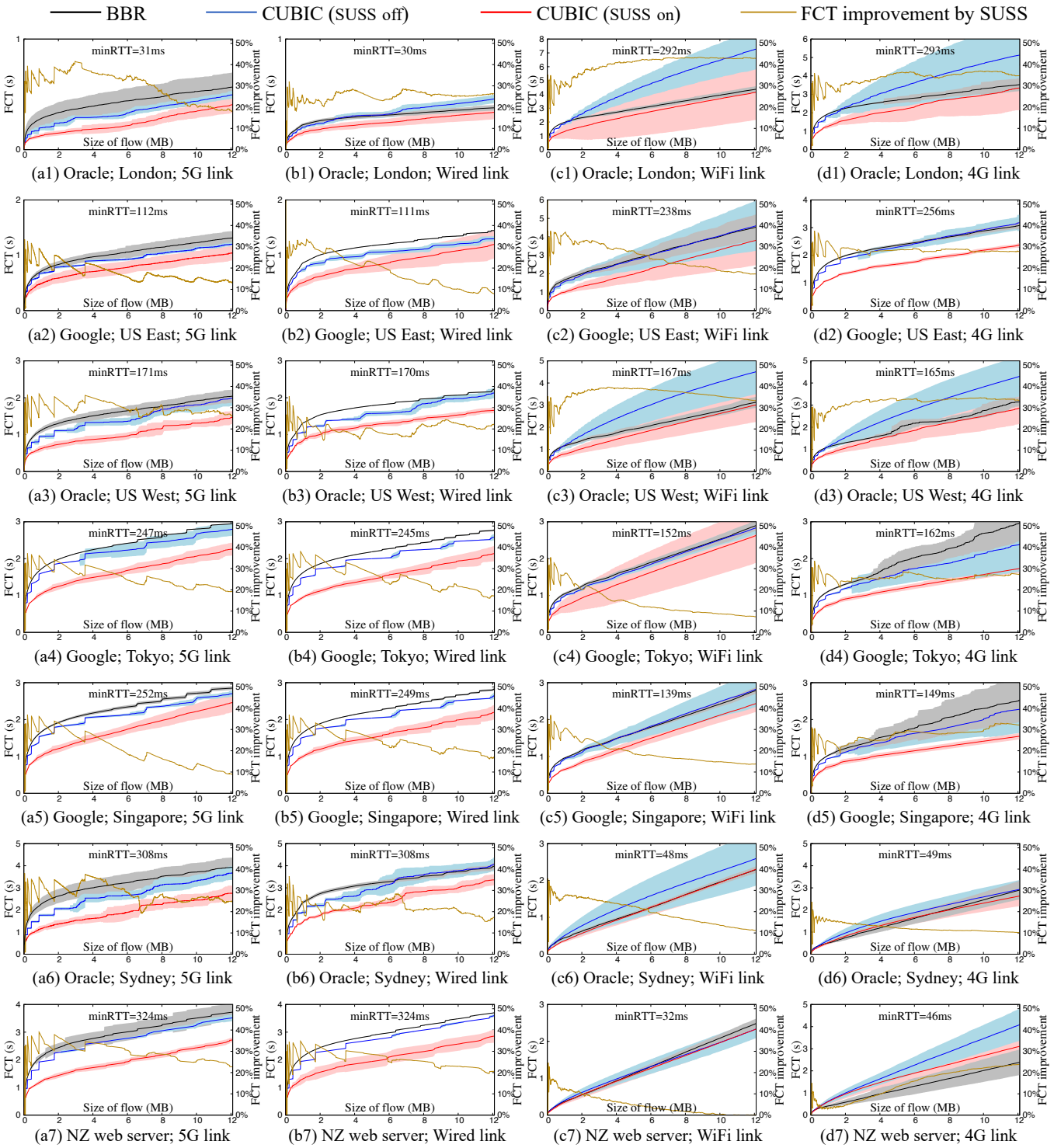


Figure 18: Comparative analysis of FCT in BBR, CUBIC with SUSS on, and CUBIC with SUSS off. Each row corresponds to a specific server, while columns represent client link types. The location of the client for the first two columns (5G and Wired) is Sweden and for the last 2 columns (WiFi and 4G) is New Zealand (NZ).