R-TCP: A Framework to Optimize TCP Performance Over Rate-Limiting Networks

Shengtong Zhu[1], Yan Liu[2], Lingfeng Guo[2], Jack Y. B. Lee[1]
[1]*The Chinese University of Hong Kong* [2]*Independent Researcher*

Abstract

Many mobile operators provide subscription plans that include a data quota for full-speed access, beyond which the service will be throttled to a low data rate — rate-limited service. This is designed to control costs and to motivate users to upgrade. Our recent measurements in a country-scale service suggested that the proportion of TCP flows subjected to rate limiting can be as high as 28%. More importantly, TCP flows under rate limiting can exhibit excessive retransmission rates, exceeding 20% in many cases. The extra bandwidth costs incurred by the retransmissions for large service providers are very significant, not to mention bandwidth wastage. This work develops a novel R-TCP framework to mitigate the excessive retransmissions problems in various TCP designs (e.g., Cubic and BBR) under rate limiting networks. R-TCP is specifically designed and optimized for sender-side kernel implementation with minimal overheads. It has been implemented into Linux where extensive experiments in real-world networks and applications show that it can substantially reduce excessive retransmissions by up to 88% with negligible tradeoff in goodput and application-layer performance.

1 Introduction

Mobile operators commonly employ rate limiting as a tool to provision lower-cost subscription plans to end users [33]. For example, the operator may allow users full-speed access within a fixed monthly data quota (e.g., 5 GB), beyond which the maximum transfer rate will be artificially restricted to a low level (e.g., 1 Mbps). Another common subscription plan is to allow users unlimited data but always subject to a maximum transfer rate (e.g., 2 Mbps). Finally, some operators also employ rate limiting to offer unlimited data for streaming [21]. Rate limiting enables the operator to balance the cost of service provisioning against the subscription income, especially under 5G where its high bandwidth can consume large amount of network resources very quickly (e.g., at 400 Mbps, one can download 5-GB data in just 100 s).

Mobile rate limiting is often implemented using a token bucket mechanism [33] within the mobile operator's network. The token bucket is configured with two parameters: the token replenishment rate (R) and the token bucket size (B). R controls the long-term data rate while B controls the short-term data burst size at full speed. If packets arrive at the token bucket faster than R, then the excess packets will either be discarded (traffic policing) or queued (traffic shaping). Note that packets could still be dropped in the latter case once the buffer overflows.

Recent studies [21,32,33] revealed that mobile rate limiting can interact with the congestion control algorithm (CCA) inside existing transport protocols including TCP and QUIC, resulting in excessive retransmissions (over 20% in many cases) that incur significant bandwidth costs to the service provider and bandwidth wasted along the network path. This work takes it one step further by developing a new lightweight framework for TCP, called R-TCP, to detect and optimize TCP's performance in rate-limiting networks. Compared to previous work, the main contributions of this work are:

- R-TCP is designed to handle *multi-transfer flows* — use of a persistent TCP connection for transferring multiple data objects, a common technique employed by many applications (e.g., video streaming) which can render existing methods [18, 32, 33] ineffective.
- R-TCP supports *concurrent flows* — use of multiple concurrent TCP flows to transfer data simultaneously from a server, another technique employed by some applications (e.g., short-video service) which can degrade the performance of existing algorithms.
- Existing optimization algorithms in BBRv1 [15] and mBBR [32] were BBR-specific and cannot be applied to other CCAs, including Cubic, which is still widely used by service providers. R-TCP's detection algorithm is CCA-agnostic, as it does not depend on or make any assumption on the underlying CCA. Moreover, its optimization framework can be readily adapted to different CCAs, including Cubic, BBRv1, BBRv3, as well as learning-based ones. Extensive experiments demonstrate that R-TCP offers more robust performance than exist-

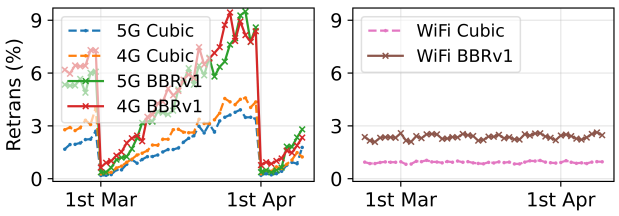

Figure 1: Daily retransmission (Retrans) percentages measured from a production short-video service.

ing solutions, and outperforms them across a wide range of network and application settings.

The rest of the paper is organized as follows: Section 2 introduces the background and reviews previous related works; Section 3 discusses the challenges; Section 4 and 5 present R-TCP's detection and optimization framework, respectively; Section 6 evaluates and compares R-TCP to existing solutions; and Section 7 summarizes the study and outlines some future research directions.

2 Background and Related Work

We first introduce the background to mobile network rate limiting and then give a brief survey of the existing literature.

2.1 Mobile Network Rate Limiting in Practice

Rate limiting can be easily spotted in the mobile data SIM's service specification. Zhu *et al.* [33] listed some 14 operators around the world as examples. An interesting question would be the extent of rate limiting being applied in practice.

To shed light on this question, we collaborated with a short-video service provider to measure the retransmission percentages at the transport protocol over a period of 46 days. We collected 6 million requests per day for short videos streamed over both WiFi and mobile networks. These requests were served using either BBRv1 [15] or Cubic [20] as TCP's CCA at the server side. A total of 30 servers in three different geographical regions were included in the study.

Fig. 1 plots the daily-averaged TCP retransmission percentages over the period for mobile networks and WiFi, respectively. The retransmission percentages for mobile networks exhibited a remarkable sawtooth pattern where the retransmission percentage progressively increased over the course of a calendar month until it reached the end of the month and then abruptly dropped to the lowest level again at the 1-st of the next month.

The measurements were conducted in a region where the mobile data quota are commonly reset according to the calendar month. The abrupt drop and steady increase in the retransmission percentage clearly correlate with the data quota reset cycle. As the month progresses, more users exhausted their data quotas and became subject to a rate-limited service. As a control, flows from WiFi did not show such a trend throughout the month. We note that Cubic exhibited lower retransmission percentages than BBR due to its sensitivity to packet losses (c.f. Section 3.1).

Using data from the study, we can apply a simple model to estimate the proportion of flows that are subject to rate limiting. Assuming no flow subjects to rate limiting on the 1st of the month. Let u be the average retransmission percentage for a TCP flow under rate limiting, and u_f and u_l be its overall retransmission percentages on the first and last day of the month, respectively. Let p be the proportion of rate-limited flows on the last day of the month. Then we have the relation:

$$up + (1-p)u_f = u_l, \tag{1}$$

where only u and p are unknown.

We obtained an estimate of u from another top-10 production short-video service that also employs BBRv1, configured to detect when a flow is rate-limited and to record its retransmission percentage. The results, measured over one month, show that rate-limited flows exhibited a mean retransmission percentage of $u = 29.6\%$. Hence, together with the measured u_f and u_l for BBRv1 on 1 Mar and 31 Mar, respectively, we can then solve for p using Eq. (1) to obtain the proportion of rate-limited flows, which amounts to 28% on 31 March. As large content providers easily consume bandwidth in Tbps, so even 1% additional retransmissions incur significant bandwidth costs. Moreover, as mobile rate limiting is used by operators for service (and thus price) differentiation purposes, it is likely to remain in the future.

2.2 Review of Previous Work

The research on network rate-limiting can be broadly categorized into two main areas: (i) measurement studies that examine the deployment and impact of network rate-limiting mechanisms; and (ii) detection and optimization for protocols and services subject to network rate-limiting.

Network rate limiting has been observed in practice by several studies, such as Flach *et al.* [18], Zhang *et al.* [31], Kakhki *et al.* [21], Lakshminarayanan and Padmanabhan [23], Dischinger *et al.* [17], Bauer *et al.* [14], Kanuparthy and Dovrolis [22]. As network rate limiting can degrade protocol and application performance, researchers first developed algorithms to detect the presence of rate limiting. This includes passive approaches (via observing flow analytics) such as PD [18], BBRv1's inbuilt detector [15], mBBR [32], and MODRL [33], as well as active approaches (via explicit probing) such as Shaperprobe [22], NarrowTokenRate [29].

More importantly, the goal to detect rate limiting is to apply the detection results towards optimizing protocols and application performance. For example, BBRv1 and mBBR [32] applied it to optimize the performance of TCP-BBR. Due

Table 1: Properties of mobile networks tested. Op 1 to 3 are rate limiting while Op 4 and Op 5 are non-rate-limiting.

	Op 1	Op 2	Op 3	Op 4	Op 5
Network type	4G	5G	5G	4G	4G
Mean RTT (ms)	36	18	21	45	36
Advertised R (Mbps)	2	1	1	–	–
Est. B Median (KB)	733	385	223	–	–
Est. B Std (KB)	1.42	61.03	27.05	–	–
Est. R Median (Mbps)	2.07	1.3	1.04	–	–
Est. R Std (Mbps)	0.0004	0.049	0.0003	–	–

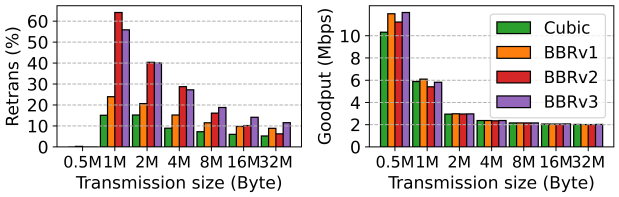

Figure 2: Retransmission percentages and goodputs under Op 1.

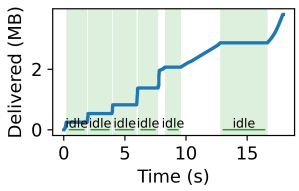

Figure 3: Multi-transfer flow used by the X app in playing videos.

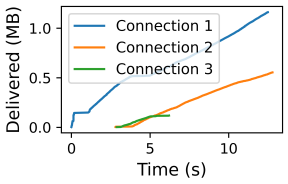

Figure 4: Concurrent flows used by the Douyin app in playing video.

to space limitation, the interested readers are referred to the respective studies for more details.

3 Challenges

We first demonstrate TCP's excessive retransmission problem under rate-limiting networks in Section 3.1 and then illustrate the limitations of existing solutions under multi-transfer scenarios in Section 3.2 and concurrent flow scenarios in Section 3.3. The experiments were conducted using a stationary desktop connected to mobile operator (Op) #1 (c.f. Table 1) via a 5G modem, downloading data from a server with a high-speed connection to the local Internet exchange where the mobile operator also peers with. Both the client and server run Linux 5.4 equipped with TCP Cubic [20], BBRv1 [15], BBRv2 [3], and BBRv3 [16].

3.1 Single-Transfer Flows

We begin with the simplest data transfer use case — downloading a file over non-persistent HTTP, with 15-s idle period between successive downloads. Fig. 2 compares the retransmission percentages for file sizes from 0.5 MB to 32 MB, averaged over 30 runs for each configuration. Note that the retransmission percentages in Fig. 2 are much higher than those in Fig. 1, because the latter is averaged across both rate-limited and non-rate-limited flows. We observe that retransmission is negligible when downloading 0.5-MB files, because Op 1's token bucket size (estimated to be around 730 KB) is larger than the file size. The 15-s idle time fills the bucket with tokens so that rate limiting is not activated at all during transfer (hence the high goodput).

For files of 1 MB and larger, TCP exhibited very high retransmission rates, exceeding 50% in the cases of BBRv2 and BBRv3 at 1-MB file size, and decreased with larger file sizes. We note that BBRv1 exhibited markedly lower retransmission percentages than BBRv2 / v3 in most cases, presumably due to BBRv1's built-in rate-limiting detector and optimizer [32] which were not presence in BBRv2 / v3. The exact reasons are more complicated and are explained in Appendix A.

Interestingly, Cubic exhibited the lowest retransmission percentages due to its higher sensitivity to packet loss that triggers reduction in its CWnd. This suggests a simple optimization strategy — switch CCA to Cubic whenever a flow is detected to be under rate limiting. However, there is still much room for improvement, as Cubic's retransmission percentages under rate limiting are still much higher than normal (e.g., 15% at 1-MB file size).

Despite the high retransmission percentages, TCP's goodput is not degraded significantly as evident in Fig. 2. This is because lost packets were primarily discarded by the rate limiter *before* they reach the mobile link, thereby wasting bandwidth only in the wired network path but not the wireless network path. The rate limiter's low data rate resulted in very small BDP (e.g., Op 1's BDP ∼10 KB) so that even TCP sender's reduced CWnd under congestion is still sufficient to keep the mobile link fully utilized (see Appendix B for additional results with larger BDPs). Therefore, the challenge is to reduce TCP's excessive retransmission percentages without degrading its goodput performance.

3.2 Multi-Transfer Flows

Many applications employ persistent connection to request and download multiple data objects over a single TCP connection [9]. This is illustrated in Fig. 3 which plots the data transfer pattern over time, when playing videos using the X mobile app [13]. The app uses a single TCP connection to transfer multiple data objects from the server. Successive data object transfers are separated by some idle time.

To assess how well existing solutions work under such multi-transfer scenarios, we extracted the data object sizes

and the idle time between them and then developed a client application to recreate the multi-transfer pattern by downloading files from an Apache 2.4.29 server using a persistent HTTP connection.

We applied three existing rate-limiter detection algorithms — PD [18], mBBR [32], and P-MODRL [33], to the traffic trace to detect and estimate rate-limiter parameters. Note that only mBBR operated online, while PD and P-MODRL analyzed BBRv1's traffic trace generated by the client application. All three algorithms failed to detect rate limiting in this experiment, because they did not account for tokens accumulated during the idle periods. They may work under different multi-transfer settings, but even then the accuracy of the estimated rate limiter parameters can be significantly impaired.

3.3 Concurrent Transfer

In addition to multi-transfer flows, some applications also transfer data using multiple TCP connections simultaneously. This is illustrated in Fig. 4 for the Douyin mobile app [8], where up to three connections are active simultaneously during video streaming. This concurrent transfer scenario can impact rate limiting detection and optimization.

For example, we replicated and tested the three existing algorithms under the concurrent-transfer pattern in Fig. 4 using the method described in Section 3.2. PD and P-MODRL were both unable to detect rate limiting, while mBBR detected rate limiting in only one of the three connections, with an estimated rate limit 64% lower than the reference value in Table 1.

To tackle these challenges, we present in the next two sections a new rate limiting detection and optimization framework which is also memory and CPU efficient, thereby facilitating its implementation inside the kernel TCP module.

4 R-TCP Rate Limiting Detection

We develop in this section a new rate-limiting detection algorithm for R-TCP to tackle the challenges discussed in Section 3.2 and 3.3. It comprises three phases: (i) a detection trigger to initiate subsequent phases only when rate limiting is likely presence (Section 4.1); (ii) a detection phase where R-TCP continuously estimates the rate limiter parameters and classifies the presence of rate limiting (Sections 4.2 and 4.3); and (iii) a post-detection phase where R-TCP looks for signs of false positives or lifting of the rate limiting condition (Section 4.4). See Appendix C for a summary of notations used throughout this section.

To facilitate integration with TCP in the kernel, R-TCP is designed to be *light weight* with optimized kernel memory and CPU utilization. It is a passive detector which only makes use of flow analytics, such as ACK and TCP internal states, to carry out rate limiter parameter estimation and classification. It is *CCA-agnostic* and so can be applied to different types of CCAs such as Cubic and BBR.

4.1 Detection Trigger

The most significant impact of rate limiting on TCP is the bandwidth costs incurred by excessive retransmissions (e.g., 64% for a 1-MB flow) [18, 31–33]. This motivates the use of packet loss to trigger rate limiting detection. Let l and d be the number of packets which are lost and delivered, respectively, during an interval of α minimum-RTT (min-RTT) rounds that begins with the first packet loss. Calculations for rate limiter parameter estimation and detection will only begin, if the packet loss rate exceeds a set threshold β in the past α rounds,

$$l/(d+l) > \beta. \tag{2}$$

Otherwise, it waits for the next packet loss to begin a new cycle of α min-RTT rounds to check Eq. (2) again[1].

As flows not subject to rate limiting are likely to exhibit much lower packet loss rates, they will not trigger Eq. (2) and so will not incur any further processing overheads. For example, we measured the packet loss rate (with α=7) in mobile network Op 1 (c.f. Table 1) downloading a 32-MB file. When Eq. (2) triggers, the packet loss rate in the detection cycle were 80.5% (BBRv1) and 72.3% (Cubic), respectively, averaged over 30 runs. These loss rates are much higher than the mean retransmission rates (at 9% for BBRv1 and 5% for Cubic) as the initial token depletion caused more severe packet losses (see Appendix A). We also measured the packet loss rates in Op 2 and Op 3, both of which are above 40% for Cubic and BBRv1. This allows one to choose a higher loss threshold (e.g., $\beta = 0.2$) to reduce false positives for non-rate-limiting flows. Please refer to Appendix I for sensitivity analysis of the detection phase hyper-parameters.

4.2 Parameter Estimation

Once triggered, R-TCP performs two tasks for each ACK packet received: (i) estimate the rate limiter parameters; (ii) classify if the flow is under rate limiting. We cover the first task below and the second task in Section 4.3.

Multi-Transfer Flows — In a multi-transfer flow, data are transferred in chunks separated by idle periods, e.g., DASH streaming [7] transfers video data in chunks using a separate HTTP transaction for each chunk over a persistent TCP connection. This presents a problem as tokens *accumulate* during the idle time. Consequently, when the next chunk transfer begins, the flow will *not* be rate limited until all accumulated tokens in the bucket are exhausted. This can cause existing algorithms [18, 32, 33] to fail detecting rate limiting, as they do not account for multi-transfer scenarios.

[1] We do not employ a sliding window here to eliminate the need for storing historical loss data inside the kernel.

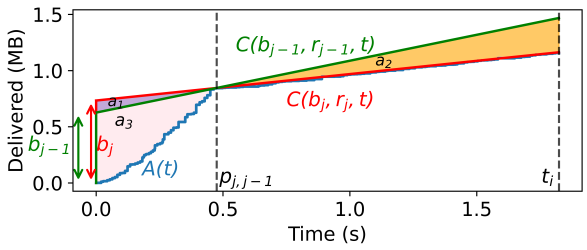

Figure 5: Given two tuples with candidate bucket size index j and $j-1$, to find the one with a smaller area gap according to Eq. (5), we only need to compare the areas of the two triangles a_1 and a_2 (c.f. Appendix D).

To tackle this challenge, R-TCP distinguishes each data transfer using Linux kernel's built-in application-limited detector [4] and then performs detection for each transfer independently. Specifically, Linux classifies a TCP flow to be in the application-limited phase when: (i) there is no packet waiting to be sent out; (ii) transmission is not limited by the send window; and (iii) all lost packets have been retransmitted. These three conditions are satisfied when a data transfer is completed, and the flow becomes idle. R-TCP uses the *exit* from the application-limited phase to detect the beginning of a new transfer to initiate rate-limiting detection.

Model Fitting and Memory Overheads — As R-TCP processes each transfer independently, we focus on a single transfer in the following discussions. Let $t = 0$ be the beginning of the current data transfer. Then the maximum amount of data that can be delivered by a rate limiter with parameters $\{B,R\}$ is given by:

$$C(B,R,t) = B + Rt, \tag{3}$$

where B is the token bucket size and R is the token replenishment rate (see Fig. 5).

Note Eq. (3) assumes the token bucket is full at $t = 0$. This may not always be true, especially in multi-transfer scenario, where the inter-transfer idle time may be too short to completely fill the bucket. To compensate for this potential inaccuracy, the estimated bucket size of a flow is taken as the maximum estimated bucket size across all transfers that passed the classification criteria (c.f. Section 4.3).

Let $A(t)$ be the cumulative amount of data acknowledged by the latest ACK packet at time t. If the flow is constrained by the rate limiter, then $A(t)$ must be upper limited by $C(B,R,t)$:

$$A(t_i) \leq C(B,R,t_i), \tag{4}$$

where t_i is the receiving time of ACK packet $i (i = 0,1,2\ldots)$ of the current transfer. Assuming the rate limiter is the bottleneck of the flow's throughput, then finding the rate limiter parameters is equivalent to finding the $\{B,R\}$ tuple that most closely matches $A(t)$ — measured by the gap (i.e., area) between the two curves $A(t)$ and $C(B,R,t)$, from:

$$\Delta(t) = \int_0^t (C(B,R,x) - A(x))\,dx. \tag{5}$$

Note that direct calculation of Eq. (5) needs $A(x) \mid \forall x \in (0,t)$, which can be calculated from the sequence number acknowledged and timestamp of all ACKs received since the beginning of the current transfer. However, this presents a practical problem for implementation inside the kernel.

For example, according to a major short-video service provider, the typical goodput on a 32-core server is around 5 Gbps. At full load with video bitrate of 600 Kbps [30], each server can serve over 8,000 concurrent connections. The average application session, e.g., on TikTok, lasts 5.93 minutes [5], downloading 25 MB and generating about 18,000 ACK packets. Storing data for each ACK requires 8 bytes of kernel memory, comprising 4 bytes for the sequence number and 4 bytes for the timestamp. For 8,000 flows, this could consume up to 1.1 GB of kernel memory, which is impractical. To tackle this challenge, we present below a new model-fitting method that eliminates the need for past ACK information.

Model Fitting by Successive Approximations — The first technique is to store only the intermediate calculated results for a small set of candidate values for the bucket size B so that they can be reused in subsequent rounds without referencing past ACK information.

We first need to determine the limits of the bucket size range. The obvious lower limit is $B_{min} = 0$. For the upper limit, we observe that by the time detection is triggered by Eq. (2), high packet losses have already occurred due to token bucket depletion. Assuming the bucket is full at the beginning of a transfer, then the amount of data transferred up to the detection triggering point must have exceeded the bucket size. Let h be the ACK number received at time t_h, which represents the highest ACK number before detection is triggered. Then, the total amount of data transferred, given by $A(t_h)$ becomes the upper limit for B:

$$B_{\max} = A(t_h) > B. \tag{6}$$

We choose the set of B candidates b_j to be ω (e.g., 9) equally spaced values within the range $\{B_{min}, B_{max}\}$, i.e.,

$$b_j = \left\{ B_{\min} + j\left(\frac{B_{\max} - B_{\min}}{\omega - 1}\right) \mid j = 0,1,\ldots,\omega-1 \right\}. \tag{7}$$

As a result, the estimation of B can result in the worst-case error of the step size Ω, given by:

$$\Omega = \frac{B_{\max} - B_{\min}}{\omega - 1}, \tag{8}$$

which is **Approximation #1**. The choice of ω trades parameter estimation accuracy against storage and CPU overheads.

Now, instead of storing all past ACK information, the system only updates and stores the latest estimated rate limit for each of the ω candidate bucket sizes, denoted by r_j for bucket size b_j, whenever an ACK is received, say at time t_i. Specifically, for a candidate bucket size b_j, we can simply choose the lowest rate that satisfies the constraint in Eq. (4) to minimize the area gap in Eq. (5):

$$\begin{aligned} C(b_j, r_j, t_i) &\geq A(t_i), \\ b_j + r_j t_i &\geq A(t_i), \\ r_j &\geq \frac{A(t_i) - b_j}{t_i}. \end{aligned} \tag{9}$$

In addition, to ensure the newly calculated rate from Eq. (9) for the current ACK also satisfies Eq. (4) for the previous ACKs, we update it only if it is larger than the current value:

$$r_j = \max\left(\frac{A(t_i) - b_j}{t_i}, r_j\right). \tag{10}$$

This is done for all ω candidate bucket sizes every time an ACK is received, resulting in a set of ω tuples $\{b_j, r_j\}$ that need to be stored. Using the previous short-video server example with $\omega = 9$ and 8 bytes for storing each tuple, 8,000 connections will consume only 0.6 MB of kernel memory.

Note that rate estimation begins only from ACK h when the detection is triggered by Eq. (2) (c.f. Section 4.1). Since past ACK information is not stored and there are no rate estimates for ACKs $i < h$, the estimated rate from Eq. (9) is not guaranteed to satisfy the constraint in Eq. (4) for ACKs 0 to $h-1$. This is **Approximation #2** which trades detection accuracy for reduced kernel memory and CPU overheads.

The next step is to determine the best-fitting tuple from $\{b_j, r_j\} | j = 0, 1, ..., \omega - 1$, that minimizes the gap area in Eq. (5). Consider the $C(\cdot)$ curves for two tuples $\{b_{j-1}, r_{j-1}\}$ and $\{b_j, r_j\}$ in Fig. 5, with the latest ACK packet i received at time t_i. They intersect at time $p_{j,j-1}$ given by:

$$p_{j,j-1} = \frac{b_j - b_{j-1}}{r_{j-1} - r_j}. \tag{11}$$

Theorem 1 below proves a computationally efficient condition for tuple $\{b_{j-1}, r_{j-1}\}$ to be a better fit than tuple $\{b_j, r_j\}$.

Theorem 1. *If $t_i < 2p_{j,j-1}$, then tuple $\{b_{j-1}, r_{j-1}\}$ is a better fit than tuple $\{b_j, r_j\}$.*

Proof. The proof is presented in Appendix D. □

Theorem 1 allows us to simply compare t_i and $p_{j,j-1}$ to find the better of any two tuples. Beginning with the tuple with the largest bucket size, i.e., $\{b_{\omega-1}, r_{\omega-1}\}$, the system first compares it against the next smaller one $\{b_{\omega-2}, r_{\omega-2}\}$. If Theorem 1 is satisfied, then it proceeds to compare $\{b_{\omega-2}, r_{\omega-2}\}$ and $\{b_{\omega-3}, r_{\omega-3}\}$, and so on, until Theorem 1 is no longer satisfied. In that case, the current tuple will be the best fit, established in Theorem 2 below.

Theorem 2. *Given three tuples $\{b_j, r_j\}$, $\{b_{j-1}, r_{j-1}\}$, and $\{b_x, r_x\}$ where $x < j-1$ and $j > 1$. If $\{b_j, r_j\}$ is a better fit than $\{b_{j-1}, r_{j-1}\}$, then $\{b_{j-1}, r_{j-1}\}$ will also be a better fit than $\{b_x, r_x\}$.*

Proof. The proof is presented in Appendix E. □

Theorem 2 further reduces computation, as rate estimation can stop once Theorem 1 is not satisfied for the first time.

Bucket Size Range Compensation — In our experiments, we observed some cases where the B_{max} set by Eq. (6) can be smaller than the ground truth. Further analysis revealed that this is due to packet losses not caused by the rate limiter (because its token bucket has not yet been depleted), but due to other factors such as sudden network condition deterioration.

If the losses are sufficiently large to exceed the detection trigger threshold β in Eq. (2), then B_{max} will be set prematurely to a value smaller than the actual token bucket size. To mitigate this potential error, if the calculated best-fit bucket size at ACK i is already at the upper limit $b_{\omega-1} = B_{max}$, then the system will extend the set of bucket size candidates by one step size Ω in Eq. (8) via adding $b_\omega = b_{\omega-1} + \Omega$. It will also remove the smallest bucket size candidate, b_0 in this example, to keep the total number of candidates (and hence kernel memory requirement) the same.

As b_ω is new, no prior rate estimate, i.e., r_ω, is available and hence its initial value is directly computed from Eq. (9). This is **Approximation #3**, as it cannot guarantee complying with Eq. (4) for the previous ACKs. The system then reapplies Theorem 1 to check if $\{b_\omega, r_\omega\}$ becomes the best-fit tuple, and if so, repeat the upper limit extension process until the best-fit bucket size is no longer at the upper limit.

Together, the three approximations significantly reduce kernel memory and CPU overheads with negligible tradeoffs in detection performance.

4.3 Classification

After detection is triggered by Eq. (2), the system will attempt to classify if the flow is under rate limiting whenever an ACK is received, using the best-fit $\{B, R\}$ tuple obtained from the parameter estimation process described earlier. Once a flow is classified as under rate limiting, the rate-limiting optimization framework in Section 5 will be applied. Note that parameter estimation will continue to refine the estimates until the current transfer ends.

For a flow to be classified as rate limiting, two conditions must be satisfied simultaneously:

Abrupt rate reduction — When the accumulated tokens in the token bucket are exhausted, the output data rate of the limiter will be clamped immediately to the rate limit. The system detects this abrupt rate reduction by comparing the flow's mean throughput *before* detection triggers by ACK h at time t_h, denoted by R_h:

$$R_h = A(t_h)/t_h, \tag{12}$$

to the estimated rate limit R. If the rate limiter is the bottleneck, then R_h must be larger than R. Let θ be the abrupt rate reduction threshold where a rate-limiting flow must satisfy:

$$R/R_h < \theta. \tag{13}$$

The choice of θ (e.g., 0.6) trades false positives against false negatives. Note that the threshold also constrains the set of feasible tuples $\{B,R\}$, so it can be exploited to further narrow the range of bucket size candidates to save computation (c.f. Appendix F).

Convergence — If the flow is under rate limiting, then $A(t)$ should converge to $C(t)$ for $t > t_h$, implying the estimated tuple $\{B,R\}$ should remain unchanged. Therefore, to pass the convergence test, the estimated $\{B,R\}$ from all ACKs received in a convergence window of ε (e.g., 10) consecutive min-RTT rounds must remain unchanged. This parameter trades longer detection time to reduce false positives.

The two conditions above are designed to reduce false positives in case the detection trigger in Eq. (2) is inadvertently triggered by packet losses not caused by rate limiting, e.g., random packet losses. In this case, the flow will be less likely to satisfy the abrupt rate reduction condition, and the resultant throughput will also fluctuate (as opposed to fixed by a rate limiter), failing the convergence test as well. As a further safeguard, R-TCP includes an un-detecting mechanism (c.f. Section 4.4) to revert false positive detections.

In multi-transfer flows, both parameter estimation and classification will continue to be performed for subsequent transfers wherever the detection threshold in Eq. (2) triggers. If the classification succeeds again in a subsequent transfer, then the system will update the token bucket size to be the maximum of the old and the new one, and also record the highest R_h for un-detection purposes as explained next.

4.4 Practical Considerations

We consider four practical issues in actual implementation and deployments in this section.

Concurrent flows to the same client — The detection algorithm discussed thus far works for a single TCP flow. In some applications (c.f. Section 3.3), the client may maintain two or more TCP connections to the server for data transfer. In particular, if the server sends data to a rate-limited client using multiple concurrent flows, then these flows will share the rate limit and thus impact rate-limiting detection accuracy.

To tackle this problem, R-TCP implements a hash table to store one set of data for each client (as opposed to each flow) based on its IP address. A reference counter keeps track of the number of flows *not* under the application-limited state (i.e., actively sending data). When a flow exits the application-limited phase and if the counter changes from 0 to 1, then a new transfer begins with all per-client data reset.

Otherwise (counter value larger than 1), more than one flow is actively sending data to the same client. In this case, the ACK information from all active flows will be merged (i.e., treating all ACKs as if they were the same flow) into one set of rate-limiter data stored into the client's hash table entry. Detection is then performed in the same way as discussed earlier using the merged data in the hash table.

BBRv1's internal rate limit detector — BBRv1 has an inbuilt rate limiting detector and optimizer [32]. Our experiments show that it may interfere with R-TCP's parameter estimation process. Specifically, in some cases BBRv1 may significantly underestimate the rate limit temporary (details presented in Appendix G), causing it to set its pacing rate to the underestimated rate limit for 48 RTTs. This will in turn cause R-TCP to underestimate the rate limit as well.

Therefore, R-TCP will reset the state of BBRv1's inbuilt detector when the abrupt rate reduction condition Eq. (13) is met for the first time. This prevents BBRv1's detector from restricting the pacing rate for 48 RTTs which can impair R-TCP's own rate estimation. After R-TCP detects rate limiting, BBRv1's inbuilt detector and optimizer will be disabled for the rest of the flow.

Un-detecting rate limiting — In case of false positive, a flow's throughput will be restricted by the optimizer (c.f. Section 5) to an incorrectly estimated rate limit, preventing it from fully utilizing the available bandwidth. Same for the cases where rate limiting is lifted in the middle of a flow, e.g., due to data quota reset or top-up. Therefore, it is desirable to have a mechanism to detect these conditions.

Specifically, if a rate-limited flow becomes unlimited then its transmission rate will increase, causing the continuously estimated R to increase as well. Eventually, R will approach R_h, thereby invalidating the abrupt rate reduction condition in Eq. (13). When this happens, the flow will be reclassified as non-rate-limiting and reverted back to normal non-rate-limiting operation (see Appendix H). Detection will not be attempted again for this flow.

Scalability — Real-world production servers often serve thousands of concurrent connections, so memory and CPU overheads must be kept within practical levels. With the approximations presented in the previous sections, our stress test using a server with 8K concurrent connections showed that R-TCP consumes only 2 MB kernel memory, with negligible increase in CPU utilization (see Appendix N).

5 R-TCP Performance Optimization

The goal of detecting rate limiting is to optimize TCP's operations to improve its performance by mitigating excessive retransmissions without degrading its goodput. While the optimizations are designed in conjunction with the CCA employed, the principle is the same — regulate data transmissions to keep it within the confine of the rate limiter, i.e., $C(\cdot)$. This can be realized via controlling the sending window (e.g., CWnd) and / or the sending rate (e.g., pacing rate). This section presents optimization algorithms for Cubic and BBRv1 /

BBRv3[2], and the reader is referred to Appendix M for another application to a learning-based CCA.

5.1 TCP-Cubic Optimization

Cubic regulates sending rate via the sending window and ACK clocking [20]. Therefore, instead of controlling the sending rate directly, R-TCP-Cubic constrains the inflight packets to one BDP by capping the congestion window (CWnd):

$$CWnd = \min(CWnd, R \cdot sRTT), \tag{14}$$

where $sRTT$ is the smoothed RTT and R is the estimated rate limit.

To cater to potential detection errors or rate limiter changes, once every η (e.g., 20) $sRTT$ rounds, the CWnd cap is increased by γ (e.g., 20%) for three rounds:

$$CWnd = \min(CWnd, (1+\gamma)R \cdot sRTT). \tag{15}$$

During these three rounds, if the estimated rate limiter parameters $\{B, R\}$ remain unchanged, then the original CWnd cap in Eq. (14) is reapplied. Otherwise, it suggests that either the detection results were inaccurate or the rate limiter has changed. In this case, the upper limit is lifted, allowing CWnd to grow according to the original Cubic algorithm, while it continues to update its $\{B, R\}$ estimates. Once $\{B, R\}$ remains unchanged for three consecutive rounds, then it will reactivate the CWnd cap in Eq. (14).

Capping the sending window to one BDP based on the estimated rate limit prevents Cubic's continuous CWnd growth to overshoot and cause packet losses at the rate limiter. Note that if the return path aggregates or delays ACKs then the sender may run out of available CWnd to transmit. However, the idling rate limiter simply stores the unused tokens in its bucket for use when the delayed ACKs return to the sender. The experimental results in Section 6 confirmed that R-TCP-Cubic can fully utilize the available bandwidth in rate-limiting mobile networks where ACK aggregation is common.

5.2 TCP-BBR Optimization

The BBR CCA family employs pacing to regulate the transmission rate directly. Let r be the original pacing rate calculated by BBR from its own bandwidth estimation. R-TCP simply clamps the applied pacing rate, denoted by r^*, to the estimated rate limit R:

$$r^* = \min(r, R). \tag{16}$$

Similarly, to cater to potential detection errors or rate limiter changes, once every η rounds, the upper limit R is increased by γ for three rounds:

$$r^* = \min(r, (1+\gamma)R). \tag{17}$$

[2]While R-TCP can also be applied to BBRv2, the latter has been obsoleted by BBRv3, so it is not included here.

During these three rounds, if the estimated $\{B, R\}$ remains the same, then the pacing rate cap in Eq. (16) is reapplied. Otherwise, it lifts the pacing rate cap and let BBR enter its probing phase to probe for bandwidth normally, while it continues to update its $\{B, R\}$ estimates. Once the estimated $\{B, R\}$ remains unchanged for three consecutive rounds, it reapplies the cap in Eq. (16) using the new R estimate. Sensitivity analysis of $\{\eta, \gamma\}$ can be found in Appendix J.

5.3 Multi-Transfer Flows

The above designs for Cubic and BBR do not take multi-transfer flow into consideration. They may still work, but may become sub-optimal in a multi-transfer flow. Specifically, in the idle time leading to a new transfer, tokens would have been accumulated in the token bucket. If the pacing rate in BBR is clamped to the rate limit R, then these accumulated tokens will not be utilized, thereby underutilizing the link.

To address this limitation, R-TCP continuously estimates the token bucket level and uses that information to fine-tune the transmission regulation. Let e_i^k be the estimated number of accumulated tokens at time t_i, when ACK i is received during transfer k where the flow is detected to be rate limited. Then the token bucket level is simply the difference between the amount of data allowed by the rate limiter $C(B, R, t_i)$ and the actual amount transferred $A(t_i)$:

$$e_i^k = B + Rt_i - A(t_i). \tag{18}$$

Let e_{last}^k be the token bucket level when the last ACK of transfer k is received. Then the token bucket level at the beginning of the next transfer $k+1$ can be estimated from

$$e_0^{k+1} = \min(e_{last}^k + Rt_{idle}, B), \tag{19}$$

where t_{idle} is calculated as the duration it spends in the application-limited phase.

In other words, transfer $k+1$ begins with e_0^{k+1} tokens already in its token bucket. Subsequent token bucket level can then be estimated from

$$e_i^{k+1} = \min\left(e_0^{k+1} - A(t_i) + t_i R, B\right). \tag{20}$$

Armed with this information, R-TCP will not restrict the transmission (by Eq. (14) for Cubic and Eq. (16) for BBR) to allow the tokens to be utilized, provided that

$$e_i^{k+1} > R \cdot sRTT. \tag{21}$$

This is a conservative measure to allow unconstrained transmission, only when the token bucket level is high compared to the flow's BDP. As further protection, if subsequent packet loss rate exceeds the threshold in Eq. (2), then the rate limit cap will be reapplied immediately.

5.4 Concurrent Transfer

R-TCP can integrate ACK information from multiple flows to estimate the (shared) link's rate limiter parameters. However, if multiple flows transmit at the rate limit R simultaneously, then the combined data rate will exceed the link's rate limit, causing congestion and packet losses.

Intuitively, if there are g flows actively transmitting to the same client simultaneously, then each flow should be assigned a sending rate cap of R/g for fair bandwidth sharing. In practice, it is more complicated, as applications may only send a tiny amount of data (e.g., text) in a flow. While the small transfer takes much less than one sRTT time to transmit, it will reduce the sending rate cap for all other active flows until its ACK returns in one sRTT. This is an implementation limitation in Linux's TCP congestion control module, as R-TCP codes can only execute to update the sending rate cap upon ACK processing.

Therefore, R-TCP will count a flow as active only if the sum of unsent data, denoted by u, and packets inflight, denoted by v, is larger than one BDP at the reduced sending rate cap:

$$u + v \geq \frac{R}{g} \cdot sRTT. \tag{22}$$

6 Performance Evaluation

We implemented R-TCP into Cubic, BBRv1, and BBRv3, in Linux kernel 5.4 within its TCP congestion control module. The new modules, namely R-TCP-Cubic, R-TCP-BBRv1, and R-TCP-BBRv3, can be dynamically loaded / unloaded into / from the Linux kernel in the same way as standard modules. Apart from the TCP congestion control module, no other kernel modifications are needed[3].

The experiments were performed using three common types of applications, namely file download (single transfer), on-demand video streaming (multi-transfer), and a production short-video app (concurrent transfers), to assess and compare various algorithms' performance under real-world application settings. Table 1 lists the used production mobile services, and Table 2 lists the experiment setup and default values for the hyper-parameters (see Appendix I and J for sensitivity analysis). We focus on R-TCP's optimization performance in this section. Readers are referred to the appendices for its detection performance (Appendices K and L), application to learning-based CCA (Appendix M), memory / CPU overheads (Appendix N), and fairness (Appendix P).

6.1 File Download

The experiment setup consists of a stationary client host connected to the mobile network via a USB 5G modem, downloading a file using non-persistent HTTP from an Apache server. We measured the retransmission rate and goodput for file sizes ranging from 0.5 MB to 32 MB using three rate-limited mobile services. Note that with a rate limit of 1 Mbps, a 32-MB file would take at least 270 s to download. 40 experiment runs were conducted for each configuration with the results averaged and plotted in Fig. 6.

[3]The code is open-sourced at: https://github.com/mclab-cuhk/R-TCP.

Table 2: Experimental settings.

	Description	Value(s)
R-TCP	Parameters	$\alpha = 7, \beta = 0.2, \theta = 0.6, \omega = 9,$ $\varepsilon = 10, \eta = 20, \gamma = 20\%$
Server & Client	Hardware	Dell OptiPlex 7020 Plus
	Software	Ubuntu 18.04, Linux 5.4
File Download	File size	0.5, 1, 2, 4, 8, 16, 32 MB
DASH Streaming	Chunk length	4 s
	No. of chunks	12
Short-video	Viewing dur.	10 s
	No. of videos	60

Overall Performance - Overall, R-TCP substantially reduced the excessive retransmissions when applied to all three CCAs. At 32-MB file size and averaged over all three operators, R-TCP reduced the retransmission percentages of Cubic by 57%, BBRv1 by 80%, and BBRv3 by 76%. In terms of goodput, the R-TCP versions are all within 5% of their unmodified versions, so there is negligible tradeoff there.

Impact of File Size - As the file size decreases, the reduction in retransmission also decreases, because R-TCP takes a certain amount of data transfer to detect rate limiting and then perform optimization. The amount of data needed is most influenced by the token bucket size of the mobile service, as the rate limiter will not kick in until all tokens in the bucket are exhausted. This explains why R-TCP performed similarly to its unmodified counterpart at 1-MB file size in Op 1, as this service has an estimated token bucket size of over 700 KB. For the same reason, there was little to no retransmissions at 0.5-MB file size in Op 1 as the file size is smaller than the token bucket size so that it is not subject to rate limiting at all.

As file size increases, the retransmission rate generally decreases. This is because the initial packet losses upon token bucket exhaustion are much more severe than the losses caused by subsequent bandwidth probing after the CCA has exited the Slow-Start / Startup phase. For example, the proportion of packet retransmissions during Slow Start to total packet retransmissions for Cubic decreased from 73% at 0.5-MB file size to 11% at 32-MB file size in Op 3.

Impact of Mobile Operator — BBRv1 exhibited much more retransmission in Op 3 compared to Op 1 & 2. Its retransmission percentages for a 32-MB file in Op 3 is 23.2% versus 8.6% in Op 2 even though the two operators have similar rate limits (1.32 versus 1.04 Mbps). We instrumented BBRv1's internal rate limit estimates and found that (see Fig. 7) it overestimated the rate limit significantly more often

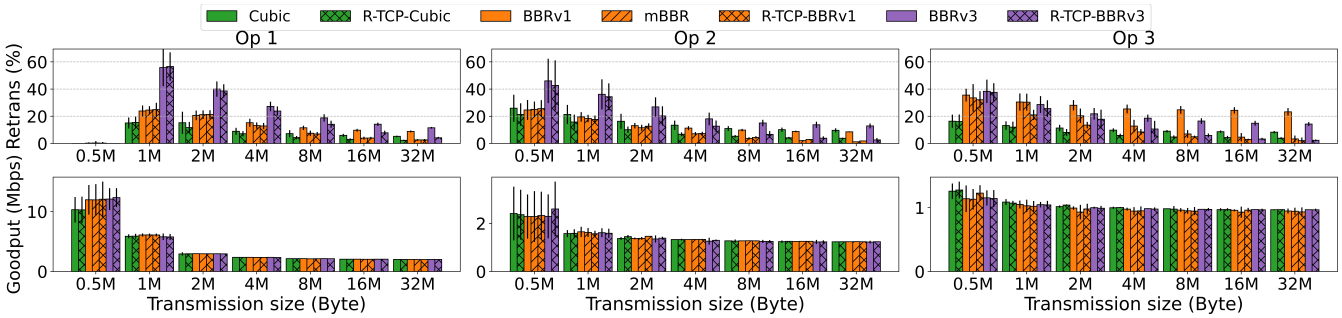

Figure 6: Compairons of retransmission percentages and goodputs for file download in rate-limiting mobile networks. Black lines represent standard deviation.

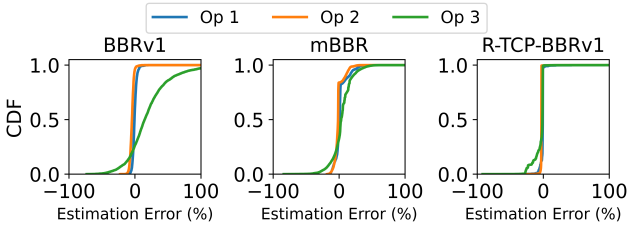

Figure 7: Distribution of rate limit estimation errors. R-TCP exhibited more consistent and accurate rate limit estimates.

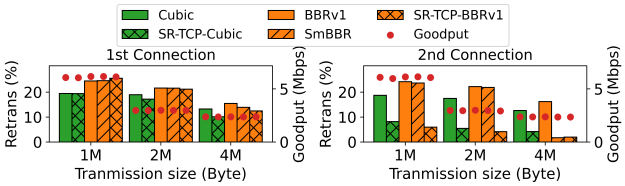

Figure 8: Impact of stateful optimization (in Op 1).

in Op 3, resulting in congestion losses even after applying optimization (with the inaccurate rate limit estimates).

mBBR achieved more accurate rate limit estimations in Op 3 as shown in Fig. 7, leading to less retransmission (e.g., reduce by 50% compared to BBRv1 at 4-MB file size). Finally, R-TCP-BBRv1's even more accurate rate estimations (Fig. 7) resulted in the lowest retransmission percentages (e.g., 8.4% versus 12.8% for mBBR at 4-MB file size).

In addition to rate-limiting mobile operators, we also tested R-TCP in Op 4 and Op 5, both of which are not rate-limiting. R-TCP did not detect rate limiting in all 40 experiment runs, i.e., no false positives, and simply ran the original CCAs.

Stateful Optimization — R-TCP can optimize a flow only after rate limiting is successfully detected. However, most applications initiate many TCP flows to the server over the course of a user session, so we can cache and use the detection results from a previous TCP flow to optimize subsequent TCP flows to the same client right from the beginning. This stateful optimization technique has also been adopted by mBBR (as SmBBR) [32] and others [10, 19].

We implemented stateful optimization for R-TCP, resulting in SR-TCP, by storing the detection results into the hash table employed in supporting concurrent transfers (c.f. Section 5.4). We only need to extend the token bucket level estimation method in multi-transfer optimization (c.f. Section 5.3) to multiple TCP flows.

We conducted an experiment to download two files of the same size using two separate TCP connections, with 5-s idle time in-between in Op 1. Each configuration is repeated 30 times with the results averaged and summarized in Fig. 8. SmBBR and stateful R-TCP (SR-TCP) both store the detection results from connection #1 to be reused by connection #2 to apply optimization right from the beginning of the flow.

We focus on the second connection where stateful optimization can be applied. Compared to the original CCA, R-TCP achieved significantly lower retransmission percentage (reduce by up to 88%) in the second connection, as it can apply optimization early on to prevent the initial burst loss due to token bucket exhaustion. SmBBR also benefits in the 4-MB file case, but not in the smaller file sizes, because it takes more than 2-MB data transfer to detect rate limiting.

6.2 Video Streaming

We used two video streaming applications to evaluate performance in multi-transfer flows. The first one is streaming videos within the X mobile app over a rate-limiting mobile network as described in Section 3.2. None of the existing algorithms can detect the rate limiting condition due to the idle times between transfers.

We applied R-TCP-BBRv1 to the same streaming sequence as shown in Fig. 9. It detected rate limiting at transfer 5 (k_4) where the estimated rate limit is within 2% of the actual one.

To enable quantitative performance comparisons, we setup

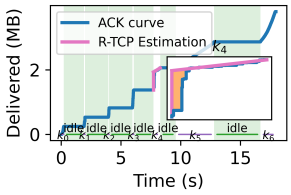

Figure 9: R-TCP detected rate limiting in transfer 5 under the X app.

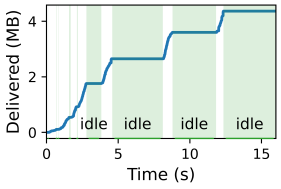

Figure 10: Multi-transfer pattern in DASH video streaming.

a second DASH-based streaming platform with sufficiently large client buffer to ensure that BBRv1 and mBBR can successfully detect rate limiting. We adopted the adaptive bitrate (ABR) algorithms implemented by Mao *et al*. [26] and dash.js [7], namely Pensive [26], BOLA [27], RobustMPC [28] and FastMPC [28]. The server runs Apache, delivering video data to the player running inside the Chrome browser. The test video (with 12 video chunks) and bitrate ladder followed those used in Mao *et al*. [26]. All experiments were repeated 15 times with the results averaged.

Fig. 10 plots the data transfer pattern in the DASH streaming platform. A key observation here is that the first few video chunks are delivered in almost back-to-back manner, because DASH requests the next video chunk immediately upon downloading the current one, as long as buffer space is available. This initial long period of continuous data transfer enables BBRv1 / mBBR to successfully detect rate limiting.

All ABR algorithms except BOLA transfer video chunks using a single persistent HTTP connection. BOLA may use multiple connections to transfer different video chunks but the transfers are in sequence, not concurrent. We enabled stateful optimization in both R-TCP and mBBR to enhance their performance under BOLA.

Fig. 11 compares the retransmission percentages and the streaming video's Quality-of-Experience (QoE) [26]. Overall, SR-TCP can reduce retransmission percentages by 28.4% (vs. Cubic), 39.9% (vs. BBRv1), 43.5% (vs. BBRv3), and 5.2% (vs. SmBBR) averaged over all configurations. SR-TCP's QoE performances are within 3% of its unmodified counterparts, so there is no noticeable tradeoff there.

On the one hand, the results clearly show that QoE performance is still primarily determined by the ABR algorithm. On the other hand, we observed that their QoE performances varied much more widely in Op 1 and 2 than in Op 3. We conjecture that this is due to interactions between the ABR algorithm and the rate limiter. Specifically, most ABR algorithms measure chunk download throughput to estimate available bandwidth to inform its bitrate choices for future chunks. At the beginning of a session, the rate limiter does not kick in until all accumulated tokens in the bucket are exhausted, which may mislead the ABR algorithm to estimate a bandwidth higher than the rate limit and thus choosing higher video bitrate version for the future chunks (e.g., FastMPC selected its 2nd chunk with bitrate at 4.3 Mbps, which is far higher than the rate limit of 2 Mbps in Op 1).

Once rate limiting kicks in, the much lower data rate combined with high-bitrate chunks will cause playback rebuffering to occur, degrading QoE. The extent of these interactions obviously depends on the design of the ABR algorithm and the rate limiter configuration. For example, the four ABR algorithms' achieved more similar QoE performance in Op 3, because the latter's rate limiter has a small bucket size (223 KB) compared to the total size of dash.js file plus the video chunk size (∼560 KB) so that the non-rate-limiting period is short and thus has much less impact on the ABR algorithms.

6.3 Short-video Applications

The third set of experiments aims at evaluating R-TCP performance in a real-world application. Ideally, deploying R-TCP into a production service's servers will provide the most definitive real-world performance results. The authors are working with a service provider to conduct field trials of R-TCP and hope to report such results in the future.

In the meantime, we employ a proxy-server setup depicted in Fig. 12 to apply R-TCP to a real-world short-video application without modifying the service provider's servers. We installed a mobile app called Super Proxy [12] into the Android smartphone. Once enabled, the client proxy app then redirects all network traffics to the Apache Proxy [6] running at the proxy server. The proxy server then forwards requests / responses between the smartphone app and the service provider's servers. We deployed R-TCP into the proxy server for delivering data over the mobile network to the smartphone.

The anonymous short-video app is among the top-10 video app worldwide. This short-video app exhibits complex network behavior, including the use of persistent connections for multi-transfer and the use of concurrent connections for simultaneous transfer as well. These short-video apps often dynamically preload images and videos, sometimes out-of-sequence [24] to optimize the user experience. Therefore it offers a good platform to not only evaluate R-TCP's performance, but also to uncover any potential limitations.

The short-video app's service provider provided the actual video playback statistics recorded by the app. We used Klick'r [11] to automate the click and swipe actions on the short-video application according to a set sequence. Each viewing session swiped and played 60 videos. Each experiment is repeated 30 times with the results averaged.

Table 3 summarizes the results for Op 1 and Op 2, respectively[4]. The results demonstrate that R-TCP can significantly

[4]Results for Op 3 are not available as there are compatibility issues between the proxy setup and Op 3, where connections were constantly dropped whenever the proxy is used, regardless of which CCA is in use.

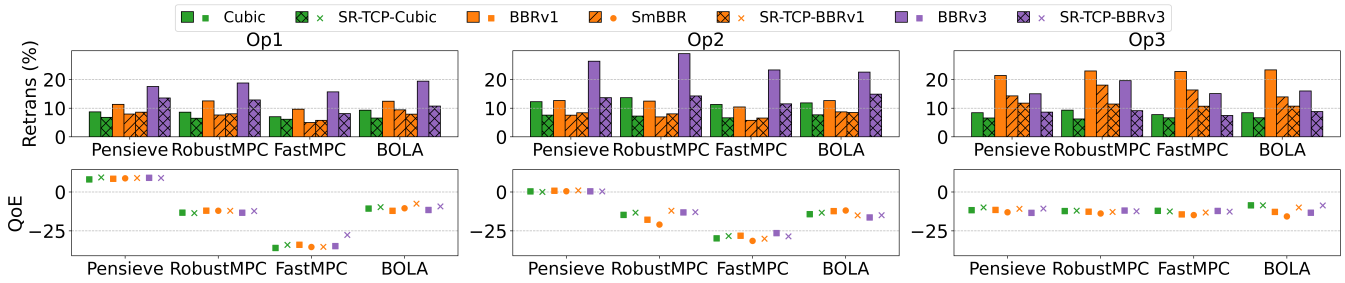

Figure 11: Comparison of retransmission percentages and QoEs for on-demand video streaming in rate-limiting mobile networks.

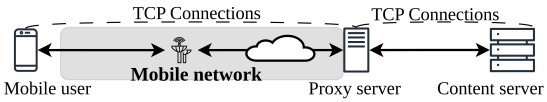

Figure 12: The setup for the short-video application.

Table 3: Performance for the short video application.

		Avg. Retrans. (%)	Rebuf. Count Ratio	Rebuf. Dur. Ratio	Avg. Bitrate (Kbps)
Op 1	BBRv1	13.5	2×10^{-7}	0.7×10^{-3}	572
	SmBBR	6.9	7×10^{-7}	2×10^{-3}	548
	SR-TCP-BBRv1	6.6	3×10^{-7}	0.5×10^{-3}	565
	Cubic	8.9	2×10^{-7}	4×10^{-4}	580
	SR-TCP-Cubic	5.3	8×10^{-8}	2×10^{-4}	584
Op 2	BBRv1	18.4	0.4×10^{-5}	1×10^{-2}	533
	SmBBR	7.3	1×10^{-5}	3×10^{-2}	542
	SR-TCP-BBRv1	7.3	0.4×10^{-5}	1×10^{-2}	537
	Cubic	12.8	4×10^{-6}	1×10^{-2}	536
	SR-TCP-Cubic	8.1	4×10^{-6}	1×10^{-2}	535

reduce the retransmission percentages by 40% and 60% compared to BBRv1 and Cubic, respectively. Similar reductions are also observed in applying R-TCP to BBRv3 (Appendix O). It is worth noting that although BBRv1 has an in-built rate limiting detector and optimizer, it is not very effective in this application as its retransmission percentages are higher than Cubic. Finally, SmBBR also achieved similar reductions in retransmission percentages under BBR, albeit with some tradeoffs in streaming performance.

Considering the rebuffering count / duration ratio — computed from the total number / duration of rebuffering events divided by the total viewing duration in milliseconds in a user session, SmBBR exhibited more than doubled the rebuffering count / duration compared to BBRv1 and R-TCP in both mobile networks, which is considered significant by our industry collaborator. We conjecture that the short-video app's complex network behavior combining multi-transfers and concurrent transfers caused SmBBR to over / under-estimate the rate-limiter parameters, which in turn degraded its rate-limiting optimization performance.

In comparison, streaming performance under R-TCP is similar to BBRv1 and Cubic. This demonstrates that R-TCP not only can reduce excessive retransmissions, saving total bandwidth usage by 13% and 24% compared to BBRv1 and BBRv3 (Appendix O), but does so without sacrificing streaming performance.

7 Summary and Future Work

This work shows that R-TCP can be an effective tool to optimize existing TCP CCA to mitigate their excessive retransmissions under rate-limiting networks without sacrificing goodput or application QoE performance. There are many fruitful directions for future work. The authors are actively working with large service providers to carry out large-scale evaluations of R-TCP in production services. The goal is to assess R-TCP's performance across different types of networks and applications, and to uncover any edge cases or unexpected behaviours in real-world environments.

A promising future research direction would be to apply R-TCP to other TCP designs. Moreover, the design principles in R-TCP is not limited to TCP either, and so applications to other transport protocols, QUIC in particular, would be of great interest to service providers. In addition, while server-side implementations are easier to be deployed by service providers, adapting R-TCP to the client side could benefit all kinds of services even without support from the service provider. This would be a challenging but fruitful topic for future research.

Acknowledgments

We thank our shepherd Professor Zeqi Lai and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by grants GRF14206521 and GRF14219022 from the HKSAR Research Grants Council.

References

[1] Dummynet. https://cs.baylor.edu/~donahoo/tools/dummy/. Accessed: 2025-07-22.

[2] Cisco business 250 series smart switches. https://www.cisco.com/c/en/us/support/switches/business-250-series-smart-switches/series.html, 2020. Accessed: 2025-07-22.

[3] BBR: Congestion Control Algorithm (v2alpha Branch). https://github.com/google/bbr/tree/v2alpha, 2021. Accessed: 2025-04-24.

[4] Linux Kernel: TCP Rate Control (tcp_rate.c). https://github.com/torvalds/linux/blob/059dd502b263d8a4e2a84809cf1068d6a3905e6f/net/ipv4/tcp_rate.c, 2023. Commit 059dd502b263, Accessed: 2025-04-24.

[5] Digital 2024 Deep Dive: The Time We Spend on Social Media. https://datareportal.com/reports/digital-2024-deep-dive-the-time-we-spend-on-social-media, 2024. Accessed: 2025-04-24.

[6] Apache HTTP Server Project. https://httpd.apache.org/, 2025. Accessed: 2025-04-24.

[7] dash.js: A JavaScript Library for MPEG-DASH Adaptive Streaming. https://github.com/Dash-Industry-Forum/dash.js, 2025. Accessed: 2025-04-24.

[8] Douyin: Short-Video Social Media Platform. https://www.douyin.com/, 2025. Accessed: 2025-04-24.

[9] HTTP persistent connection. https://en.wikipedia.org/wiki/HTTP_persistent_connection, 2025. Accessed: 2025-04-24.

[10] QUICHE: QUIC, HTTP/2, HTTP/3, and Related Protocols Implementation. https://github.com/google/quiche, 2025. Accessed: 2025-04-24.

[11] Smart AutoClicker: Screen-Based Automation Tool. https://play.google.com/store/apps/details?id=com.buzbuz.smartautoclicker&hl=en, 2025. Accessed: 2025-04-24.

[12] Super Proxy: HTTP and SOCKS5 Proxy Tunneling App. https://play.google.com/store/apps/details?id=com.scheler.superproxy&hl=en, 2025. Accessed: 2025-04-24.

[13] X: Social Media Platform. https://x.com/, 2025. Accessed: 2025-04-24.

[14] Steven Bauer, David Clark, and William Lehr. Powerboost. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Home Networks*, pages 7–12, 2011.

[15] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[16] Neal Cardwell, Yuchung Cheng, Kevin Yang, David Morley, Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Van Jacobson, Ian Swett, Bin Wu, et al. BBRv3: algorithm bug fixes and public internet deployment. In *Proc. IETF 117th Meeting*, pages 1–36, 2023.

[17] Marcel Dischinger, Andreas Haeberlen, Krishna P Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 43–56, 2007.

[18] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 468–482, 2016.

[19] Lingfeng Guo and Jack YB Lee. Stateful-tcp—a new approach to accelerate tcp slow-start. *IEEE Access*, 8:195955–195970, 2020.

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

[21] Arash Molavi Kakhki, Fangfan Li, David Choffnes, Ethan Katz-Bassett, and Alan Mislove. Bingeon under the microscope: Understanding t-mobiles zero-rating implementation. In *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 43–48, 2016.

[22] Partha Kanuparthy and Constantine Dovrolis. Shaperprobe: end-to-end detection of isp traffic shaping using active methods. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 473–482, 2011.

[23] Karthik Lakshminarayanan and Venkata N Padmanabhan. Some findings on the network performance of broadband hosts. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 45–50, 2003.

[24] Zhuqi Li, Yaxiong Xie, Ravi Netravali, and Kyle Jamieson. Dashlet: Taming swipe uncertainty for robust short video streaming. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1583–1599, 2023.

[25] Xudong Liao, Han Tian, Chaoliang Zeng, Xinchen Wan, and Kai Chen. Astraea: Towards fair and efficient learning-based congestion control. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 99–114, 2024.

[26] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[27] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM transactions on networking*, 28(4):1698–1711, 2020.

[28] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[29] Ertong Zhang and Lisong Xu. Capacity and token rate estimation for networks with token bucket shapers. *Computer Networks*, 88:1–11, 2015.

[30] Jupeng Zhang, Yan Liu, Jack YB Lee, and Shengtong Zhu. Congestion control optimization for short video services: User-end and edge server collaboration in practice. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2024.

[31] Yuming Zhang, Yan Liu, Lingfeng Guo, and Jack YB Lee. Measurement of a large-scale short-video service over mobile and wireless networks. *IEEE Transactions on Mobile Computing*, 22(6):3472–3488, 2022.

[32] Shengtong Zhu, Yan Liu, Lingfeng Guo, Rudolf KH Ngan, and Jack YB Lee. mBBR-Improving BBR performance over rate-limited mobile networks. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2023.

[33] Shengtong Zhu, Yan Liu, Lingfeng Guo, Yuming Zhang, and Jack YB Lee. On rate-limiting in mobile data networks. *IEEE Transactions on Mobile Computing*, 2024.

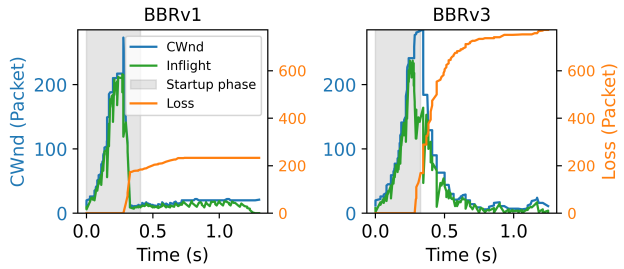

Figure 13: CWnd and packet loss evolution of BBRv1 and BBRv3 under Op 1. BBRv3's higher losses are due to its CWnd setting upon entering fast recovery.

A Analysis of BRBv1 and BBRv3 under Rate Limiting Networks

Fig. 13 plots the evolution of CWnd, packets inflight, and cumulative losses for BBRv1 and v3, respectively. The shaded area marked BBR's Startup phase. Initially, there is no packet loss in both CCAs, when they progressively probe for bandwidth and expand their CWnd. At around 0.4 s, the rate limiter ran out of tokens in its bucket and began throttling its output rate to the rate limit, causing a burst of packet losses. From this point, BBRv1 and v3 behaved differently.

Specifically, BBRv1 upon fast recovery sets its CWnd to packets inflight plus ACKed packets, as evident by the overlapping CWnd and packets inflight curves. As a result, the sending rate is reduced significantly, which also lowered the retransmission percentages. At around 0.7 s, BBRv1 successfully detected rate limiting and applied optimization to reduce retransmissions further.

In contrast, BBRv3 upon fast recovery sets its CWnd according to the bandwidth-delay product (BDP), calculated from the maximum bandwidth observed over the last 10 rounds. As the rate limiter has only just kicked in, the maximum bandwidth is really measuring the link's non-rate-limited bandwidth. Consequently, the BDP and, in turn, the CWnd are far too large for the now rate-limited link, causing far more packet losses than BBRv1 as evidence in Fig. 2.

Note that despite the higher retransmission percentages, BBRv3 still achieved goodput similar to BBRv1. This is because BBRv3 does not incorporate a rate-limit detector so that the mean pacing rate remained higher than the network's rate limit (e.g., 2.7 Mbps versus 2.2 Mbps in Op 1), thereby causing excessive packet losses, while still fully utilizing the link's bandwidth.

B Impact of Larger Bandwidth Delay Product

We explore the behavior of rate-limiting flows with larger BDPs in this appendix. As rate limits tend to be low in practice (up to 2 Mbps in most cases), larger BDPs are primarily due to longer RTT, e.g., when downloading data from a geographically distant server. To emulate such scenarios, we implemented Op 1's rate limit (at 2.07 Mbps) using a Cisco Business 250 Series Smart Switch [2] and then varied the end-to-end RTT between 40 ms, 120 ms, and 200 ms using Dummynet [1], resulted in BDPs of 10 KB, 30 KB, and 50 KB, respectively. Fig. 14 plots the retransmission percentages and goodput for file size ranging from 0.5 MB to 32 MB. Each configuration was repeated 30 times with the results averaged.

Overall, the results are consistent with the results for Op 1 (i.e., 10 KB) reported in Section 3.1. Specifically, the 0.5 MB flow exhibited zero retransmissions regardless of BDP size, as the file size is smaller than the estimated bucket size (733 KB) and thus the flow is not subject to rate limiting at all. Significant retransmissions occur once the file size exceeds the bucket size as expected.

For file sizes of 2-MB and larger, flows with longer RTT exhibited higher retransmission percentages than flows with shorter RTT. Our analysis shows that the longer RTT extended the detection time (in multiples of RTTs) for BBRv1's built-in rate-limiting detector and during this time, BBRv1 operates in the bandwidth probing phase. The longer it remains in the probing phase, the more it overshoots the rate limit, resulting in more packet losses once the rate limiter kicks in. We note that the 1-MB file size case does not exhibit the same behavior, because the file size is only slightly larger than the bucket size. Hence, more of the data are transferred without rate limiting. Longer RTT also offers more time for the token bucket to accumulate tokens before the rate limiter kicks in, resulting in lower retransmission percentages.

Finally, for all three RTT settings, the flows again have no problem achieving goodput close to the rate limit, consistent with the findings in Section 3.1.

C Table of Notations

For clarity and ease of reference, Table 4 consolidates the notations used throughout this paper. To provide full context, each description also includes a cross-reference to the section where the notation is first introduced and defined.

D Proof of Theorem 1

Proof. Referring to Fig. 5, let a_1 and a_2 be the area between the two curves from time $t = 0$ to $t = p_{j,j-1}$, and from $t = p_{j,j-1}$ to $t = t_i$, respectively. Let a_3 be the area between $A(\cdot)$ and the minimum of the two $C(\cdot)$ curves. Then we can determine which bucket size can result in the smaller gap area in Eq. (5) without the need for historical ACK information. Let Δ_{j-1} and Δ_j be the gap areas for bucket size b_{j-1} and b_j,

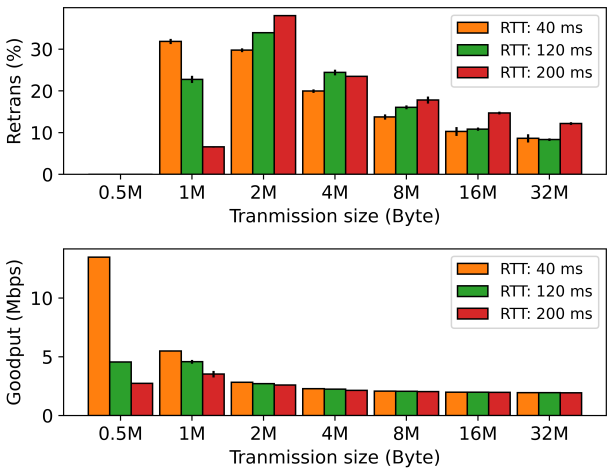

Figure 14: Impacts of BDP on BBRv1.

respectively. Then, we have:

$$\Delta_j = \Delta_{j-1} + a_1 - a_2. \tag{23}$$

Hence, we can compare them using just a_1 and a_2:

$$\Delta_j - \Delta_{j-1} = a_1 - a_2, \tag{24}$$

which can be calculated from the area of the triangles:

$$\begin{aligned} a_1 &= \frac{(b_j - b_{j-1})p_{j,j-1}}{2}, \\ a_2 &= \frac{(t_i - p_{j,j-1})\Big((b_{j-1} + r_{j-1}t_i) - (b_j + r_j t_i)\Big)}{2}. \end{aligned} \tag{25}$$

Substituting Eq. (25) into Eq. (24), we have:

$$\begin{aligned} &\Delta_j - \Delta_{j-1} \\ &= \frac{(b_j - b_{j-1})}{2}\left[p_{j,j-1} - (t_i - p_{j,j-1})\left(\frac{r_{j-1} - r_j}{b_j - b_{j-1}}t_i - 1\right)\right], \\ &= \frac{(b_j - b_{j-1})}{2}\left[p_{j,j-1} - \left(\frac{t_i^2}{p_{j,j-1}} - t_i - t_i + p_{j,j-1}\right)\right], \\ &= -\frac{(b_j - b_{j-1})}{2p_{j,j-1}}\left[\left(\frac{t_i}{p_{j,j-1}} - 1\right)^2 - 1\right]. \end{aligned} \tag{26}$$

Thus, $\Delta_j > \Delta_{j-1}$ implies:

$$\left(\frac{t_i}{p_{j,j-1}} - 1\right)^2 - 1 < 0 \tag{27}$$

or:

$$t_i < 2p_{j,j-1}. \tag{28}$$

□

Table 4: List the notations.

Notation	Description
u	The mean retransmission percentage for rate-limited flows of BBRv1 measured over one month is 29.6% (Section 2.1)
α	The length of an observation interval of packet loss to trigger detection (Section 4.1)
β	The packet loss threshold to trigger detection (Section 4.1)
B	Bucket size (Section 4.2)
R	Token replenish rate (Section 4.2)
$C(B,R,t)$	The maximum amount of data that can be delivered by a rate limiter with parameters $\{B,R\}$ at time t (Section 4.2)
$A(t)$	The cumulative amount of data acknowledged by the latest ACK packet at time t (Section 4.2)
B_{min}	Lower limit of the bucket size range (Section 4.2)
B_{max}	Upper limit of the bucket size range (Section 4.2)
ω	The number of bucket size candidates (Section 4.2)
Ω	The step size of the bucket size (Section 4.2)
R_h	The flow's mean throughput before detection triggers by ACK h at time t_h (Section 4.3)
θ	Abrupt rate reduction threshold of classification (Section 4.3)
ε	Convergence check threshold of classification (Section 4.3)
η	The frequency for increasing the cap (Section 5)
γ	The intensity factor of increasing the cap (Section 5)
e_i^k	The estimated number of accumulated tokens at time t_i during transfer k (Section 5.3)
g	The number of active flow (Section 5.4)
u	The sum of unsent data of a flow (Section 5.4)
v	The sum of packet inflight of a flow (Section 5.4)

E Proof of Theorem 2

Proof. We shall prove by contradiction. Given $\{b_j, r_j\}$ is a better fit than $\{b_{j-1}, r_{j-1}\}$, then invoking Theorem 1 by substituting $p_{j,j-1}$ in Eq. (11), we have:

$$\frac{b_j - b_{j-1}}{r_{j-1} - r_j} < \frac{t_i}{2}. \tag{29}$$

Assume $\{b_x, r_x\}$ is a better fit than $\{b_j, r_j\}$, then it must also be a better fit than $\{b_{j-1}, r_{j-1}\}$. Again, invoking Theorem 1 by substituting $p_{j-1,x}$ in Eq. (11), we have:

$$\frac{b_{j-1} - b_x}{r_x - r_{j-1}} > \frac{t_i}{2}. \tag{30}$$

Let t_q be the time at which $C(b_{j-1}, r_{j-1}, t_q) = A(t_q)$. For the tuple $\{b_j, r_j\}$, constraint Eq. (4) implies:

$$\begin{aligned} C(b_j, r_j, t_q) &\geq A(t_q), \\ b_j + t_q r_j &\geq b_{j-1} + t_q r_{j-1}, \\ \frac{b_j - b_{j-1}}{r_{j-1} - r_j} &\geq t_q. \end{aligned} \tag{31}$$

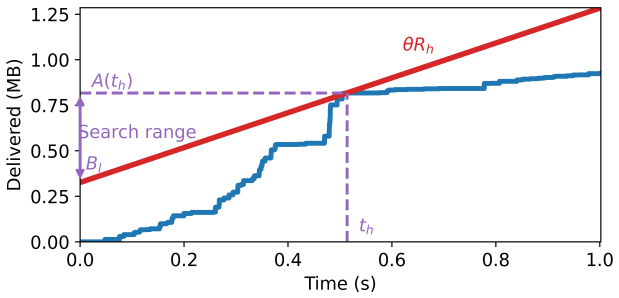

Figure 15: Narrow the range of candidates for bucket size.

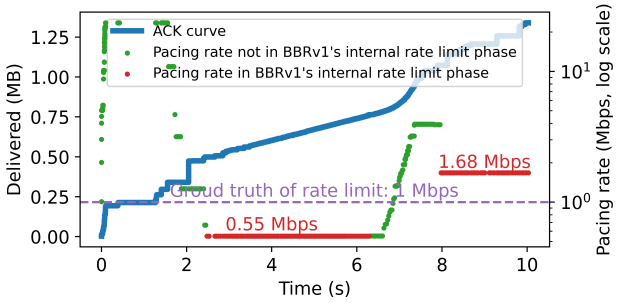

Figure 16: BBRv1's internal rate limit detector may underestimate the actual rate limit (e.g., between 2 s and 6 s), and impair R-TCP's own rate limit estimation.

Similarly, for the tuple $\{b_x, r_x\}$, constrain Eq. (4) implies:

$$\begin{aligned} C(b_x, r_x, t_q) &\geq A(t_q), \\ b_x + t_q r_x &\geq b_{j-1} + t_q r_{j-1}, \\ t_q &\geq \frac{b_{j-1} - b_x}{r_x - r_{j-1}}. \end{aligned} \tag{32}$$

Finally, substitute Eq. (32) into t_q in Eq. (31), and we get:

$$\begin{aligned} \frac{b_j - b_{j-1}}{r_{j-1} - r_j} &\geq \frac{b_{j-1} - b_x}{r_x - r_{j-1}}, \\ \frac{b_j - b_{j-1}}{r_{j-1} - r_j} &> \frac{t_i}{2}, \end{aligned} \tag{33}$$

which contradicts Eq. (29), so $\{b_x, r_x\}$ cannot be a better fit than $\{b_j, r_j\}$. □

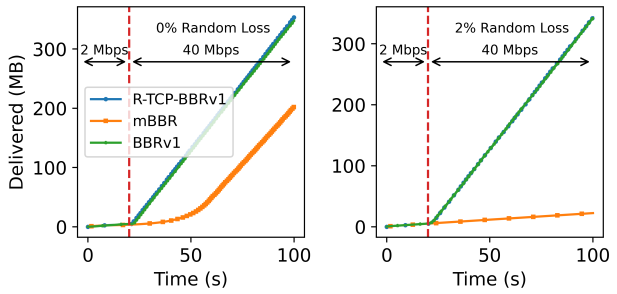

Figure 17: Experiment for lifting the rate limit after 20 seconds with (right) and without (left) random packet loss.

F Further Optimizations of Bucket Size Candidates

The abrupt rate reduction constrain in Eq. (13) limits the feasible estimated rate limit to the range $(0, \theta R_h)$. This can be exploited to further narrow down the feasible range for bucket size candidates as illustrated in Fig. 15. Specifically, the pivot point at t_h together with it being the maximum slope for a feasible $C(\cdot)$ curve, implies that bucket size candidates must be equal to or larger than B_l, which can be calculated from

$$\begin{aligned} B_l &= A(t_h) - t_h \theta R_h \\ &= A(t_h) - t_h \theta \frac{A(t_h)}{t_h} \\ &= A(t_h)(1 - \theta). \end{aligned} \tag{34}$$

Using B_l as the lower limit, then the optimized bucket size candidates become:

$$b_j = \begin{cases} B_l + j\left(\frac{B_{\max} - B_l}{\omega - 1}\right) & \text{for } j = 1, \ldots, \omega - 1, \\ 0 & j = 0. \end{cases} \tag{35}$$

The step size Ω is also updated to

$$\Omega = \frac{B_{\max} - B_l}{\omega - 1}. \tag{36}$$

G Interation Between BBRv1's Internal Rate Limiting Detector and R-TCP

BBRv1 has an inbuilt rate limiting detector / optimizer, which can interfere with R-TCP's parameter estimation process. This is illustrated in Fig. 16, where BBRv1's estimated rate limit from 2.5 s to 6.3 s (at 0.55 Mbps) is significantly lower than the actual one.

This is caused by the initial overshoot after sudden token depletion at the rate limiter, resulting in severe packet losses and low goodput (see Appendix A). As a result, BBRv1 will incorrectly set its pacing rate to the underestimated rate limit for 48 RTTs, which in turn cause R-TCP to underestimate the rate limit as well.

H Undetecting Rate Limiting

In this appendix, we investigate the scenario where the rate limit is suddenly raised significantly in the middle of a TCP flow. This can happen due to data quota reset (e.g., when crossing an accounting period boundary). We employed the Linux Traffic Control (tc) to emulate a link with RTT of 40 ms and a rate limit of 2 Mbps. A TCP flow was established at time 0 s transferring unlimited data. After 20 s, the rate limit is increased to 40 Mbps.

We tested BBRv1, mBBR, and R-TCP-BBRv1 and plotted their data delivery curve over time in Fig. 17. We observed that the curves for BBRv1 and R-TCP-BBRv1 overlap, both were able to (un)detect the rate limiting after 20 s and ramp up their transmission rates. In contrast, mBBR took considerably more time to ramp up its transmission rate, delivering 43% less data compared to R-TCP-BBRv1 over the same period. This is because mBBR only increases the maximum pacing by 20% once every 48 rounds, when there is no packet loss. If we introduce 2% random loss using tc, mBBR will not be able to exit from its rate-limiting state at all.

In contrast, R-TCP will lift the pacing rate cap completely, if the estimated $\{B,R\}$ changes during probing and eventually undetect the rate limiting condition.

I Hyper-Parameters Sensitivity Analysis for R-TCP's Detection Algorithm

Table 5 presents the hyper-parameters in R-TCP, with the value chosen as default in bold. We evaluated their impact on detection errors by varying their values over the ranges listed in Table 5. The primary metric we used is total classification error — defined as the sum of mean classification errors (false positives and false negatives) when applied to traces from Op 1 to 5.

The results are plotted in Fig. 18 for all five hyper-parameters. The first observation is that the total classification error is insensitive to α, ε, and ω. The choices of α and ε trade detection time against classification accuracy. The choice of ω trade memory/computation overheads against classification accuracy as well as parameter estimation accuracy. The latter is also plotted in Fig. 18. The current defaults ($\alpha = 7, \omega = 9, \varepsilon = 10$) strike a balance between the tradeoffs, although the differences are relatively small.

The choice of the detection trigger threshold β, by contrast, has more impact. Too small a β will trigger more often, but the classification error is not increased significantly, because the classification process is done separately as described in Section 4.3. Too large a β does degrade classification performance, as it will cause more false negative cases by suppressing the detection trigger. The current default of $\beta = 0.2$ offers low classification error with minimal overheads as reported in Appendix N.

Table 5: The internal parameters of R-TCP's detection algorithm for sensitivity analysis.

Setting	Description	Value
α	Detection trigger (Section 4.1)	4, 5, 6, **7**, 8
β	Detection trigger (Section 4.1)	0.1, **0.2**, 0.3, 0.4
θ	Abrupt rate reduction (Section 4.3)	0.2, 0.4, **0.6**, 0.8
ω	Number of bucket candidates (Section 4.2)	5, **9**, 13, 17
ε	Convergence (Section 4.3)	2, 3, 4, 5, 6, 7, 8, 9, **10**, 11, 12, 13, 14

Next, the abrupt rate reduction threshold θ obviously impacts classification accuracy. Here, too small a θ will result in more false positives, as normal bandwidth fluctuations would be incorrectly treated as abrupt rate reduction under rate limiting. Too large a θ, on the other hand, will result in more false negatives, as the rate reduction may not be large enough to satisfy the threshold even under rate limiting. The current default of $\theta = 0.6$ offers a good balance in the 5 operators tested.

J Hyper-Parameters Sensitivity Analysis for R-TCP's Optimization Algorithm

R-TCP regulates transmission by setting caps on the CWnd or pacing rate. To account for potential estimation errors, the parameters η and γ, introduced in Section 5, are proposed to increase or lift these caps periodically to update the estimated rate limiter parameters. To assess the hyper-parameters' sensitivity, we evaluated their impact on retransmission percentages and goodput in transferring an 8-MB file in Op 2 and Op 3 (Table 1).

Table 6 and Table 7 present the results of the sensitivity analysis for BBRv1 and Cubic, respectively, averaged over 100 samples. The tables reveal that smaller cap increases (γ) and less frequent cap adjustments (η) result in fewer retransmissions. However, in the case of BBRv1, this reduction in retransmissions may come at the expense of lower goodput. This is because BBRv1's pacing rate is determined by the estimated bandwidth from previous rounds, so it takes time for it to ramp up the pacing rate, when the pacing rate cap is raised.

In contrast, Cubic can increase the CWnd whenever an ACK packet is received, so it can ramp up transmission more quickly, when the CWnd cap is raised. As a result, Cubic's goodput performance is insensitive to these two hyper-parameters.

We selected the parameter set ($\eta = 20, \gamma = 20\%$) to balance goodput and retransmission percentage.

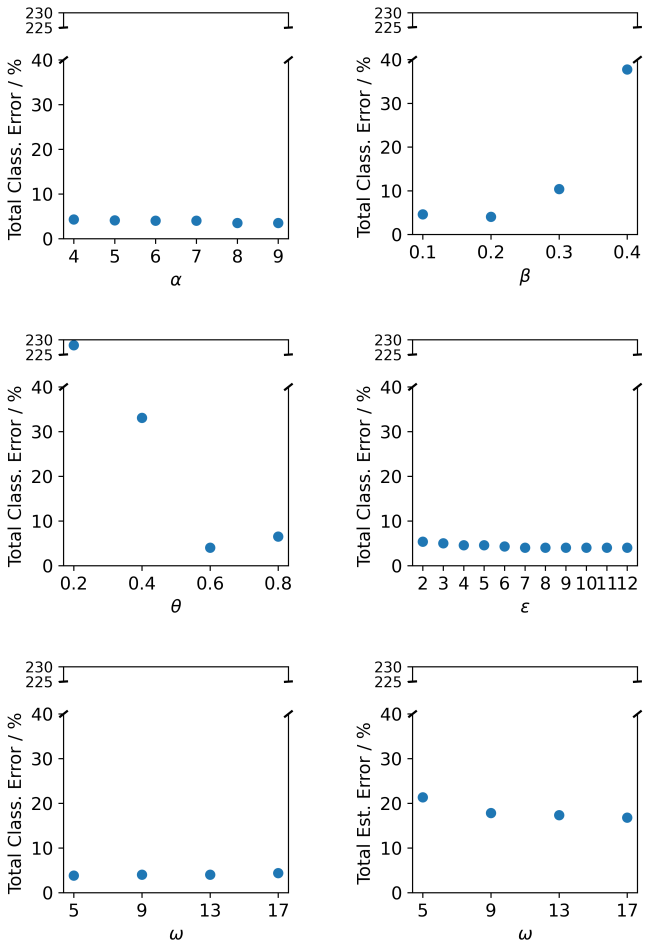

Figure 18: Sensitivity analysis of hyper-parameters for R-TCP's detection algorithms.

K Detection Performance on Real-world Trace Data

We present more experiment results for R-TCP's detection performance and compare it to two existing rate limiting detection algorithms, namely PD [18] and P-MODRL [33]. Detection performance has two components: rate-limiting classification error and parameter estimation error. The former is sub-divided further into two types:

- False positive, where a non-rate-limiting network is incorrectly classified as rate limiting.
- False negative, where a rate-limiting network is incorrectly classified as non-rate-limiting.

Parameter estimation error is the normalized difference between the estimated bucket size B and rate limit R, and the ground truth. We note that, although the rate-limiting mobile service does specify a rate limit (after data quota are exhausted), none of them discloses the bucket size. Moreover, our measurements indicate that the actual rate limit may not necessary equal to the stated one exactly.

Table 6: The internal parameters of R-TCP's optimization algorithm for sensitivity analysis under BBRv1.

		Op 2		Op 3	
η	γ	Retrans	Goodput	Retrans	Goodput
15	10%	3.38%	1.25 Mbps	4.18%	0.84 Mbps
15	15%	3.97%	1.25 Mbps	5.91%	0.92 Mbps
15	20%	4.70%	1.27 Mbps	6.13%	0.95 Mbps
15	25%	5.07%	1.27 Mbps	7.00%	0.95 Mbps
20	10%	3.15%	1.24 Mbps	3.85%	0.80 Mbps
20	15%	3.51%	1.25 Mbps	5.04%	0.87 Mbps
20	20%	3.95%	1.27 Mbps	5.79%	0.95 Mbps
20	25%	4.43%	1.27 Mbps	6.38%	0.95 Mbps
25	10%	2.93%	1.22 Mbps	4.06%	0.79 Mbps
25	15%	3.36%	1.24 Mbps	4.17%	0.82 Mbps
25	20%	3.61%	1.25 Mbps	5.39%	0.87 Mbps
25	25%	3.99%	1.26 Mbps	5.87%	0.91 Mbps

Table 7: The internal parameters of R-TCP's optimization algorithm for sensitivity analysis under Cubic.

		Op 2		Op 3	
η	γ	Retrans	Goodput	Retrans	Goodput
15	10%	4.60%	1.27 Mbps	4.57%	0.98 Mbps
15	15%	4.62%	1.27 Mbps	4.61%	0.98 Mbps
15	20%	4.65%	1.27 Mbps	4.65%	0.98 Mbps
15	25%	4.69%	1.27 Mbps	4.75%	0.98 Mbps
20	10%	4.44%	1.27 Mbps	4.44%	0.98 Mbps
20	15%	4.46%	1.27 Mbps	4.51%	0.98 Mbps
20	20%	4.47%	1.27 Mbps	4.55%	0.98 Mbps
20	25%	4.61%	1.27 Mbps	4.66%	0.98 Mbps
25	10%	4.42%	1.27 Mbps	4.40%	0.98 Mbps
25	15%	4.44%	1.27 Mbps	4.42%	0.98 Mbps
25	20%	4.45%	1.27 Mbps	4.44%	0.98 Mbps
25	25%	4.60%	1.27 Mbps	4.52%	0.98 Mbps

Therefore, we follow the methods in Zhu *et al.* [33] to measure and estimate the mobile service's actual rate limiter parameters $\{B,R\}$ using a customize software, which transmits UDP datagrams at high data rate to a client through the mobile network. In the following, we evaluate R-TCP's detection performance using both offline and online experiments.

We collected TCP trace data using both Cubic and BBR as the CCA under three rate-limiting and two non-rate-limiting networks (c.f. Table 1), utilizing a stationary client host connected to the mobile network via a 5G modem. Each trace was generated from a 30-second download task from an Apache server. For each network service, we collected over 1,000 traces for both BBR and Cubic.

We modified BBRv1 and Cubic to log key metrics, includ-

Table 8: Comparison of classification errors.

		PD	P-MODRL	R-TCP
Op 1	BBRv1	91.41%	0.28%	0.46%
	Cubic	81%	0.37%	0.73%
Op 2	BBRv1	100%	5.16%	0.82%
	Cubic	100%	0.29%	1.18%
Op 3	BBRv1	100%	1.00%	0.14%
	Cubic	100%	1.18%	0.07%
Op 4	BBRv1	0%	5.56%	0.46%
	Cubic	0%	2.93%	0.15%
Op 5	BBRv1	0%	1.47%	0%
	Cubic	0%	1.70%	0%
Op 4M	BBRv1	0%	33.63%	3.59%
	Cubic	0%	21.56%	3.67%

Table 9: Comparison of parameter estimation error.

		P-MODRL		R-TCP	
		B (%)	R (%)	B (%)	R (%)
Op 1	BBRv1	3.13	0.11	5.71	0.41
	Cubic	2.94	0.08	6.97	0.50
Op 2	BBRv1	49.76	5.26	56.16	4.84
	Cubic	60.91	2.72	60.33	2.78
Op 3	BBRv1	24.30	5.58	25.54	5.47
	Cubic	36.21	2.87	34.06	2.96

ing the number of acknowledged bytes, reception time, RTT, and the number of delivered and lost packets upon receiving an ACK. The goal is to apply different detection algorithms to the same TCP trace data to compare their performance under the exact same network conditions.

Table 8 compares the detection errors of PD, P-MODRL, and R-TCP, when applied to Cubic and BBRv1 TCP traces. PD has a very high error rate, consistent with the results from *Zhu et al.* [33]. For the three rate-limiting networks (Op 1 to 3), R-TCP has similar false negative performance to P-MODRL except for BBRv1 in Op 2 where P-MODRL exhibited a higher error rate (5.16% vs. 0.82%). For the two non-rate-limiting networks (Op 4 and 5), R-TCP generally outperforms P-MODRL with very low (Op 4) to zero (Op 5) false positive rates. Note that PD achieved zero false positive rates, as it tends to err on the side of classifying a flow to be non-rate-limiting. To be fair, unlike P-MODRL and R-TCP, PD was not specifically designed for mobile networks, so this may not be its intended network environment to operate in.

The above TCP traces were captured from a stationary client. To investigate the impact of more challenging network environments, we captured another set of traces using Op 4, when riding subway and buses, denoted by Op 4M. The coefficient-of-variation (CoV) of the throughput of the TCP traces in Op 4M is 39%, much higher than that of Op 4 (at 23%) when the client is stationary.

As expected, Op 4M's more challenging network environment increased R-TCP's false positive rates but are still within a few percentage points. In contrast, P-MODRL's error rates increased significantly to 33.63% (BBRv1) and 21.56% (Cubic). The false positives are caused by sudden bandwidth drops in Op 4M, due to the rapidly changing network conditions. These drops are then incorrectly detected as abrupt rate reduction (c.f. Section 4.4), which led to the false positives.

R-TCP's detection trigger in Eq. (2) filtered bandwidth drops not accompanied by high packet losses and its un-detection mechanism in Section 4.4 mitigated many of the false positives. Without them, R-TCP's false positives under Op 4M would increase to 8.07% (BBR) and 5.96% (Cubic).

Table 9 compares the parameter estimation errors between R-TCP and P-MODRL. Both algorithms achieved similar accuracy. We note that both algorithms exhibited large errors in estimating the token bucket size B in Op 2 and 3. Our own estimations using UDP to flood the rate limiter also showed much larger variations in the measured token bucket size (c.f. Table 1). The reason for the variations is not clear at the time of writing and the authors are currently investigating this anomaly further.

Despite the large errors in estimating B, the rate limit estimates are still reasonably accurate. R-TCP primarily makes use of R in its optimization algorithm, so the token bucket size error does not impact optimization performance significantly.

L Detection Performance in Controlled Testbed

In this appendix, we evaluate the detection performance of R-TCP across a wide range of known token bucket configurations. These configurations include a broad range of bucket sizes (4 KB to 3 MB) and rate limits (0.2 to 20 Mbps). The extended ranges are designed to test the operational boundaries of R-TCP's detection algorithm as implemented into BBRv1.

The testbed utilized a Cisco Business 250 Series Smart Switch [2] to emulate the token bucket and employed Dummynet [1] to introduce 40 ms round-trip time [31]. Each experiment run lasted for 30 seconds, with each token bucket configuration repeated five times and the results averaged.

Fig. 19 and 20 show the mean percentage error in estimating the bucket size and rate limit, calculated from:

$$E_B = \frac{100}{Q}\sum_{i=1}^{Q}\frac{|B-b_i|}{B}, \tag{37}$$

$$E_R = \frac{100}{Q}\sum_{i=1}^{Q}\frac{|R-r_i|}{R}, \tag{38}$$

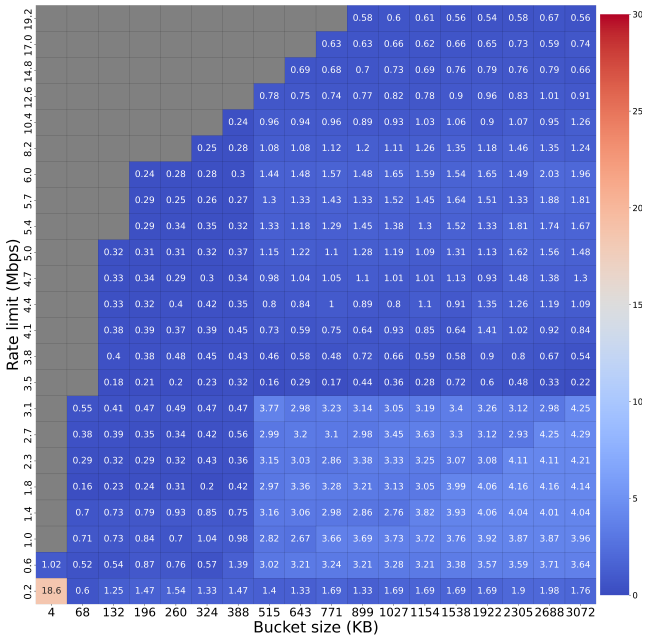

Figure 19: Estimation error percentages for rate limit under controlled testbed (grey boxes indicate failures to detect rate limiting).

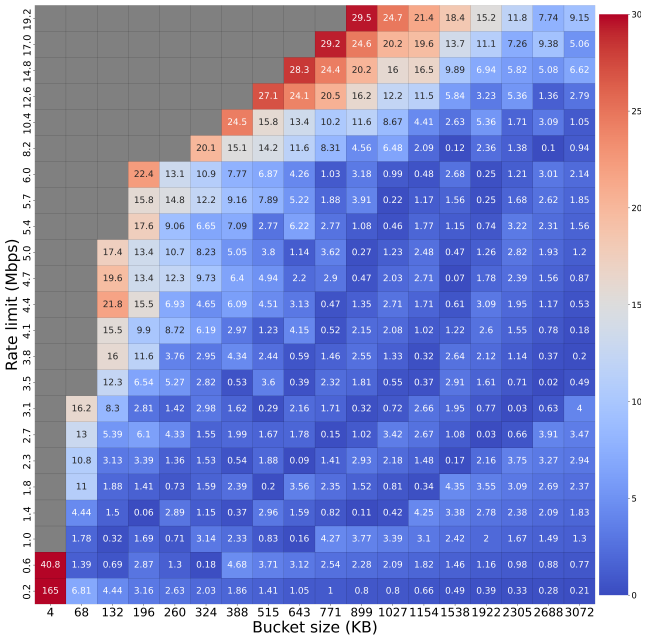

Figure 20: Estimation error percentages for bucket size under controlled testbed (grey boxes indicate failures to detect rate limiting).

where Q is the number of runs that can successfully detect rate limiting under the given token bucket configuration, and $\{b_i, r_i\}$ and $\{B, R\}$ represent the estimated and ground truth token bucket parameters, respectively.

R-TCP's rate limit estimation errors are remarkably small and consistent across most token bucket configurations. Bucket size estimation errors do increase at the boundary configurations, i.e., larger rate limit to bucket size ratios. The grey area in Fig. 19 and 20 represents regions where R-TCP fails to detect rate limiting, i.e., false negative. Nevertheless, these out-of-bound regions involve either very high rate limit (>6 Mbps), very small bucket size (<260 KB), or both. Given that most of the rate-limiting mobile services have (intentionally) low rate limits (e.g., hundreds of Kbps to 2 Mbps), they are well within R-TCP's operating boundary.

M Application of R-TCP to Learning-based CCAs

This appendix explores the application of R-TCP to unconventional TCP CCAs to demonstrate how easy it is to adapt R-TCP to different CCA designs. Specifically, we applied R-TCP to Astraea [25] - a reinforcement-learning-based CCA. Astraea is composed of two components. The first component is a modified TCP CCA module in the Linux kernel, where it accepts a CWnd setting from a custom socket option. It also employs pacing where the pacing rate is computed from CWnd/smoothed_RTT. The second component runs in user mode executing the reinforcement learning module. It takes inputs such as throughput, RTT, CWnd, pacing rate, number of packets in flight, number of retransmissions packets, and loss ratio from the custom socket option API. The module generates an output CWnd, which is then passed to the kernel module via the same custom socket option API.

Adapting R-TCP to a TCP CCA involves two steps. The first step is to include R-TCP's detection module into the TCP module. This can be done in a straightforward manner, as it is passive and hence does not interact with the CCA itself. The second step is to make use of the detected rate-limit parameters $\{B, R\}$ to regulate transmission. This can be done via limiting the CWnd, pacing rate, or both.

For Astraea, we apply both to convert it into R-TCP-Astraea. First, in the kernel TCP module, the CWnd computed upon ACK processing will be capped to the estimated BDP as in Section 5.1. Second, the computed pacing rate will also be capped to the detected rate limit R. The other optimization mechanisms are the same as those introduced in Section 5.

We conducted an experiment using the file download test settings from Section 6.1 in Op 3 (Table 1). Fig. 21 compares the mean retransmission percentages and goodput averaged over 30 runs. It is evident that Astraea alone exhibited similar excessive retransmissions (20∼30%) under rate limiting networks. Moreover, its goodput is also degraded, especially for smaller file sizes.

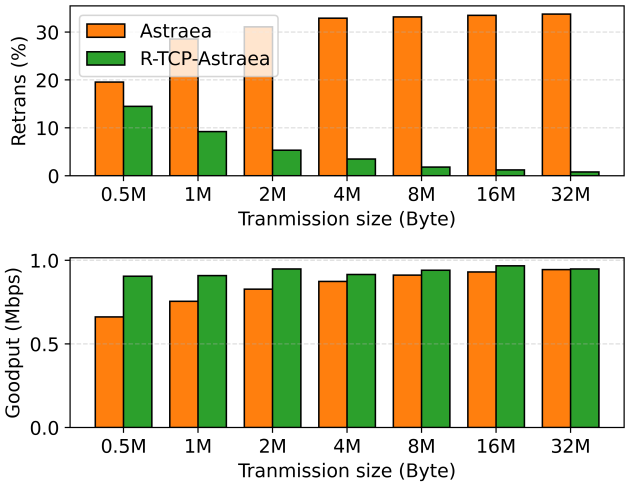

Figure 21: Comparison of retransmission percentages in rate-limiting mobile networks.

In comparison, R-TCP-Astraea effectively reduced Astraea's retransmission percentages by up to 98% and improved its goodput by up to 36% (at 0.5-MB file size). This experiment further demonstrates R-TCP's applicability to different types of CCA and its effectiveness in reducing excessive retransmissions, while also enhancing their goodput performance in this case.

N R-TCP Overheads

We measured the memory and CPU overheads of our R-TCP implementation in Linux. R-TCP allocates a 250-byte data structure for each TCP connection in the kernel. According to our industry collaborator, a typical video server is dimensioned to support 8K concurrent connections. This translates into 2 MB additional kernel memory consumption, which is well within acceptable level for today's servers.

For CPU utilization, we conducted benchmark tests, where the server with the configuration listed in Table 2 activately transfers data over 8K concurrent HTTP connections using Apache. The recorded CPU utilizations when running all connections using BBRv1 versus R-TCP-BBRv1 are both around 24%. This shows that R-TCP's additional processing does not raise the CPU utilization by any observable amount.

O R-TCP-BBRv3 in Short-video Applications

Table 10 compare the performance of R-TCP-BBRv3 to BBRv3 in Op 1 and Op 2. Note that the numerical results in Table 10 are not directly comparable to Table 3, as the two experiments were conducted at different times. The test setting and scenario are the same as those described in Section 6.3.

Table 10: BBRv3 performance in short video application.

		Avg. Retrans. (%)	Rebuf. Count Ratio	Rebuf. Dur. Ratio	Avg. Bitrate (Kbps)
Op 1	BBRv3	23.0%	3×10^{-7}	5×10^{-4}	487
	SR-TCP-BBRv3	11.6%	2×10^{-7}	4×10^{-4}	485
Op 2	BBRv3	35.0%	2×10^{-6}	6×10^{-3}	412
	SR-TCP-BBRv3	15.6%	2×10^{-6}	4×10^{-3}	409

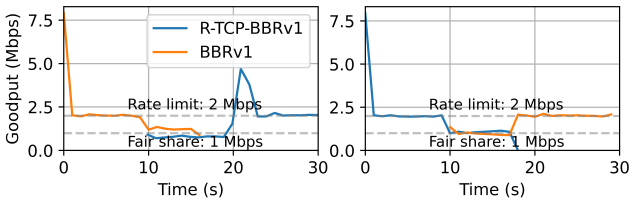

Figure 22: Study of fairness between R-TCP and BBRv1 with a BBRv1 (R-TCP) flow started first in the left (right) figure followed by a R-TCP (BBRv1) flow joining 10 seconds later. Both flows transferred a 4-MB file in Op 1.

R-TCP can reduce the retransmission rates by 50% and 55% compared to BBRv3 under Op 1 and Op 2, respectively. Furthermore, the streaming performance under R-TCP is comparable to BBRv3. Due to the lower retransmission rates, R-TCP reduced the total bandwidth usage by 24% without compromising streaming performance.

P Fairness

We study R-TCP's fairness when competing with BBRv1 using two scenarios in Fig. 22. In the first scenario (left figure), R-TCP joins an existing BBRv1 flow at 10 s where they begin to share the maximum rate (∼2 Mbps) allowed by the rate limiter. The R-TCP flow in this case did not detect rate limiting initially, so it is in fact operating as BBRv1, thereby exhibiting the same inherent fairness behavior as BBRv1. After the BBRv1 flow is completed, the R-TCP flow probes for more bandwidth following the BBRv1 logic.

In the second scenario (right figure), the order is reversed. R-TCP detected rate limiting early on and capped BBR's pacing rate to the detected rate limit as expected. When the BBRv1 flow joined at 10 s, the R-TCP flow released bandwidth to share with the BBRv1 flow. This is in fact accomplished by the BBR CCA alongside R-TCP as the latter only caps the maximum pacing rate, so downward adjustments are not affected. These two experiments show that the R-TCP framework does not interfere with BBR's original bandwidth sharing behavior.