Optimizing TCP Loss Recovery Performance Over Mobile Data Networks

Ke Liu, Zhongbin Zha, Wenkai Wan, Vaneet Aggarwal, Senior Member, IEEE, Binzhang Fu, and Mingyu Chen

Abstract—Recent advances in high-speed mobile networks have revealed new bottlenecks in ubiquitous TCP protocol deployed in the Internet. In addition to differentiating non-congestive loss from congestive loss, our experiments revealed two significant performance bottlenecks during the loss recovery phase: flow control bottleneck and application stall, resulting in degradation in QoS performance. To tackle these two problems, we first develop a novel opportunistic retransmission algorithm to eliminate the flow control bottleneck, which enables TCP sender to transmit new packets even if receiver’s receiving window is exhausted. Second, application stall can be significantly alleviated by carefully monitoring and tuning the TCP sending buffer growth mechanism. We implemented and modularized the proposed algorithms in the Linux kernel thus they can plug-and-play with the existing TCP loss recovery algorithms easily. We evaluated our proposed algorithms over emulated and real experiments and showed that, compared to existing TCP loss recovery algorithms, the proposed optimization algorithms improve the bandwidth efficiency by up to 133 percent and completely mitigate RTT spikes, i.e., over 50 percent RTT reduction, over the loss recovery phase.

Index Terms—TCP, loss recovery, mobile data network, opportunistic retransmission, application stall

1 INTRODUCTION

Recent advances in high-speed mobile networks have revealed new bottlenecks in the ubiquitous TCP protocol deployed in the Internet. For example, Mascolo et al. [1] and Fu and Liew [2] first proposed to regulate the congestion window (CWnd) adaptively upon every loss event to differentiate the random packet loss in mobile networks from congestion loss, instead of reducing CWnd blindly. They showed that their methods are effective – unnecessary throughput degradations resulting from random losses can be mitigated, thus improving TCP’s throughput performance.

More recently, adaptive loss recovery algorithms were invented and commonly adopted in numerous systems [3], [4], [5], [6] and rate/congestion control algorithms [7], [8], [9], [10], [11], which have been used to replace the existing congestion control algorithms in conventional TCP. The key adaptive loss recovery algorithms can be divided into two categories: queue length (or delay) adaptive loss recovery algorithm and bandwidth adaptive loss recovery algorithm. By decoupling the packet loss recovery from the congestion control (as fair bandwidth sharing is ensured by the link layer in mobile networks), their sending rates/CWnds for both retransmissions as well as new data packets are only affected by network variations, e.g., bandwidth.

Specifically, queue length adaptive loss recovery algorithm continuously adjusts its sending rate or CWnd during the loss recovery phase so that the estimated queue length will oscillate around a target queue length. Bandwidth adaptive loss recovery algorithm tracks the bandwidth availability by estimating the DUPACK and ACK returning rate, and regulates the sending rate/CWnd during the loss recovery phase accordingly. Compared to conventional TCP, these adaptive loss recovery algorithms effectively alleviate the throughput drop due to the congestion control bottleneck during the loss recovery phase.

Nonetheless, these works did not consider the flow control bottleneck in TCP’s loss recovery phase – receiver’s receiving buffer (AWnd) prevents new data packets from transmitting (i.e., Head-of-Line Blocking) if AWnd is exhausted, resulting in bandwidth underutilization. However, those new data packets will be transmitted in burst after loss recovery phase, resulting in the significant increase in RTT – RTT spikes. Recent measurements in productions of mobile networks showed that the receiver’s receiving buffer size mainly limits the application throughput [3], [12]. Although this problem can be alleviated by increasing the AWnd size, first, it is not cost-effective and presents significant difficulties in practical deployment, as there are several OS need to modify (e.g., Linux, Microsoft Windows, Apple’s iOS). In addition, this could result in another problem – bufferbloat [13], [38], [39], as a larger AWnd size allows more buffering in the network. This problem can be addressed by DRWA [13], RSFC [14] and Windows receiver window auto-tuning algorithm [37], all of which control the AWnd size dynamically according to BDP, but these solutions also suffer from the throughput degradation limited by flow control bottleneck during the loss recovery.

To optimize the throughput performance over the loss recovery phase, we revisit the existing loss recovery
algorithms, and find that the effect from the congestion control algorithm, e.g., CUBIC and BBR, on the throughput performance over the loss recovery phase is limited and dominated by the loss recovery algorithm. Thus, we first develop two adaptive loss recovery algorithms: Queue-length Aware Rate Reduction (QARR) and Bandwidth-Aware Rate Reduction (BARR) to eliminate the congestion control bottleneck. Second, we develop a novel opportunistic retransmission algorithm to address the flow control bottleneck by relaxing the AWnd constraint during TCP’s loss recovery phase, hence allowing TCP sender to transmit new TCP segments as soon as the lost TCP segments (which have been detected so far) have been retransmitted irrespective of the AWnd size.

Interestingly, the throughput performance improves, but not as much as we expected. After investigating the trace data we discovered that TCP sender always runs out of packets—application stall. As the sender’s send buffer (sndbuf) cannot fetch new packets if the free space (free_space) does not exceed 1/3 of sndbuf, which affects the performance of opportunistic retransmission requires a large number of new packets, e.g., BDP. Given current and future mobile networks, e.g., 4G/LTE offers 100Gbps bandwidth [3] and 5G offers near 20 Gbps maximum bandwidth [40], which leads to a large BDP, the bandwidth inefficiency due to application stall becomes significant. This does occur during transmissions, especially for the loss recovery, causing TCP sender has no new packets to transmit opportunistically. Our experiments (c.f., Sections 5 and 6) show that application stall occurs during over 61.1 percent of the loss recovery phase in practice. Dukkipati et al. [15] also demonstrated this problem but did not propose any solution. Similarly, increasing sending buffer size can alleviate the problem but we found that increasing it too large could counter-interact with the underlying TCP implementation, causing TCP throughput collapse, and it is not cost-effective in terms of memory by unnecessarily allocating more memory. In this paper, we tackle this problem by adaptively refining the sending buffer growth mechanism to guarantee that TCP sending buffer always has data packets to transmit.

It is now increasingly common for mobile vendors to deploy transparent proxies or servers inside the networks of mobile operators [3], [4], [5], [6], [12], [13], [14], [16], [17]. The TCP flow will end-up terminating at the proxy/server. The mobile operators are very likely to deploy an optimized TCP stack rate controller [7], [8], [9], [10], [11], [14], [18], [19], [20] at the proxies/servers for the last leg from the proxies/servers to the mobile device. Therefore, we could realize the proposed algorithms in those proxies/servers. Therefore, we modularized the proposed algorithms in the Linux kernel and thus they can plug-and-play with existing proxy/server’s TCP implementations easily. This approach is practical because they do not require any modifications to the existing TCP implementation at the real client/server hosts, thus can be readily deployed into current mobile data networks.

We evaluate our proposed algorithms using emulated experiments and real experiments, and port them to a prevalent application video streaming. We show that, with our proposed algorithms, (a) the bandwidth over the loss recovery phase can be efficiently utilized and RTT spikes are mitigated completely; (b) the bandwidth efficiencies of all conventional TCP variants can be improved substantially and achieve near-optimal bandwidth utilization (over 89 percent) over mobile data networks; (c) the QoE performance of video streaming is also improved.

Compared to previous accepted conference paper [34], we made the following extensions and contributions:

- We introduce the mobile network properties that differ from the other networks, e.g., wired networks and Wi-Fi, which motivates our solution (c.f., Section 2.1).
- We compare our proposed algorithms to the recent related works and discuss the pros and cons of our approach (c.f., Section 2.2).
- We leverage more extensive controlled experiments to verify the impact of flow control bottleneck and application stall and performance gain of our proposed algorithms (c.f., Sections 5 and 6).
- We evaluate our proposed algorithms over diverse network conditions and one production of mobile networks (c.f., Sections 8.7 and Section 8.9).
- We evaluate the improvement of the proposed algorithms on QoE of the video streaming (Section 8.9).
- We present the limitation of our proposed algorithms (c.f. Section 9).

## 2 Motivations and Related Works

In this section, we first highlight some differences between mobile networks and other networks such as wired network and Wi-Fi, which motivates our work. Second, we review the related works and compare them to our approach.

### 2.1 Mobile Network Properties

Mobile networks differ from wired networks and Wi-Fi from several aspects [3], [41], [42], [43]. First, the wired network and Wi-Fi routers do not explicitly schedule traffic flows from different users, thus the TCP congestion control is instrumental in achieving fair bandwidth sharing at bottleneck links. By contrast, base stations in mobile networks maintain separate uplink/downlink queues for each user and perform link-layer bandwidth regulation independently. Our previous study verifies it by showing that one user’s leaving and joining has little effect on the other user’s throughput. Thus, one user’s throughput degradation cannot be shared by other users.

Second, 3G/LTE base stations typically have large per-user link buffers – in the order of multiple MBs [3], [41], [42], [43]. We believe that such a large buffer is needed to (a) support link-layer retransmission, and (b) reduce packet loss rates due to buffer overflow in the presence of rapid bandwidth fluctuations. Consequently, fair bandwidth sharing of the radio link among multiple users is already enforced by the base station. This opens a new design space for rate and congestion controls.

Third, we observed the occurrence of non-negligible packet losses [3], [44], [53] (c.f., Table 12), although it was believed that the loss rate is extremely low over mobile networks.

Fourthly, it was confirmed that the receiver’s window size (AWnd) limit becomes the main performance bottleneck in application throughput over mobile data networks [3], [43], including loss recovery phase. This work explores these properties, design and develop a set of techniques to eliminate the possible throughput degradation.
2.2 Related Works and Comparisons

Previous works mainly addressed the performance bottlenecks in mobile networks with two approaches. First, the rate and congestion control algorithms based on TCP/IP stack. They developed new TCP sender-side rate control algorithm, such as TCP-RRE [7], BBR [45] and ProRate [46], which can be compiled as a pluggable congestion control module thus can be readily deployed in Internet servers. Second, the rate control algorithms based on a newly developed framework, such as Remy [47], Tao [48], Sprout [41], Verus [42], Copa [49], PCC [50], PCC-Vivace [51], Pantheon [52], mTCP [69], etc., which has several practical issues. First, some of the frameworks, e.g., mTCP, bypass the kernel and replace TCP/IP stack with a new user-space framework, which causes security problem, because kernel can monitor every packet thus prevent common network attacks from happening (e.g., iptables). Additionally, Linux kernel keeps evolving with security patches to strengthen the userspace applications security. Second, they require modifications on both the server and client-side, which presents significant difficulties in practical deployment, as there are many different OS implementations in use (e.g., Linux, Microsoft Windows, Apple’s iOS). Third, some of them use UDP or UDT to carry payload, which could be filtered, rate-limited or even blocked in some ISP. Due to the above reason, our work follows the first approach. QUIC is an emerging user-space transport that takes a long time to standardized [66]. However, rate-limiting and block on QUIC traffic is still possible from [67], [68], thus QUIC can seamlessly fallback to TCP if its connection is block [67], [68]. Besides optimizing TCP traffic, we also can implement our algorithms based on QUIC to optimize QUIC traffic, which can be one of our future works.

Compared to previous TCP sender-side rate and congestion control algorithms which address the performance bottlenecks during the TCP normal phase, i.e., without packet losses, this work investigated the performance bottleneck in TCP loss recovery algorithms, and found that, during the loss recovery phase, the throughput is govern by a separate algorithm, i.e., RH (c.f., Section 3.2) and PRR (c.f., Section 3.3), which is independent of the congestion control algorithm in the TCP normal phase, e.g., BBR [45], CUBIC, thus their optimization cannot apply to the TCP loss recovery phase. We first eliminate the congestion control bottleneck in RH and PRR by proposing two adaptive loss recovery algorithms: QARR (c.f., Section 3.4) and BARR (c.f., Section 3.5), both of which decouples the bandwidth sharing from rate control, thus the throughput is only governed by the network variations. Second, we identified the flow control bottleneck during the loss recovery phase as described in Section 2.1 and addressed it by proposing our optimizations. To ease deployment, we modularized the loss recovery algorithm in Linux kernel so that users can switch between existing loss recovery algorithms easily and plug-and-play their own loss recovery algorithms (Section 7).

Our work aims to improve the performance of TCP loss recovery algorithm which is a reactive retransmission scheme that triggered by DUPACKs. Another option to recover lost packets is redundancy-based recovery. One category is coding like FEC [54]. While coding provides resilience, in practice, the amount of redundancy can be from 20 to 50 percent [54]. The other category is redundant packet transmission [55], [56], [57] that sends multiple copies of the same packet. If at least one copy of the packet avoids network congestion, the data is received. Both categories might not be cost-effective when the network is the bottleneck. A. Vulimiri et al. [65] studies when redundancy improves the application performance (i.e., latency) and when it does not, and find a threshold load below which redundancy always helps improve mean latency, but beyond which extra load it adds overwhelms any latency reduction (e.g., the server-side threshold load lies between 25 and 50 percent utilization in its model). Thus, this approach fits for applications with limited throughput (i.e., network is not the bottleneck), which could safely send redundant packets or add redundancy to the network. Compared to it, our approach is based on the reactive retransmission scheme and requires applications to be throughput-intensive, thus their throughputs are possible to be limited by the flow control bottleneck over mobile networks, and it is observed that mobile application throughput is mainly limited by the flow control bottleneck. In addition, redundancy-based recovery also requires significant modifications on both ends, which is more difficult to deploy than our solution for mobile data networks that contain machines/devices with heterogeneous OSes.

3 TCP Loss Recovery Algorithms Revisited

TCP loss recovery algorithm includes two mechanisms, i.e., fast retransmit that retransmit the missing packet immediately when the third DUPACK in a row is received, and fast recovery that defines the way to adjust CWnd before the ACK arrives that acknowledges new data. Although fast recovery algorithm was standardized in RFCs [21], [58], the actual implementation differs from OS distributions. Thus, we revisit the five existing loss recovery algorithms: (a) the standard TCP loss recovery algorithm as defined in RFC3517 [21]; (b) rate-halving [22]; (c) proportional rate reduction; (d) queue length adaptive rate reduction and (e) bandwidth adaptive rate reduction.

3.1 Standard TCP Loss Recovery Algorithm

According to RFC3517, TCP enters the loss recovery phase either after a packet retransmission timeout or upon receiving dupthresh (typically three) number of duplicate ACKs (DUPACKs). In the timeout case, the CWnd size will be reduced to one and the TCP sender enters the slow start phase again. In this case, CWnd is likely larger than CWnd and thus unlikely to be the bottleneck. Therefore, we will not consider the timeout case in the rest of the paper.

In the second case, the TCP sender will enter the loss recovery phase and set both CWnd and slow start threshold (ssthresh) to equal to $0.5 \times \text{FlightSize}$ – where FlightSize is the amount of data that has been transmitted but not yet cumulatively acknowledged. Next, the TCP sender will retransmit the first unacknowledged TCP segment – known as fast retransmit.

If there are more than one lost segments reported in the SACK block [23], then the TCP sender will retransmit the remaining lost segments if the pipe is less than the congestion window, i.e., pipe < CWnd, where pipe is equal to FlightSize [24] (The amount of data that has been sent but not yet cumulatively acknowledged) minus the amount of out-of-sequence data reported as received by SACK blocks. The loss recovery phase ends when the sender’s highest sequence
number at the time of entering the loss recovery phase is cumulatively acknowledged or when a timeout occurs.

After the TCP sender retransmits all lost segments reported by SACK blocks, it will transmit new TCP segments if both of the following conditions are met:

a) the amount of inflight data is less than the congestion window, i.e., pipe < CWnd;

b) the highest sequence number transmitted is less than the limit set by the receiver advertised window (AWnd).

Therefore, the TCP sender might be prevented from sending new TCP segments if the AWnd becomes the bottleneck. As discussed in Section 1, the AWnd is increasingly becoming the bottleneck.

3.2 Rate-Halving
Rate-Halving (RH) [22] is the default TCP fast recovery algorithm in Linux before kernel version 3.2, which still dominates the Linux kernel deployed in today’s Internet servers. Thus, its loss recovery algorithm impacts a major portion of the Internet traffic.

The RH algorithm can be described in the following five steps:

a) The Linux TCP sender enters the loss recovery phase as soon as one out-of-order segment is guaranteed to be lost based on SACK [23] blocks and FACK [24] instead of receiving dupthresh DUPACK, e.g., dupthresh = 3, described in RFC3517 [21]. As we only consider the loss recovery phase, the algorithm to determine whether to enter the loss recovery phase is out of the scope of this paper. Interested readers can refer to [25] for details.

b) Then it sets ssthresh to equal to \( \beta \times CWnd \), where \( \beta \) depends on the TCP variants used, e.g., \( \beta = 0.7 \) for TCP CUBIC, \( \beta = 0.5 \) for TCP Reno, \( \beta = 0.8 \) for TCP Veno if the loss is determined to be non-congestive, otherwise, \( \beta = 0.5 \). TCP Westwood is designed to avoid the blind ssthresh reduction due to random loss hence it sets ssthresh to equal to the product of the estimated bandwidth and the minimum RTT.

c) The CWnd size is set to pipe + 1 if pipe + 1 < CWnd, where pipe is the amount of data outstanding in the network. This is designed to prevent transmission bursts (e.g., when CWnd–pipe is large) by limiting the sender to transmit at most one packet until a new ACK arrives.

d) If CWnd > ssthresh, TCP sender will reduce CWnd by one for every two ACKs received until CWnd = ssthresh – this is known as rate-halving and is designed to enable the sender to retransmit/transmit lost/new segments earlier and to space out the transmissions [22]. If CWnd < ssthresh, CWnd is limited by pipe + 1 (as shown in step c) and cannot increase to ssthresh, which makes it possible to underutilize the bandwidth during the loss recovery phase as the target CWnd, ssthresh, is not satisfied.

e) The loss recovery phase ends when the sender’s highest sequence number at the time of entering the loss recovery phase is cumulatively acknowledged.

When the TCP sender has retransmitted all lost segments reported by SACK blocks, it will transmit new TCP segments when the above two conditions, i.e., (a) and (b) in Section 3.1, are met.

3.3 Proportional Rate Reduction
Proportional Rate Reduction (PRR) [15] has been introduced as the default TCP loss recovery algorithm in Linux kernel since version 3.2. The PRR differs from RH in two ways:

a) If CWnd > ssthresh, the TCP sender reduces CWnd by \((1 - \beta)\) for every received DUPACK/ACK until CWnd = ssthresh – this is known as proportional reduction. Like RH, PRR converges to the target CWnd, ssthresh, determined according to the used TCP variants, but at a higher convergence rate.

b) Similar to RH, PRR sets CWnd = pipe + 1 if pipe < CWnd due to heavy packet losses [22]. Different from RH that maintains CWnd at pipe + 1 even if CWnd < ssthresh, PRR updates CWnd in the same manner as the TCP slow start phase, increasing CWnd by 1 for every received DUPACK/ACK until CWnd = ssthresh, which helps achieve the better bandwidth utilization during the loss recovery phase compared to RH.

Similarly, the PRR sender also complies with the same two conditions, i.e., (a) and (b) in Section 3.1, as in RH and standard TCP loss recovery algorithm, when transmitting new segments. Therefore, PRR suffers from the same AWnd bottleneck as RH and standard TCP loss recovery algorithm.

3.4 Queue Length Adaptive Loss Recovery Algorithm
TCP Veno continuously estimates the queue length to differentiate random losses from congestion losses. Let \( Q_i \) denote the queue length after receiving the \( i^{th} \) ACK/DUPACK, and \( Q_T \) denote the target queue length. Upon a loss event, TCP Veno treats that loss as a random loss and sets ssthresh to 0.8CWnd if \( Q_i \leq Q_T \), because it considers the network is not congestive. Otherwise (i.e., \( Q_i > Q_T \)), TCP Veno treats that loss as a congestion loss and then sets ssthresh to 0.5CWnd. But its CWnd is still governed by the loss recovery algorithms used, e.g., RH and PRR, during the loss recovery phase.

To alleviate this congestion control bottleneck during the loss recovery phase, Leong et al. [7], Liu and Lee [8], and Ren and Lin [4] developed a queue length congestion control algorithm that estimates the backlog at the bottleneck link and limits it to a target queue length by adjusting CWnd or the transmission rate accordingly during the whole transmission including the loss recovery phase. Hence, its sending rate/CWnd is only affected by the queue length and thus packet losses are decoupled from CWnd or rate controls.

As we cannot find the source code of the current queue length adaptive loss recovery implementation, we develop a Queue length Adaptive Rate Reduction algorithm (QARR) in Linux kernel based on the similar idea. The QARR is different from RH and PRR in the following three ways:

a) QARR does not employ ssthresh, thus its CWnd will not converge to ssthresh but is only affected by the estimated queue length.

b) After entering the loss recovery phase, QARR adopts the Vegas/Veno method to estimate the queue
length for every received ACK/DUPACK. Let $RTT_i$ denote the RTT measured after receiving the $i^{th}$ ACK/DUPACK using the TCP timestamp option appended in the received ACK/DUPACK’s TCP header; $\text{pipe}_i$ denote the number of inflight packets after receiving the $i^{th}$ ACK and $\text{base}RTT_i$ denote the minimum RTT during the transmission. Hence $Q_i$ can be estimated as follows:

$$Q_i = \text{pipe}_i \times \frac{(RTT_i - \text{base}RTT_i)}{RTT_i}.$$  \hfill (1)

c) Then QARR regulates the queue length around a target queue length by adjusting CWnd. Let $W_i$ denote the CWnd after receiving the $i^{th}$ ACK/DUPACK, then $W_i$ is set as follows:

$$W_i = W_{i-1} - \max(Q_i - Q_T, 0).$$  \hfill (2)

QARR complies with conditions (a) and (b) in Section 3.1 just like RH and PRR and hence suffers from potential AWnd bottleneck.

### 3.5 Bandwidth Adaptive Loss Recovery Algorithm

TCP Westwood continuously estimates the bandwidth to differentiate mobile networks’ random packet losses from congestion losses. Let $B_i$ denote the estimated receiving bandwidth after receiving the $i^{th}$ ACK/DUPACK. TCP Westwood sets its $ssthresh$ to $B_i \times RTT_{min}$, i.e., the effective bandwidth delay product (BDP), after the loss recovery phase. Similar to TCP Veno, its CWnd is also governed by RH and PRR during the loss recovery phase. To address the congestion control bottleneck (CWnd growth and shrink) in RH and PRR, previous works [3, 5], [6], [9], [10], [11] proposed to decouple the packet loss recovery from the congestion rate control, so CWnd/the transmission rate during the loss recovery phase, in this case, is only governed by the estimated bandwidth. We also develop a Bandwidth Adaptive Rate Reduction algorithm (BARR) in Linux kernel based on the similar idea. BARR is different from RH and PRR in the following ways:

a) BARR does not employ $ssthresh$, thus CWnd will not converge to $ssthresh$ but is only affected by the estimated bandwidth.

b) BARR continuously performs an estimate of the available bandwidth by measuring the average rate of returning ACKs. Specifically, let $t_i$ be the arrival time of ACK $i$ with acknowledged sequence number $ack_i$. Then for a positive integer $M$, $B_i$ is computed from

$$B_i = \frac{ack_{i+M} - ack_i}{t_{i+M} - t_i},$$  \hfill (3)

where the numerator is the amount of data received by the receiver during the time interval $(t_i, t_{i+M}]$. The parameter $M$ controls the duration of the estimation interval (in number of ACKs) and can be adjusted to a tradeoff between accuracy and timeliness of rate estimations.

c) BARR sets $W_i$ at the $i^{th}$ received ACK according to $B_i$ in (3). Thus, $W_i$ is computed from

$$W_i = B_i \times RTT_i.$$  \hfill (4)

Similarly, BARRs in previous works also suffer from the AWnd bottleneck i.e., condition (a) and (b) in Section 3.1, as in RH and PRR.

### 4 System Model

In this section, we develop a system model to investigate the efficiency of the five loss recovery algorithms reviewed in Section 3. Specifically, we are interested in the efficiency at which these loss recovery algorithms utilize the available link bandwidth during the loss recovery phase, and quantifies the impact of various system parameters, most notably the AWnd, on the link utilization efficiency.

We consider a system where a TCP sender (e.g., an Internet server) transmits data over a mobile network to a TCP receiver (e.g., mobile handset), as shown in Fig. 1.

We assume the mobile network is the bottleneck link and that link maintains a queue for packets to be transmitted to the receiver. To efficiently utilize the bottleneck link the queue must not become empty. Hence our goal is to quantify the amount of time the queue is empty during the loss recovery phase – link idle time.

In the following we consider a single TCP flow where:

a) it lasts longer than a loss recovery phase;

b) all TCP segments are of the same size;

c) the bottleneck link bandwidth remains constant during the loss recovery phase, let $C$ denote the bandwidth during the loss recovery phase (in packets per second);

d) the RTT in the system model includes propagation delay, transmission delay and queuing delay, we assume that propagation delay remains constant during the loss recovery phase and let $U$ denote the propagation delay;

e) TCP flow is not source-limited, i.e., the sender sends as much data as CWnd and AWnd allow;

f) the receiver’s receiving window size is not smaller than the network’s bandwidth propagation delay product. Let $AW$ denote the AWnd size, thus $AW \geq CU$;

Furthermore, at the time the TCP flow enters the loss recovery phase, we assume that:

g) the AWnd size remains constant during the loss recovery phase;

h) the growth of the CWnd size is limited by $AW$, let $CW$ denote the CWnd size;

After entering the loss recovery phase, the sender retransmits the first lost packet reported in the SACK blocks. Further DUPACKs from the receiver will trigger the retransmissions of the remaining lost packets. Note that whenever a retransmitted packet is successfully recovered,
it will enable the sender to transmit a new TCP segment once all lost packets have been retransmitted.

In the following analysis, we focus on the loss recovery period from the time instant the first retransmitted packet is sent by the bottleneck link ($T_2$ in Fig. 1), to the time instant the first new packet is sent by the bottleneck link (the time after or at $T_3$ in Fig. 1). In particular, we first derive the minimum link idle time $I_{\text{min}}$, where the minimum link idle time is achieved when the lost packets are consecutive $-$ best case scenario (because all the lost packets in this case are decoded before $T_2$ and thus can be retransmitted back to back). Then we use it to compute the maximum bandwidth utilization, denoted by $\eta$, for each of the five loss recovery algorithms, which measures the proportion of available bandwidth utilized for packet transmission under the best-case scenario.

### 4.1 Standard TCP Loss Recovery Algorithm

The minimum link idle time for the standard TCP loss recovery algorithm [22] is stated in Theorem 1 below.

**Theorem 1.** The minimum link idle time for the standard TCP loss recovery algorithm in the loss recovery period of $n$ lost packets is given by:

$$I_{\text{min}} = \begin{cases} \max\{U - (n - 1)/C, 0\} & n < \lfloor 0.5AW \rfloor \\ \max\{U - (\lfloor 0.5AW \rfloor - 1)/C, 0\} & n \geq \lfloor 0.5AW \rfloor. \end{cases}$$

(5)

**Proof.** Let $n$ denote the number of the lost TCP segments in a loss recovery period. Consider the period from $T_2$ to $T_5$ as depicted in Fig. 1. Ignoring processing delay the duration from $T_2$ to $T_5$ is equal to one round-trip propagation delay, i.e., $U$ seconds. As the sender must retransmit all lost packets before transmitting a new TCP segment at $T_5$, it can at most send out $n - 1$ retransmitted packets during this interval, subject to the CW. Note that upon entering the loss recovery phase TCP will set $CW = \lfloor 0.5AW \rfloor$. Next, we consider two cases:

**Case 1:** $n < CW = \lfloor 0.5AW \rfloor$: In this case, the sender can send out at most $n - 1$ retransmitted packets during the interval $[T_2, T_5]$, with a total transmission time of $(n - 1)/C$. Thus the minimum link idle time can then be computed from

$$I_{\text{min}} = \max\{U - (n - 1)/C, 0\}. \quad (6)$$

**Case 2:** $n \geq CW = \lfloor 0.5AW \rfloor$: In this case, the sender can retransmit at most $\lfloor 0.5AW \rfloor$ lost packets subject to the CW;nd constraint. Therefore, the corresponding minimum link idle time is given by

$$I_{\text{min}} = \max\{U - (\lfloor 0.5AW \rfloor - 1)/C, 0\}. \quad (7)$$

In the best case scenario there is no waiting time for retransmitting the rest of lost packets and the 1st new packet after $T_5$, so (7) gives the overall minimum link idle time and the result follows.

$$I_{\text{min}} = \begin{cases} \max\{U - (n - 1)/C, 0\}, & n \leq \lfloor 0.5AW \rfloor \\ \max\left\{\min\left\{\beta AW, (AW - n)/2\right\} > n, \left(U - (\min\{\beta AW, (AW - n)/2\} - 1)/C, 0\right)\right\} & n > 2(\lfloor 0.5AW \rfloor) \quad (10) \end{cases}$$

With the minimum link idle time, we can then compute the maximum bandwidth utilization from

$$\eta = n/(CI_{\text{min}} + n), \quad (8)$$

where the numerator is the amount of data transmitted (i.e., $n - 1$ retransmitted packets and 1 new packet) and the denominator is the total amount of data that could be transmitted (i.e., amount transmitted plus the amount which would have been transmitted during the link idle time) during the loss recovery period.

### 4.2 Rate-Halving

The minimum link idle time for rate-halving loss recovery algorithm is stated in Theorem 1 below.

**Theorem 2.** The minimum link idle time for RH’s loss recovery algorithm is given by:

$$I_{\text{min}} = \begin{cases} \max\{U - (n - 1)/C, 0\}, & n \leq \lfloor 0.5AW \rfloor \\ \max\{U - (n)/2 - (n - 1)/C, 0\}, & n > \lfloor 0.5AW \rfloor. \quad (9) \end{cases}$$

**Proof.** The proof is detailed in Appendix I.A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/2909888.

Similarly, the maximum bandwidth utilization can be computed from (8).

### 4.3 Proportional Rate Reduction

The minimum link idle time for PRR loss recovery algorithm is stated in Theorem 2 below.

**Theorem 3.** The minimum link idle time for PRR’s loss recovery algorithm is given by:

$$I_{\text{min}} = \max\{U - (n - 1)/C, 0\}. \quad (10)$$

**Proof.** The proof is detailed in Appendix I.B, available in the online supplementary material.

Similarly, the maximum bandwidth utilization can be computed from (8).

### 4.4 Queue Length Rate Adaptive Reduction

The minimum link idle time for QARR loss recovery algorithm is stated in Theorem 4 below.

**Theorem 4.** The minimum link idle time for QARR’s loss recovery algorithm is given by:

$$I_{\text{min}} = \max\{U - (n - 1)/C, 0\}. \quad (11)$$

**Proof.** The proof is detailed in Appendix I.C, available in the online supplementary material.

Similarly, the maximum bandwidth utilization can be computed from (8).
4.5 Bandwidth Adaptive Rate Reduction

Similar to QARR, BARR loss recovery algorithm has the same minimum link idle time, which is stated in Theorem 5 below.

**Theorem 5.** The minimum link idle time for BARR's loss recovery algorithm is given by:

\[
I_{\text{min}} = \max\{U - (n - 1)/C, 0\}. \tag{12}
\]

**Proof.** The proof is detailed in Appendix I.D, available in the online supplementary material.

Similarly, the maximum bandwidth utilization can be computed from (8).

4.6 Discussions

The previous analysis reveals two properties of the five loss recovery algorithms. First, in a network with high bandwidth (i.e., C) and long delay (i.e., U), the link will be more likely to become idle during the loss recovery phase (c.f. (5), (9), (10), (11) and (12)). Current 3G/HSPA+ networks have a typical bandwidth of 20 Mbps and a delay of 100 ms. Within a propagation delay, i.e., U, the network has sufficient bandwidth to send 20 Mbps × 100 ms/8 bytes = 250 KB data. With a typical packet size of around 1.5 KB it is clear that the link will likely become idle during loss recovery unless the loss event comprises a burst of over 167 lost packets. Moreover, the deficiency increases with higher bandwidth and thus will become even more significant in the emerging LTE/4G and 5G networks. In the next section, we propose a novel opportunistic retransmission algorithm to tackle this challenge.

5 OPPORTUNISTIC RETRANSMISSION

Liu and Lee [3], [26] proposed opportunistic transmission to tackle the flow control bottleneck in mobile networks with a large bandwidth-delay product (BDP), especially for networks with BDP larger than the receiver advertised window (AWnd). The motivation is that TCP’s flow control algorithm was designed to prevent fast senders from overflowing slow receivers. However, rapid advances in processors have equipped modern day receivers such as PCs and smartphones with very high processing power, and thus the ability to process incoming packets at very high data rates. As a result, the AWnd size reported by the receiver is often kept at the maximum value, as few packets require extensive buffering to wait for processing.

Opportunistic transmission exploits the receiver’s processing power by allowing the sender to transmit packets beyond the maximum sequence number allowed by AWnd. In practice, by the time the out-of-AWnd packets arrive at the receiver, the previously received packets would have been processed already, thus allowing the receiver to receive them without buffer overflow. But opportunistic transmission was only designed for TCP’s normal phase of operation, i.e., when there is no packet loss occurred. Therefore, we extend the idea to TCP’s loss recovery phase – opportunistic retransmission.

Specifically, as shown in Fig. 1, the TCP sender has to wait for a cumulative ACK with an increase in the highest sequence number acknowledged after retransmitting the lost packets due to AWnd constraint. If the lost packets are recovered successfully, the application at the receiver can process the packets quickly and free up the receiving window/buffer immediately for new packet arrivals if the receiver has sufficient processing power, e.g., at time T₃ in Fig. 1. However, the sender has to wait for the ACK acknowledging the first retransmitted packet to return before it can begin transmitting a new packet at time T₄ in Fig. 1, resulting in the link idle time analyzed in Section 4.

To tackle this problem, we propose to relax the AWnd constraint during TCP loss recovery phase by allowing transmitting new packets beyond the AWnd constraint, e.g., transmit new packets during [T₁, T₄] in Fig. 1, irrespective of the AWnd. However, this must be performed judiciously as the receiver is temporarily unable to clear its buffer (by passing data to the application) until the head-of-line lost packet is successfully retransmitted. Thus, opportunistic retransmission operates according to the following three ways:

a) For every received DUPACK the TCP sender decodes the SACK blocks carried inside DUPACK to determine (i) the number of gaps (i.e., lost packets) at the receiver’s receiving buffer, denoted by n₁; and (ii) the number of out-of-order packets received, denoted by n₂.

b) Then the TCP sender first retransmits the lost packets. The receiver will advance the AWnd by n₁ + n₂ packets if they are successfully received. Therefore the TCP sender then transmits up to n₁ + n₂ new packets.

c) But the number of transmitted packets is also subject to the congestion control constraint: pipe ≤ CWDₙ.

In the following we apply this opportunistic retransmission algorithm to the five loss recovery algorithms and derive their minimum link idle time and maximum bandwidth utilization.

5.1 Performance Analysis

Opportunistic retransmission enables the five loss recovery algorithms to remove constraint (b) in Section 3. The following five theorems show the minimum link idle time when using opportunistic retransmission.

**Theorem 6.** The minimum link idle time for standard loss recovery with opportunistic retransmission is given by:

\[
I_{\text{min}} = \max\{U - ([0.5AW] - 1)/C, 0\}. \tag{13}
\]

**Proof.** The proof is detailed in Appendix I.E, available in the online supplementary material.

Theorem 7 below states the minimum link idle time for rate-halving loss recovery algorithm with opportunistic retransmission.

**Theorem 7.** The minimum link idle time for rate-halving loss recovery with opportunistic retransmission is given by:

\[
I_{\text{min}} = \begin{cases} 
\max\{U - ([AW] - 1)/C, 0\}, & n \leq AW - [AW] \\
\max\{U - (AW - n - 1)/C, 0\}, & AW - [AW] < n \leq AW - n \\
\max\{n/(AW - n)\}, & (U - (AW - n - 1)/C, 0), n > AW - n.
\end{cases} \tag{14}
\]

**Proof.** The proof is detailed in Appendix I.F, available in the online supplementary material.
The minimum link idle time for proportional rate reduction loss recovery with opportunistic retransmission is given by:

Theorem 8. The minimum link idle time for proportional rate reduction loss recovery with opportunistic retransmission is given by:

$$I_{\text{min}} = \frac{\max\{1 - (\beta_{\text{AW}} - 1)/C, 0\}}{\min\{AW_{I}, (AW - n)2^{4}\}}$$

where $AW_I$ is the maximum burst size of lost packets.

Theorem 9. Both queue length adaptive rate reduction and bandwidth adaptive rate reduction loss recovery algorithms with opportunistic retransmission will not result in any link idle time, thus $I_{\text{min}} = 0$.

Theorem 9 below states the minimum link idle time for proportional rate reduction loss recovery with opportunistic retransmission.

The proof is detailed in Appendix I.G, available in the online supplementary material.

Additionally, opportunistic retransmission can work based on the assumption that the mobile device has sufficient processing power, e.g., Apple A7 has a 64bit 1.3-1.4 GHz dual-core CPU [27]. To experimentally validate this assumption, we use cpulimit [28] to restrict the CPU utilization of a TCP process/application (e.g., iperf). We found that opportunistic retransmission still can work even if CPU utilization of the TCP process is limited to 10 percent.

5.2 Performance Validation

Using the system model we can evaluate the bandwidth utilization for those five loss recovery algorithms, and investigate the potential improvements achieved by using opportunistic retransmission. First, we compute the numerical results using the system model. Then, we verify the system model by experiments. Specifically, we adopt the typical system parameters of 3G/HSPA+ networks in the experimental setup. The system parameters are summarized in Table 3 and the experimental setup is shown in Fig. 2 (c.f., Section 8.1). We capture the TCP trace at the sender to measure the actual link idle time and thus derive the bandwidth utilization during the loss recovery period. This enables us to verify the system model by comparing the experimental results to the numerical ones. As the standard TCP loss recovery was not implemented by the major operating systems we did not validate it by experiments.

To model loss events we test different numbers of consecutive packet losses, ranging from 10 to 100, to simulate the loss events under different network setting and radio signal conditions. Table 1 evaluates the impact of the burst size of packet loss events, ranging from 10 to 100 packets, on bandwidth utilization achieved over the loss recovery phase. We first observe that the numerical results computed from the system model are quite consistent with the experimental results for original TCP loss recovery algorithms under different loss rates, and the bandwidth utilization increase for larger burst size – a direct result of longer link utilization time to retransmit the larger number of lost packets, e.g., the bandwidth utilizations of QARR increase from 0.057 to over 0.51 with the increase in burst size of lost packets. However, the numerical results from the system model do not match the experimental results for TCP loss recovery algorithms with opportunistic retransmission under small burst size, e.g., 0.246 versus 1 for QARR under 10 burst size of lost packets. The similar observation can be found in Table 2 that compares the average bandwidth utilization over the loss recovery phase under different propagation delays ranging from 50 ms to 200 ms, when 10 consecutive packet loss event occurs, the numerical results are consistent with the experimental results for the original loss recovery algorithms under all the propagation delays, while they are inconsistent after applying opportunistic retransmission. Interestingly, the throughputs of the existing loss recovery algorithms are improved by opportunistic

### Table 1: Comparison of Bandwidth Utilization During the Loss Recovery Period versus Different Packet Loss Burst Sizes

| Loss recovery algorithm | Burst size (packets) | 10  | 50  | 100 |
|-------------------------|----------------------|-----|-----|-----|
| RH                      | Original             | 0.054 | 0.294 | 0.474 |
|                         | Experimental         | 0.057 | 0.286 | 0.513 |
| w/ OR                   | Original             | 0.75  | 0.75 | 0.474 |
|                         | Experimental         | 0.485 | 0.737 | 0.437 |
| PRR                     | Original             | 0.054 | 0.294 | 0.594 |
|                         | Experimental         | 0.057 | 0.286 | 0.571 |
| QARR                    | Original             | 0.054 | 0.294 | 0.594 |
|                         | Experimental         | 0.057 | 0.286 | 0.513 |
| BARR                    | Original             | 0.75  | 0.75 | 0.75 |
|                         | Experimental         | 0.383 | 0.703 | 0.731 |

Fig. 2. Experiments testbed setup.
retransmission but not as much as shown in our system model. The gap between the numerical results and the experimental results is larger with the smaller burst size of lost packets. After investigating the data trace, we found that application stall (where TCP sender runs out of packets) always occurs during loss recovery phase. Hence, TCP sender has no new packet to transmit opportunistically, resulting in the degradation on the bandwidth utilization. Our experiments show that application stall occurs during over 61.1 percent of the loss recovery phase. In the next section, we refine the sender’s sndbuf growth mechanism to prevent application stall from occurring.

6 APPLICATION STALL

Application stall can occur due to (a) application has no new data to send, and (b) the inefficiency of the current sending buffer growth mechanism. For (a) the sender does not always have data to send, which violates the assumption in Section 4 that TCP flow is not source-limited, hence we only consider (b). We first define application stall as follows:

Definition 1. In a TCP data flow, the TCP sender cannot timely move the application data from the application layer to the transport layer, resulting in the transport layer not having sufficient data to transmit.

In the following sections, we first investigate the TCP sender’s sending process in current Linux TCP/IP stack and found that the application stall is mainly caused by the memory management of the sending buffer, i.e., sndbuf. Therefore, we first introduce the use of the sndbuf and its allocation strategy, and why it could result in the application stall problem. Then, to address this problem, we propose a refined sending buffer growth mechanism.

6.1 Linux TCP sndbuf Memory Management

When the user-space application requests to send data, the function tcp_sendmsg() is called in the kernel-space, which is responsible for passing data from the application layer to the transport layer only if the transport layer has enough memory to store that data. Since TCP is reliable, all sent data must be stored in the sndbuf before being sequentially acknowledged. Therefore, tcp_sendmsg() first checks whether there is enough memory to store that data by calling tcp_memory_free(). If no enough available memory, it adds SOCK_NOSPACE flag to the socket and goes into the sleep state until memory is available. If there is enough memory, it allocates a socket buffer (SKB, Socket Buffer) for that data and passes that data to the transport layer, i.e., TCP, where SKB is the smallest unit in managing data. When a new TCP packet (e.g., ACK) is received by the TCP sender, tcp_clean_rtx_queue() is called to clear the data in the sndbuf. If the SOCK_NOSPACE flag was previously set, function tcp_write_space() is called to wake up the process in the sleep state and continue to pass the application data to the sndbuf. Therefore, the process is awakened up only if the free space of the sndbuf is greater or equal to the 1/3 of the total space. This setting can be the primary factor resulting in application stall, which will be explained in the next sections.

6.2 Current Sending Buffer Growth Mechanism

All the transmitted/transmitting but not acknowledged segments will be stored in the sndbuf and will not be freed unless they are acknowledged sequentially. In Linux TCP, the size of sndbuf is not fixed but dynamically configured. The growth mechanism of sndbuf can be summarized in the following three ways:

a) The size of sndbuf is affected by the parameters in file tcp_wmem in the directory /proc/sys/net/tcp/wmem. It contains three values: the minimum sndbuf size, snd_wmem_min, the default sndbuf size, snd_wmem_default, and the maximum sndbuf size, snd_wmem_max. The sndbuf size is initialized to be snd_wmem_default, e.g., 16 KB, in Centos 5.5.

b) sndbuf increases multiplicatively according to the CWnd size and the increasing factor is 2, which makes sndbuf = 2CW at all times. sndbuf increases in this manner until CWnd stops increasing, being limited by AWnd, thus at this steady state, sndbuf size is twice of the current CWnd.

c) The segments occupied in the sndbuf will be freed if they are acknowledged sequentially, resulting the free space (free space) in the sndbuf. Once free space > sndbuf/3, sndbuf will move forward for free space to fetch new data segments from application to fill it up.

However, this sndbuf growth mechanism cannot completely prevent application stall from occurring, especially during the loss recovery phase, with opportunistic retransmission. When free space ≤ sndbuf/3, the application cannot send any data,
i.e., application stall occurs, which could lead to two problems: (a) no packet is transmitted when the free space is released, resulting in the bandwidth inefficiency; (b) when the new data packets are passed to sndbuf, they will be sent in burst, which affects the accuracy of the queueing delay/RTT estimations in rate controls [4, 7, 8].

Hence, we have the following theorem.

**Theorem 10.** Let \( b_{\text{free}} \) denote the free space size of the sndbuf. Assuming the sndbuf is in the steady state and AWnd is the bottleneck, with opportunistic retransmission, application stall will not occur during TCP loss recovery phase if \( b_{\text{free}} \leq (1 - \beta)CW \) or \( b_{\text{free}} > 2CW/3 \). Therefore, if \( (1 - \beta)CW < b_{\text{free}} \leq 2CW/3 \), application stall occurs during the TCP loss recovery phase.

**Proof.** Let \( b_{\text{snd}} \) denote the sndbuf size. Since the sndbuf is in the steady state, we have \( b_{\text{snd}} = 2CW \). Let \( b_{\text{occ}} \) denote the size of the sndbuf being occupied by the data, thus \( b_{\text{occ}} = b_{\text{snd}} - b_{\text{free}} \). During the loss recovery phase TCP sender can at most transmit \( \betaCW \) new segments using opportunistic retransmission. To satisfy it we need

\[
b_{\text{occ}} = b_{\text{snd}} - b_{\text{free}} \geq (1 + \beta)CW,
\]

(16)

thus \( b_{\text{free}} \leq (1 - \beta)CW \).

As shown in (c) if \( b_{\text{free}} > b_{\text{occ}}/3 \), sndbuf will advance free_space to fetch new data to fill it up, thus \( b_{\text{occ}} = b_{\text{occ}} \) and \( b_{\text{free}} = 0 \) after fetching the data. As \( b_{\text{occ}} = b_{\text{occ}} \geq (1 + \beta)CW \), TCP sender also can at most transmit \( \betaCW \) new segments if

\[
b_{\text{free}} > b_{\text{occ}}/3 = 2CW/3.
\]

Therefore, application stall will not occur if either (16) or (17) is satisfied, and application stall can occur if \((1 - \beta)CW < b_{\text{free}} \leq 2CW/3\).

The reduction factor \( \beta \) affects the period of the application stall. For TCP CUBIC and PRR combination, \( \beta = 0.7 \), application stall can occur if \( 0.3CW < b_{\text{free}} \leq 2CW/3 \), thus \((2/3 - 0.3)/(2/3) = 55\%\) of the time application stall can occur theoretically, while 61.6% of the time application stall occurs from our measurements. Similarly, for TCP CUBIC and QARR/BARR combinations, \( \beta = 1 \), application stall can occur if \( 0 < b_{\text{free}} \leq 2CW/3 \), thus 100 percent of the time the application stall can occur during the loss recovery phase, which is also validated in Section 5.2.

### 6.3 Refined Sending Buffer Growth Mechanism

Intuitively, we can remove the constraint in (c) in Section 5.2, that is, once free_space is available sndbuf will advance and fetch new data from application to fill it up. However, this method might lead to extra scheduling delay, as we let sndbuf to advance when free_space reaches 1/9 of sndbuf and found that the CPU utilization increases 2 times for one flow, hence we solve it in another way.

We modify the sndbuf increasing factor from 2 to 3, i.e., \( \text{sndbuf} = 3 \times \text{current CWnd size} \), and we can show that 3 is the minimum increasing factor that guarantees application stall would not occur during loss recovery phase, thus we have the following theorem.

**Theorem 11.** Application stall will not occur if sndbuf increases multiplicatively and the increasing factor is at least 3, thus \( b_{\text{snd}} = 3CW \).

| Loss recovery algorithm | Burst size (packets) |
|-------------------------|---------------------|
| RH w/ the proposed optimization algorithms | Numerical | 0.75 | 0.75 | 0.47 |
| PRR w/ the proposed optimization algorithms | Numerical | 0.75 | 0.75 | 0.75 |
| QARR w/ the proposed optimization algorithms | Numerical | 1 | 1 | 1 |
| BARR w/ the proposed optimization algorithms | Numerical | 0.97 | 0.98 | 0.96 |

| Loss recovery algorithm | Burst size (packets) |
|-------------------------|---------------------|
| RH w/ the proposed optimization algorithms | Experimental | 0.74 | 0.74 | 0.51 |
| PRR w/ the proposed optimization algorithms | Experimental | 0.74 | 0.74 | 0.74 |
| QARR w/ the proposed optimization algorithms | Experimental | 0.96 | 0.98 | 0.95 |
| BARR w/ the proposed optimization algorithms | Experimental | 0.97 | 0.98 | 0.96 |

**Comparison of Bandwidth Utilization During the Loss Recovery Period versus Different Loss**

| Loss recovery algorithm | Burst size (packets) |
|-------------------------|---------------------|
| RH w/ the proposed optimization algorithms | Numerical | 0.75 | 0.75 | 0.47 |
| PRR w/ the proposed optimization algorithms | Numerical | 0.75 | 0.75 | 0.75 |
| QARR w/ the proposed optimization algorithms | Numerical | 1 | 1 | 1 |
| BARR w/ the proposed optimization algorithms | Numerical | 0.97 | 0.98 | 0.96 |

### 6.4 Performance Validation

With the refined sending buffer growth mechanism, we revalidate the system model in Section 5.1 using the same experimental setup. As shown in Table 4 and Table 5, the numerical results computed from the system model become consistent with the experimental results after applying the proposed optimizations, i.e., opportunistic retransmission and refined sending buffer growth mechanism, which validates the correctness of our model and the efficiency of the proposed optimization algorithms. Note that the small gap from the 100 percent utilization for QARR is due to scheduling delay in the both sender and receiver’s kernel according to our preliminary investigation, as our model assumes zero processing delay, which warrants further research.

### 7 Implementation

In this section, we implement the optimization algorithms, i.e., opportunistic retransmission and refined sndbuf growth mechanism, in the current Linux kernel. To facilitate the comparative evaluation of the existing loss recovery algorithms w/ and w/o the proposed optimization algorithms and the future practical deployment, we modularized the
loss recovery algorithms so that those loss recovery algorithms can be implemented and switched in-and-out easily without recompiling the kernel. Specifically, we implemented all loss recovery algorithms, i.e., RH, PRR, QARR and BARR, in a separate kernel module respectively. The module handler interfaces are defined in `struct tcp_retrans_ops`, which has the similar implementation as the pluggable TCP congestion control module handler interfaces [29]. The `struct tcp_retrans_ops` is defined as follows:

```c
struct tcp_retrans_ops {
  struct list_head list_head;
  /* initialize private data when entry CWR or Recovery mode (required) */
  void (*init_cwnd_reduction)(struct sock *sk, const bool set_ssthresh);
  /* change cwnd based on packets newly delivered (required) */
  void (*cwnd_reduction)(struct sock *sk, int newly_acked_sacked, int fast_rexmit);
  /* cleanup when exiting CWR or Recovery mode (required) */
  void (*end_cwnd_reduction)(struct sock *sk);
  /* hooker for packet ack accounting (optional) */
  void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);
  /* opportunistic retransmission is handled here */
  u32 (*get_awnd_extension)(const struct tcp_sock *tp, char name[TCP_RETRANS_NAME_MAX]);
  struct module *owner;
};
```

In this `struct`, every loss recovery algorithm mainly requires two components: (1) the initialization of TCP parameters, e.g., `ssthresh`, when entering/exiting from the loss recovery phase, which are defined in the handlers of `init_cwnd_reduction` and `end_cwnd_reduction`; (2) the handler for each received ACK/DUPACK during the loss recovery phase, which is defined in `cwnd_reduction`. The loss recovery algorithm can enable/disable opportunistic retransmission by defining handler `get_awnd_extension`. We also introduce the `proc` interfaces used for `sndbuf` management and performance tunings: (a) `snd_buffer_grow_factor`: the `sndbuf` increasing factor, 3 by default; (b) `queue_len_threshold`: the target queue length in QARR, 5 by default (i.e., \(Q_t = 5\)); (c) `slide_cwnd_size`: the sliding window size for bandwidth estimations in BARR, 200 by default (i.e., \(M = 200\)).

8 PERFORMANCE EVALUATIONS

In this section, we evaluate the performance impact of the proposed optimization algorithms on the overall TCP throughput in the presence of wide network settings, instead of only focusing on the throughput performance over the TCP loss recovery phase under the loss event of consecutive packet losses.

8.1 Network Testbed Setup

We setup a testbed as depicted in Fig. 2. We employ an emulator rather than conducting experiments over a production of mobile networks as it is not feasible to control packet loss events, therefore making consistent performance comparisons difficult. To emulate various network behaviors in terms of the bandwidth, loss rate and delay, we develop a custom emulator using DPDK [30], a set of libraries and drivers for fast packet processing, thus confining the outgoing rate at the short time scale without introducing packet bursts greater than the expected one.

In configuring the emulator we adopt typical network parameters including delay, loss, and bandwidth according to the recent measurements in [3], [12], [13], [31], [32], which are summarized in Table 3 if not mentioned. Specifically, to model loss events, the emulator randomly drops the outgoing packets. Three loss rates, i.e., 0.01, 0.05 and 0.1 percent are tested in our experiments. We believe that these loss rates can cover the ones measured/estimated from production networks, which are reported in one recent measurement study [31], depending on the network configurations and radio signal conditions, including some extreme cases such as severe signal quality degradation and hands-off to another cell. To model the bandwidth, two types of mobile data networks are emulated: 3G/HSPA+ and 4G/LTE, with their capacities are set to 20 Mbps and 100 Mbps respectively according to [3], [31]. The base RTT estimated from productions networks is 150-200 ms [12], [13], [32] and 60 ms [31] in 3G and 4G respectively, thus four propagation delays, i.e., 50, 150, 200 ms and 200 ms, are tested, which cover the ones measured.

The sender host, emulation host and receiver host in Fig. 2 run Centos 5.5 Linux with kernel v.3.10 with dual Xeon E5645 2.40 GHz CPU, 32 GB RAM and 1 Gbps NICs. The peak sending throughput can reach 950 Mbps. We add the loss recovery modules into the sender’s kernel, which can switch between different loss recovery algorithms w/ and w/o the proposed optimization algorithms.

8.2 Module Verification

With the testbed, we can capture the TCP data trace and the TCP congestion control related parameters at the TCP sender, e.g., logging CWnd and `ssthresh` with `tcpprobe` module. In this section, we verify the correctness of the pluggable TCP loss recovery module. First, we compare the throughput results using the loss recovery module with the ones using the built-in implementation. We first evaluate the average throughputs of a single TCP flow under different loss rates, ranging from 0.001 to 10 percent. The congestion control and loss recovery algorithms are TCP CUBIC and PRR respectively. As shown in Fig. 3 (left), the average TCP throughputs under all the loss rates with the pluggable PRR module match the ones with the Linux built-in PRR. Second, we intentionally drop packets at five specific timings during the transmission of a TCP flow with and without using pluggable PRR module respectively.

| Loss recovery algorithm | Propagation delay (ms) | 50  | 150  | 200  |
|-------------------------|-----------------------|-----|------|------|
| RH w/ the proposed optimization algorithms | Numerical | 0.75 | 0.75 | 0.75 |
| PRR w/ the proposed optimization algorithms | Numerical | 0.75 | 0.75 | 0.75 |
| QARR w/ the proposed optimization algorithms | Numerical | 1 | 1 | 1 |
| BARR w/ the proposed optimization algorithms | Numerical | 0.99 | 0.95 | 0.98 |
network. We note that the 3.5 percent (i.e., 100%–3.5%) bandwidth degradation is primarily due to and combinations (e.g., + → draining below QARR values are much less than AWnd reported, IEEE TRANSACTIONS ON MOBILE COMPUTING, VOL. 19, NO. 6, JUNE 2020

Loss rate (%) TCP congestion controls. QARR observes it and fills that queue up by increasing net-

QARR + RH/PRR (TCP Vegas is a representative of the delay-based TCP. RH is the default loss recovery algorithm before Linux Kernel 3.2, while PRR is the default loss recovery after Linux Kernel 3.2. Both QARR and BARR are widely employed in recent proposed rate control algorithms [3], [4], [5], [6], [7], [8], [9], [10], [11].

As shown in Table 6, comparing to the no-loss case, TCP throughput degrades as the packet loss rate is increased. Even at a loss rate of 0.1 percent, which is not uncommon in mobile data networks, both TCP CUBIC+RH and TCP CUBIC+PRR's throughput drops from 17.0 Mbps to 5.2 Mbps, because TCP CUBIC blindly cuts down ssthresh and CWnd by 0.3, thus reduces the throughput. Surprise, TCP Veno, and TCP Westwood, that are optimized for mobile/wireless network to combat random loss, and TCP Vegas is designed for mobile/wireless network, still suffer from the throughput degradation due to random loss, e.g., TCP Veno+RH/PRR and TCP Westwood+RH/PRR’s throughputs dropped from 14.2 Mbps to 5.3 Mbps and 16.7 Mbps to 7.87 Mbps, respectively. After investigating the trace data we find out that they are not able to figure out an effective ssthresh/CWnd upon loss packets, leading to reducing CWnd unnecessarily. TCP Vegas+RH/PRR also reduces its CWnd/ssthresh blindly in a similar way. In summary, the throughputs of all TCP congestion controls with RH/PRR (TCP+RH/PRR) are throttled primarily by the congestion control bottleneck, e.g., CWnd/ssthresh, and their CWnds/ssthresh values are much less than AWnd reported, thus they do not suffer from the flow control bottleneck.

By contrast, all TCP + QARR/BARR combinations (e.g., TCP CUBIC+QARR/BARR) performed better than the other combinations under all loss rates but the throughput still degrades as the loss rate is increased (e.g., from 18.9 Mbps to 14.1 Mbps at 0.1 percent loss). It is worth noting that their CWnd/ssthresh are unaffected by packet losses, while the throughput degradation in TCP + QARR/BARR in the presence of packet losses is primarily resulted from the low bandwidth utilization of the loss recovery phase due to the flow control bottleneck and application stall, rather than due to the congestion control.

Next, we test the baseline TCP throughput performances of those 16 combinations under different propagation delays. The analysis in Section 4 reveals that one of the primary sources of link idle time is the waiting time for the ACK for the first retransmitted packet to return to the sender. Thus, one would expect the propagation delay to have a significant impact on bandwidth efficiency during the loss recovery phase. The results in Table 7 show that, as the propagation delay increases from 100 ms to 200 ms, for example, TCP Vegas+QARR’s throughput w/o the optimization algorithms drops from 17.3 Mbps to 11.4 Mbps. After investigating the data trace we found that, besides the flow control bottleneck and application stall, it also occasionally suffers from the congestion control bottleneck resulting from QARR algorithms. That is because the flow control bottleneck and application stall prevent QARR from transmitting new TCP packets, resulting in Qst� draining below Qr. QARR observes it and fills that queue up by increasing its CWnd, thus transmits packets in burst (c.f., Section 8.5). Those burst packets are then translated to the RTT spikes, which unnecessarily makes QARR decrease its CWnd, thus underutilize the bandwidth. Similarly, the flow control bottleneck and application stall also confuse the BARR algorithm by preventing it from transmitting new packets. Thus, BARR underestimates the bandwidth based on the insufficient ACK/DUPACKs received, which reduces its CWnd unnecessarily.

| Congestion Control | Loss recovery | Loss rate (%) |
|--------------------|---------------|---------------|
|                   |               | 0.01 | 0.05 | 0.1 |
| CUBIC             | RH            | 17.0 (88.3%) | 6.87 (34.3%) | 5.2 (26.0%) |
|                   | PRR           | 16.1 (80.3%) | 7.89 (39.5%) | 5.4 (27.1%) |
|                   | QARR          | 18.8 (94.0%) | 17.6 (88.0%) | 15.6 (78.5%) |
|                   | BARR          | 18.7 (93.5%) | 17.9 (90.5%) | 16.6 (83.0%) |
| Veno              | RH            | 14.2 (71.0%) | 6.35 (31.8%) | 4.72 (23.6%) |
|                   | PRR           | 14.1 (70.6%) | 7.20 (36.0%) | 5.3 (26.4%) |
|                   | QARR          | 18.9 (94.5%) | 16.6 (83.0%) | 14.1 (70.3%) |
|                   | BARR          | 18.8 (94.0%) | 17.9 (90.5%) | 16.5 (82.5%) |
| Westwood          | RH            | 14.6 (72.5%) | 12.6 (61.2%) | 7.87 (39.4%) |
|                   | PRR           | 16.7 (83.5%) | 9.57 (47.9%) | 8.30 (41.6%) |
|                   | QARR          | 18.6 (93.0%) | 16.8 (84.0%) | 15.3 (77.0%) |
|                   | BARR          | 18.2 (91.0%) | 16.8 (84.0%) | 16.2 (81.0%) |
| Vegas             | RH            | 13.0 (865.0%) | 6.69 (33.5%) | 4.62 (22.1%) |
|                   | PRR           | 13.6 (86.6%) | 6.07 (30.4%) | 4.36 (21.8%) |
|                   | QARR          | 18.3 (90.6%) | 16.9 (84.5%) | 14.3 (71.5%) |
|                   | BARR          | 18.7 (93.5%) | 17.6 (88.0%) | 16.2 (81.0%) |
TABLE 7  
Throughput Performance of the Existing Combinations of Congestion Control and Loss Recovery Algorithms Under Different Propagation Delays

| Congestion Control | Loss recovery | Propagation delay (ms) |
|--------------------|---------------|------------------------|
|                    | 50            | 150                    | 200                  |
| CUBIC              |               |                        |                      |
| RH                 | 6.98(34.9%)   | 4.19(20.9%)            | 3.76(18.8%)          |
| PRR                | 7.25(36.2%)   | 4.63(23.1%)            | 3.93(19.6%)          |
| QARR               | 18.00(0.0%)   | 16.10(0.5%)            | 14.7(3.5%)           |
| BARR               | 18.00(0.0%)   | 16.28(0.1%)            | 14.97(4.5%)          |
| Veno               |               |                        |                      |
| RH                 | 9.58(47.9%)   | 3.51(17.5%)            | 2.52(12.6%)          |
| PRR                | 10.35(5.1%)   | 3.22(16.1%)            | 2.38(11.9%)          |
| QARR               | 18.39(15.1%)  | 11.57(5.7%)            | 10.10(5.0%)          |
| BARR               | 18.10(15.5%)  | 15.67(8.8%)            | 14.57(2.2%)          |
| Westwood           |               |                        |                      |
| RH                 | 18.39(15.1%)  | 7.64(38.2%)            | 5.70(28.5%)          |
| PRR                | 17.70(8.5%)   | 8.68(34.3%)            | 4.92(24.6%)          |
| QARR               | 17.80(0.9%)   | 14.07(0.5%)            | 12.76(3.5%)          |
| BARR               | 17.78(8.5%)   | 14.47(2.0%)            | 13.06(5.0%)          |
| Vegas              |               |                        |                      |
| RH                 | 9.60(48.0%)   | 3.17(15.9%)            | 2.28(11.4%)          |
| PRR                | 9.85(49.2%)   | 3.03(15.1%)            | 2.31(11.5%)          |
| QARR               | 17.30(6.5%)   | 11.78(5.8%)            | 11.45(7.0%)          |
| BARR               | 17.70(8.5%)   | 14.67(8.0%)            | 14.57(2.5%)          |

TABLE 8  
Performance Impact of the Proposed Optimization Algorithms on Existing Loss Recovery Algorithms Under Different Packet Loss Rates

| Congestion Control | Loss recovery | Loss rate (%) |
|--------------------|---------------|---------------|
|                    | 0.01          | 0.05          | 0.1            |
| CUBIC              |               |               |                |
| RH                 | 17.05(8.1%)   | 7.8 (38.8%)   | 5.28(26.2%)    |
| PRR                | 16.30(5.1%)   | 9.0 (45.1%)   | 5.19(29.9%)    |
| QARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.10(5.5%)    |
| BARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.09(5.0%)    |
| Veno               |               |               |                |
| RH                 | 15.75(7.4%)   | 7.7 (38.4%)   | 5.60(28.0%)    |
| PRR                | 15.75(7.8%)   | 8.4 (41.9%)   | 6.25(31.3%)    |
| QARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.10(5.5%)    |
| BARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.09(5.5%)    |
| Westwood           |               |               |                |
| RH                 | 16.78(3.3%)   | 12.96(6.5%)   | 8.20(41.0%)    |
| PRR                | 18.39(1.6%)   | 13.46(6.9%)   | 9.09(45.5%)    |
| QARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.10(5.5%)    |
| BARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.10(5.5%)    |
| Vegas              |               |               |                |
| RH                 | 13.06(6.0%)   | 7.103(5.3%)   | 4.79(23.9%)    |
| PRR                | 14.37(1.3%)   | 6.80(4.1%)    | 4.62(24.1%)    |
| QARR               | 19.19(5.5%)   | 19.10(5.5%)   | 19.10(5.5%)    |
| BARR               | 19.19(5.5%)   | 19.10(5.5%)   | 18.90(4.5%)    |

8.4 Impact of the Optimization Algorithms on TCP Throughput Performances

We evaluate the impact of the proposed optimization algorithms on TCP’s throughput with respect to two system parameters, namely packet loss rate and propagation delay over the same network. We first consider the packet loss rate in Table 8. With opportunistic retransmission and refined sndbuf growth mechanism, the throughput performances of all TCP+QARR/BARR combinations efficiently utilize the bandwidth at all loss rates, e.g., at least 94.5 percent bandwidth utilization. For example, TCP Vegas+QARR’s throughput degrades insignificantly even at a loss rate of 0.1 percent. As compared to the original TCP Vegas+QARR, the proposed optimization algorithms improve the throughput by 33.6 percent at a loss rate of 0.1 percent, enabling TCP Vegas+QARR to achieve over 95.5 percent bandwidth utilization under all loss rates, and we expect that the improvement will be higher with the increase in loss rate, e.g., over 37 percent improvement at 0.2 percent loss rate, but the loss rate higher than 0.1 percent might not be common over current mobile data networks as measured. As the bandwidth utilization computed includes the TCP slow start period, thus the bandwidth utilization will be higher without slow start period. Note that the throughput performances of all TCP+RH/PRR combinations are not improved by the proposed optimization algorithms, as their throughputs are limited by the congestion control bottleneck, e.g., CWnd/ssthresh, during most of the experiments, instead of the flow control bottleneck and application stall.

Next, we consider the propagation delay in Table 9. As compared to Table 7, with the proposed optimization algorithms the throughput degradation for all the combinations of TCP+QARR/BARR are insignificantly affected, e.g., from 19.1 Mbps to 17.9 Mbps, as QARR/BARR has sufficient data to transmit during the loss recovery period. Hence, that throughput degradation (i.e., 19.1 – 17.9 = 1.2 Mbps) is primarily due to the congestion control bottleneck. The optimization algorithms have little improvements on the combinations of TCP+RH/P RR, as RH/P RR reduces CWnd/ssthresh upon every loss event thus prevents its throughput from increasing.

8.5 Eliminate Bursty Transmission

Besides throughput degradations flow control bottleneck and application stall also result in a side-effect – bursty transmission. Specifically, it prevents TCP sender from transmitting new data during the loss recovery phase, but those data will be transmitted in burst after loss recovery phase, causing RTT spikes, which is not desirable for delay-sensitive application such as video conferencing, running at the same link bottleneck.

To demonstrate this problem, we initiate a TCP flow that downloads 50 MB data and intentionally drop a packet at 5 specific locations respectively using iptables [33] to emulate 0.01 percent loss rate. We record the CWnd and RTT dynamics using the existing loss recovery algorithms. As shown in Fig. 4 (left), we observed that, after the loss recovery phase, RTT increases by over 50 percent, e.g., 150 ms versus 100 ms, with opportunistic retransmission and refined sndbuf growth mechanism.
using PRR. In contrast, those RTT spikes are mitigated completely with the proposed optimization algorithms, e.g., remaining at 100 ms as shown in Fig. 4 (right), as the new data transmissions are spaced out over the loss recovery period by resolving the flow control bottleneck and application stall.

This problem also exists in QARR and BARR. For example, the burst size in QARR is more than the one in PRR, as it is not limited by the congestion control bottleneck, e.g., ssthresh, but the flow control bottleneck. As shown in Fig. 5 (left), the RTT increases by over 80 percent, e.g., 180 ms versus 100 ms. Similarly, the RTT spikes are completely mitigated when applying the proposed optimization algorithms (Fig. 5 (right)).

### 8.6 TCP Throughput Performances Over 4G/LTE Networks

The performance evaluations so far reveal that the optimized loss recovery algorithms efficiently utilize the bandwidth over lossy 3G/HSPA+ networks. It is also meaningful to evaluate it over high-speed mobile data networks, such as 4G/LTE networks, with the world-wide deployment of high-speed mobile data networks that offers over 100 Mbps link speed [3], [12], [31]. In this section, we evaluate the existing loss recovery algorithms w/ and w/o the optimization algorithms over 4G/LTE under the same setups except that the bandwidth of the emulator (i.e., C in Table 3) is set to 100 Mbps. The throughput results are summarized in Tables 10, 11, 12, and 13. We omit the throughput results for TCP+PRR/RH combinations, because their throughputs are throttled primarily by the congestion control bottleneck.

Compared to the 3G/HSPA+, TCP first needs a longer ramp-up period to efficiently utilize the bandwidth over the 4G/LTE network. This period becomes even longer if the loss rate increases, as TCP spends more time in the loss recovery period over which its CWnd does not increase. We tested that a period of 200 s is long enough for TCP to reach a stable throughput. Second, the bandwidth loss due to the flow control bottleneck increases with the higher link bandwidth (c.f., Section 4.6). Thus, as shown in Tables 10 and 11, the baseline bandwidth efficiencies of all the TCP+QARR/BARR combinations over 4G/LTE are further reduced compared to Tables 6 and 7, e.g., 66.3 percent or below, at 0.1 percent loss rate and 100ms delay. We note that QARR performs better than BARR, because BARR significantly underestimates the bandwidth due to few ACK/DUPACKs received during the loss recovery phase, thus reducing CWnd significantly.

As shown in Tables 12 and 13, all the TCP+QARR/BARR combinations can achieve over 89.3 percent bandwidth efficiency with the proposed optimization algorithms under all the loss rates and propagation delays. Compared to 3G/HSPA+, the performance gain of the optimization algorithms in 4G/LTE becomes more significant, e.g., 43.3 percent versus 33.6 percent for TCP Vegas+QARR at 0.1 percent loss rate and 100ms propagation delay, and keeps increasing with higher BDP, which shows that the optimization algorithms can work in a wider network setting, i.e., the larger U and C. Similarly, the proposed optimization algorithms mitigate the RTT spikes in 4G/LTE.
TABLE 13
Performance Impact of the Optimization Algorithms on Existing Loss Recovery Algorithms under Different Propagation Delays Over Emulated 4G/LTE

| Congestion Control | Propagation delay (ms) | 50 | 150 | 200 |
|--------------------|------------------------|----|-----|-----|
|                    |                        |    |     |     |
| CUBIC              | QARR                   | 95.3(95.3%) | 92.7(92.7%) | 91.5(91.5%) |
|                    | BARR                   | 94.8(94.8%) | 92.9(92.9%) | 89.6(89.6%) |
| Veno               | QARR                   | 95.3(95.3%) | 94.1(94.1%) | 93.3(93.3%) |
|                    | BARR                   | 95.3(95.3%) | 93.8(93.8%) | 92.2(92.2%) |
| Westwood           | QARR                   | 95.3(95.3%) | 92.5(92.5%) | 89.3(89.3%) |
|                    | BARR                   | 94.8(94.8%) | 90.8(90.8%) | 90.7(90.7%) |
| Vegas              | QARR                   | 95.3(95.3%) | 93.0(93.0%) | 91.4(91.4%) |
|                    | BARR                   | 95.3(95.3%) | 92.8(92.8%) | 92.1(92.1%) |

8.7 TCP Throughput Performance Over Varying Network Conditions

In this section, we further evaluate the optimized loss recovery algorithms over diverse network conditions. To the end, we modified the emulator to introduce the network variability into the emulated experiments. Specifically, we borrowed the bandwidth traces from [35], [36]. They were collected by developing a custom measurement platform to measure the actual downlink bandwidth by means of sending UDP packets at a high speed (e.g., 1 Gbps) from a server host to a client that is a notebook computer equipped with a 3G/HSPA+ USB modem for connecting to the mobile network. The client notebook recorded the time every 10 ms and the corresponding number of bytes received during that 10 ms. The bandwidth trace lasted for 285.4 hours, exhibiting the downlink bandwidth variations with an overall max/min of 11.2, 7.3, 2.0 Mbps at the downlink. First, we used our custom measurement platform to measure the packet loss rate at different locations and times. Specifically, the server host in the measurement platform sent UDP traffic at different sending rates to the client host. Every UDP datagram has a sequence number so that the client host could calculate the packet loss rate. As shown in Table 16, although it was believed that current local retransmission (e.g., HARQ [59]) could recover most of packet losses, the packet loss rate is nonzero, 0.01 percent with a very low sending rate, e.g., 0.8 Mbps. We conjecture that the local retransmission cannot hide packet losses in all radio conditions, e.g., the loss rate during the office hour is very likely to be higher than the one during the midnight as measured in [3], as the receiver radio link might experience more interferences during the office hour. It is a tradeoff between the achievable radio link utilization and loss rate: one can increase the number of local retransmissions to improve the loss rate, but incurs more radio link overhead, which also could degrade the achievable throughput [60].

TABLE 14
Performance Impact of the Proposed Optimization Algorithms on Existing Loss Recovery Algorithm Over Varying Network Conditions

| Congestion Control | Loss Recovery | 0.05 | 0.1 | 0.05 | 0.1 |
|--------------------|---------------|------|-----|------|-----|
|                    |               | w/   | w/o | w/   | w/o |
|                    |               | Optimization | Optimization | Optimization | Optimization |
| CUBIC              | RH            | 5.6  | 3.9 | 4.7  | 3.2 |
|                    | PRR           | 6.4  | 4.2 | 5.6  | 3.9 |
|                    | QARR          | 9.5  | 9.1 | 7.1  | 6.2 |
|                    | BARR          | 9.0  | 8.7 | 6.6  | 5.2 |
| BBR                | RH            | 6.1  | 5.1 | 5.4  | 4.0 |
|                    | PRR           | 6.9  | 5.8 | 6.1  | 4.5 |
|                    | QARR          | 10.1 | 9.6 | 7.7  | 7.0 |
|                    | BARR          | 9.3  | 9.3 | 7.1  | 6.6 |

TABLE 15
Performance Impact of the Proposed Optimization Algorithms on TCP BBR Over a Production of 3G/HSPA+ Network

| File size | w/ Optimization | w/o Optimization |
|-----------|-----------------|------------------|
| 512 KB    | 2.07            | 2.02             |
| 10 MB     | 6.85            | 6.53             |
| 50 MB     | 7.19            | 6.86             |

TABLE 16
The Actual Packet Loss Rate Measured at Various Sending Rate

| Sending rate (Mbps) | Loss rate (%) |
|--------------------|--------------|
| 0.8                | 0.01         |
| 1.6                | 0.03         |
| 2.4                | 0.33         |
| 3.2                | 0.14         |
| 4.0                | 2.0          |
| 4.8                | 2.73         |
| 5.6                | 2.55         |
| 6.4                | 3.59         |
AWnd size according to the receiving rate and RTT. During the BBR slow start, the CWnd/sending rate of BBR is small thus the AWnd size is limited by the current sending rate. This also results in the flow control bottleneck during BBR slow start phase when the packet loss occurs. In this case, it takes a longer period for BBR+QARR/BARR to reach the network capacity. In Addition, the optimization techniques also improve the estimation accuracy for QARR and BARR so that the CWnd can be adjusted effectively (c.f., Section 8.5).

### 8.9 Application Performance

So far, we evaluated the TCP throughput improvement from the proposed optimization techniques. In this section, we evaluated the application-level QoE improvement of the mobile video streaming from the proposed optimization techniques. We setup a HTTP-based adaptive video streaming based on DASH which consists a simple HTTP server using Apache setup at the server host, and a video client called dashc [64] at the receiver host. The reason why we choose dashc is that it can report the QoE metrics for every downloaded video segment, such as segment video quality, stall time, startup time, etc., which can be used to quantify the overall QoE for streaming video. The dashc streams the Big Buck Bunny movie with two typical video adaptation algorithms: 1) a throughput-based rate adaptation [61] and 2) BBA, a buffer-based rate adaptation [62]. Every video segment is encoded into 10 quality representation levels. To quantify the overall QoE we choose the QoE Function developed by Liu et al. [63], i.e., $U = -0.37 \text{buffratio} + \text{bitrate}/20$, where bitrate is the average video bitrate in kbps and buffratio is the ratio of total video rebuffering duration to total video session duration (i.e., video playback duration plus video rebuffering duration) in percentage. The QoE considers both video quality in terms of video bitrate and streaming performance in terms of buffer ratio.

As shown in Table 17, we evaluate the QoE improvement from the optimization techniques on the loss recovery algorithms using trace-driven emulated experiments. We adopted the same experiment configurations as the ones of the emulated experiments in Section 8.7 (i.e., the downlink bandwidth varies with an overall max/mean/min of 9.98, 7.95 and 2.85 Mbps, and the propagation delay varies with mean 92.5 ms and stddev 25.5 ms). We showed that the optimization techniques also improve the QoE performance for all rate adaptation algorithms due to their throughput performance gain.

### 9 Optimization Limitations

The assumption behind opportunistic retransmission is that the retransmitted packets can be received successfully (c.f., Section 5.1), otherwise, the receiving window cannot be freed to receive the TCP new packets transmitted opportunistically, so rendering the receiver to discard those out-of-AWND packets. However, retransmitted packets are rarely lost again by examining the data trace of the previous experiments even at the loss rate of 0.1 percent. In this section, we investigate the impact on the throughputs of the optimized loss recovery algorithms when the assumption is not true.

Intuitively, high loss rates can make that assumption false. Thus, the loss rate is increased to 0.5 percent, which is uncommon in mobile data networks, and the experiments in Section 8.4 are repeated. The throughput results with different propagation delays are summarized in Table 18. Except for CUBIC+QARR/BARR, we omit the results of the other TCP+QARR/BARR combinations as they have the similar observations. Compared to the throughput results at 0.1 percent loss rate in Table 13, CUBIC+QARR/BARR cannot efficiently utilize the bandwidth, e.g., 46.8 percent versus 92.7 percent for $U = 150$ ms as shown in Table 18 (Optimized). Analysis of the trace data reveals the following three performance bottlenecks:

a) The congestion control bottleneck: TCP spends more time during the loss recovery phase thus it needs a longer ramp-up period at a larger loss rate.

b) The flow control bottleneck: the receiving window cannot be freed up for receiving new packets transmitted opportunistically if the assumption is not true, i.e., the retransmitted packet can be dropped. Those packets sent beyond the AWnd size will be discarded by the receiver, which reduces the case without opportunistic retransmission and thus utilize the bandwidth inefficiently.

c) TCP timeouts: the receiver sends back DUPACKs without new SACKs when receiving those packets sent opportunistically, but these DUPACKs cannot trigger the second fast retransmit of the retransmitted packet at the sender, as TCP flow control ensures that all packets are received within the receiving window thus it does not recognize DUPACKs without new SACKs. As a result, the sender has to wait until that retransmitted packet timeouts, which further deteriorates the throughputs of the optimized QARR/BARR algorithms, possibly making it worse than the throughputs of the original loss recovery algorithms, e.g., 46.8 Mbps versus 47.4 Mbps, as shown in Table 18 (Optimized and Original).

To further validate the bottlenecks (b) and (c), we configure the emulator not to drop the retransmitted packets and repeat the above experiments. As shown in Table 18 (Optimized (Optimal)), the bottlenecks (b) and (c) cannot be observed in
the trace data thus the optimal bandwidth efficiency of the optimized QARR/BARR can reach over 93 percent.

Bottleneck (b) is inevitable unless making the assumption true in most cases, e.g., the network operator assigns high priority to the retransmitted packets in resource allocation and scheduling, but this requires modifications or reconfigurations of any of the mobile network equipment such as RNC or Node-B, which warrants further investigations. However, Bottleneck (c) can be resolved by modifying the sender's kernel to simply make it recognize the DUPACKs without new SACKs, thus we have the following hacks:

Hack 1: in the procedure of processing incoming ACK/DUPACKs, it counts the received DUPACKs without new SACKs in TCP_CA_CWR state (i.e., loss recovery phase), if the count exceeds dupthresh, e.g., 3, it triggers fast retransmit of the latest unacknowledged TCP packet.

As shown in Table 18 (Optimized (w/ Hack 1)), the optimized QARR and BARR with Hack 1 always perform better than the original ones – a direct result of the TCP timeout elimination, while the performance gap from the optimal ones is only due to bottleneck (b) and becomes larger with larger propagation delay, i.e., BDP. This is because, given the same loss rate, it is more likely for retransmitted packets to be lost if the pipe (i.e., packets in flight) becomes larger and thus less throughput gain obtained from opportunistic retransmission.

On the other hand, besides the loss rate, the assumption also can be false, i.e., bottleneck (b) occurs, if the BDP is large enough, even at a lower loss rate. To validate this, we further increase the bandwidth and repeat the experiments at 0.1 percent loss rate. As shown in Table 19, the bandwidth efficiencies of the optimized QARR/BARR decreases with the increase in BDP, while the optimal bandwidth efficiencies of the optimized QARR/BARR without dropping retransmitted packets always remain at over 92 percent. Therefore, we conjecture that, for any loss rate, a minimum BDP can be derived theoretically that a predefined minimum bandwidth efficiency can be guaranteed (e.g., 80 percent) with the optimized TCP + QARR and TCP + BARR combinations, which warrants further investigations.

10 Summary and Future Works

In this work we proposed two optimization algorithms: opportunistic retransmission and refined buffer growth mechanism, to tackle the flow control bottleneck and application stall. The proposed algorithms significantly improve the throughput performances of the existing loss recovery algorithms and mitigate the RTT spikes that would occur after loss recovery phase. We are currently extending this work in two directions. First, we are extending the analytical model in Section 4 to characterize TCP's overall bandwidth utilization under various loss rates and BDPs. Second, we are applying the optimization algorithms for other networks which have larger BDPs and lower loss rates, such as data-center, optical network and inter-ISP WAN.

Acknowledgments

This research was partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61502459, and the National Key Research and Development Program of China (13th Five-Year Plan) under Grant No. 2016YFB1000200.

References

[1] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, “TCP westwood: Bandwidth estimation for enhanced transport over wireless links,” in Proc. ACM MobiCom, Jul. 16, 2001, pp. 287–297.

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

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

[4] F. Ren and C. Liu, “Mitigating and improving TCP performance over cellular link with variable bandwidth,” IEEE Trans. Mobile Comput., vol. 10, no. 8, pp. 1057–1070, Aug. 2011.

[5] R. Chakravorty, S. Katti, I. Pratt, and J. Crowcroft, “Using TCP flow-aggregation to enhance data experience of cellular wireless users,” IEEE J. Sel. Areas Commun., vol. 23, no. 6, pp. 1190–1204, Jun. 2005.

[6] M. Ivanovich, P. W. Bickerdike, and J. C. Li, “On TCP performance enhancing proxies in a wireless environment,” IEEE Commun. Mag., vol. 46, no. 9, pp. 76–83, Sep. 2008.

[7] W. K. Leong, Y. Xu, B. Leong, and Z. Wang, “Mitigating egregious ACK delays in cellular data networks by eliminating TCP ACK clocking,” in Proc. 21st IEEE Int. Conf. Netw. Protocols, Oct. 7-10, 2013, pp. 1–10.

[8] Ke Liu, Jack Y, and B. lee, “Achieving high throughput and low delay by accurately regulating link queue length over mobile data network,” in Proc. IEEE 10th Int. Conf. Wireless Mobile Comput. Netw. Commun., Oct. 8-10 2014, pp. 562–569.

[9] E. H. Wu and M. Chen, “JTCP: Jitter-based TCP for heterogeneous wireless networks,” IEEE J. Sel. Areas Commun., vol. 22, no. 4, pp. 757–766, May 2004.

[10] K. Xu, Y. Tian, and N. Ansari, “TCP-Jersey for Wireless IP Communications,” IEEE J. Sel. Areas Commun., vol. 22, no. 4, pp. 747–756, May 2004.

[11] K. Winston, A. Sivaraman, and H. Balakrishnan, “Stochastic forecasts achieve high throughput and low delay over cellular networks,” in Proc. USENIX NSDI, Apr. 2-5, 2013, pp. 459–471.

[12] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, “An in-depth study of LTE: Effect of network protocol and application behavior on performance,” ACM SIGCOMM, vol. 43, no. 4, pp. 363–374, Jul. 2013.

[13] H. Jiang, Y. Wang, K. Lee, and I. Rhee, “DRWA: A receiver-centric solution to bufferbloat in cellular networks,” IEEE Trans. Mobile Comput., vol. 15, no. 11, pp. 2719–2734, Nov. 2016.

[14] Y. Xu, W. K. Leong, B. Leong, and A. Razeen, “Dynamic regulation of mobile 3G/HSPA uplink buffer with receiver-side flow control,” in Proc. 20th IEEE Int. Conf. Netw. Protocols, Oct. 30 2012, pp. 1–10.

[15] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi, “Proportional rate reduction for TCP,” in Proc. ACM SIGCOMM Conf. Internet Meas. Conf., Nov. 2 2011, pp. 155–170.

[16] F. Lu, H. Du, A. Jain, G. M. Voelker, A. C. Snoeren, and A. Terzis, “CQIC: Revisiting cross-layer congestion control for cellular networks,” in Proc. 16th Int. Workshop Mobile Comput. Syst. Appl., Feb. 12, 2015, pp. 45–56.

[17] S. Frank, L. Posee, M. Lin, G. Sarangadakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber, “Pushing CDN-ISP collaboration to the limit,” ACM SIGCOMM Comput. Commun. Rev., vol. 43, no. 3, pp. 34–44, Jul. 2013.
[67] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, “QUIC: A UDP-based secure and reliable transport for HTTP/2 draft-hamilton-early-deployment-quic-00,” Internet-Draft, Jul. 2016.

[68] QUIC Deployment Experience at Google, [Online]. Available: https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf.

[69] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, “mTCP: A highly scalable user-level TCP stack for multicore systems,” in Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation, 2014, pp. 489–502.

Ke Liu received the BEng and PhD degrees in information engineering from the Chinese University of Hong Kong, Shatin, Hong Kong, in 2008 and 2013, respectively. He was a post-doc scholar in the School of Industrial Engineering, Purdue university, IN, USA, from 2017 to 2018. He is currently an assistant professor in the Advanced System Group under the Key Laboratory of Computer System and Architecture at the Institute of Computing Technology, Chinese Academy of Science, Beijing, China, where he participated in the research of protocol optimization and cloud computing.

Zhongbin Zha received the BEng degree in computer science from Northeastern University, Dalian, China, in 2012, and the MEng degree from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2015. He is currently a senior engineer with Tencent.

Wenkai Wan received the BEng degree in information engineering and computing science from the Beijing Information Science and Technology University, Beijing, China, in 2015, and the MEng degree in software engineering from the Kunming University of Science and Technology, Kunming, China, in 2018. He is currently a software/network engineer at the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS).

Vaneet Aggarwal (S’08-M’11-SM’15) received the BTech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 2005, and the MA and PhD degrees, in electrical engineering from Princeton University, Princeton, NJ, USA, in 2007 and 2010, respectively. He was a senior member of the technical staff research with AT&T Labs Research, Bedminster, NJ, USA, from 2010 to 2014. He was an adjunct assistant professor with Columbia University, NY, from 2013-2014, and an adjunct professor with IIISc Bangalore, India, from 2018-2019. He is currently a faculty at Purdue University, West Lafayette, IN, USA. His current research interests are in communications and networking, video streaming, cloud computing, and machine learning. He received Princeton University’s Porter Ogden Jacobus Honoric Fellowship in 2009, the AT&T Vice President Excellence Award in 2012, the AT&T Senior Vice President Excellence Award in 2012, the 2017 Jack Neubauer Memorial Award recognizing the Best Systems Paper published in the IEEE Transactions on Vehicular Technology, and the 2018 Infocom Workshop HotPOST Best Paper Award. He is on the editorial board of the IEEE Transactions on Communications and the IEEE Transactions on Green Communications and Networking. He is a senior member of the IEEE.

Binzhang Fu received the BEng degree in electronics and information engineering, and computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, and the PhD degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2004 and 2011, respectively. He is currently a research scientist at Huawei.

Mingyu Chen received the BEng degree in electronics engineering, and information science from the University of Science and Technology of China, He Fei, China, and the PhD degree in computer architecture from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1994 and 2000, respectively. He is currently a professor at ICT, CAS. He is currently a professor at ICT, CAS, and the University of Chinese Academy of Sciences (UCAS), Beijing, China. His current research interests include high-performance computer architecture and operating system.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.