On Shaping TCP Traffic at Edge Gateways

Huan-Yun Wei[†], Shih-Chiang Tsao, Ying-Dar Lin
Department of Computer and Information Science
National Chiao Tung University, Hsin-Chu, Taiwan
Tel: +886-939-140280
Email: {hywei,weafon,ydlin}@cis.nctu.edu.tw

Abstract— **Many security and QoS functions have been deployed at edge gateways to provide policy-based network management. For QoS functions, the bandwidth management system can manage the narrow WAN access links. When managing the TCP traffic, pass-through TCP flows can introduce large buffer requirement, latency, buffer overflows, and unfairness among flows competing for the same queue. This study evaluates possible TCP-aware approaches through self-developed implementations in Linux, testbed emulation, and live WAN measurement. The widely deployed TCP Rate control (TCR) approach is found to be more vulnerable to WAN packet losses and less compatible to several TCP sending operating systems. The proposed PostACK approach can preserve TCR's advantages while avoiding TCR's drawbacks. PostACK emulates per-flow queuing but relocates the queuing of data to the queuing of ACKs in the reverse direction, hence minimizes buffer requirement up to 96%. PostACK also has 10% goodput improvement against TCR under lossy WAN. Experimental results can be reproduced through our open sources: (1) tcp-masq: a modified Linux kernel; (2) wan-emu: a testbed for conducting switched LAN-to-WAN or WAN-to-LAN experiments with RTT/loss/jitter emulations.**

I. Introduction

Corporate networks are often connected to the Internet by subscribing Internet access links. The links are narrow so network administrators may install a bandwidth management system at the link to manage the traffic. Thus, the important/interactive/mission-critical traffic such as voice over IP (VoIP), e-business, and ERP (Enterprise Resource Planning) flows are not blocked by the less-important traffic such as FTP. A policy rule usually consists of *condition* and *action* fields that define specific actions for specific conditions. The *condition* field defines the packet-matching criteria, such as a certain subnet or application, to classify packets into their corresponding queues. Then the queued packets are scheduled according to the specified *action*, such as "at least/most 20kbps". After quantitatively evaluating eight major players [1] in the market, we summarize a general bandwidth management model in §I-A. The issues, assumptions, and our problem statement are then detailed in §I-B.

A. General Bandwidth Management Model

Key terms used in this paper are also defined in Fig.1, where two types of policy rules can be exercised:

1. *Class-based bandwidth allocation:* Most bandwidth allocation policies are class-based. As shown in Fig.1, each such policy rule groups a set of flows into a class by the per-class packet classifier. Each class corresponds to a FIFO-based per-class data queue (PCDQ). Data packets queued at the PCDQ are scheduled out to the WAN links by the packet scheduler. A packet scheduler is often a must to control all kinds of traffic including unresponsive flows like UDP and ICMP. Many implementations employed the Class Based Queuing (CBQ) [3], which can efficiently utilize newly available bandwidth among classes. However, multiple TCP flows competing for the same queue can cause high buffer requirement at the edge gateway, hence result in large latency, frequent buffer overflows, and unfairness among the competing TCP flows within the same class. This is owing to the mismatch between the growing TCP window and the fixed bandwidth delay product [4] [6](BDP) of the flow. The microscopic details will be analyzed in §II-B.
2. *Guarantee bandwidth for each flow within a class:* Traditionally, RED [7] can be used to alleviate the unfairness among competing TCP flows within a class. However, RED is less effective to achieve perfect fairness [1] [8] [9]. Nowadays, most vendors have incorporated a per-flow ACK control add-on module (Fig.1) in the reverse direction to actively control the behavior of each TCP sender. All evaluated commercial implementations [1] fairly treat the flows within the class. Namely, if n TCP flows are now mixed in the PCDQ of class c, ideally the bandwidth for each flow BW_i obtains a share of BW_c/n.

B. Issues and Problem Statements

Guided by the above demand, this section defines the issues, problem statement, and a representation of a TCP flow's bandwidth used throughout this paper.

Issues to Study

This study aims at assessing possible per-flow rate control approaches for optimizing the following performance metrics:

1. *Buffer requirement at the edge gateway*, which implies cost and latency (§IV-A.1).
2. *Vulnerability of goodput under lossy environments* (average goodput[1] under packet losses (§IV-A.2)).
3. *Fairness among flows in one class* (flow isolation within one class (§IV-A.3)).

[1] Goodput means effective throughput, which excludes the throughput consumed by retransmissions.

0-7803-8794-5/04/$20.00 © 2004 IEEE

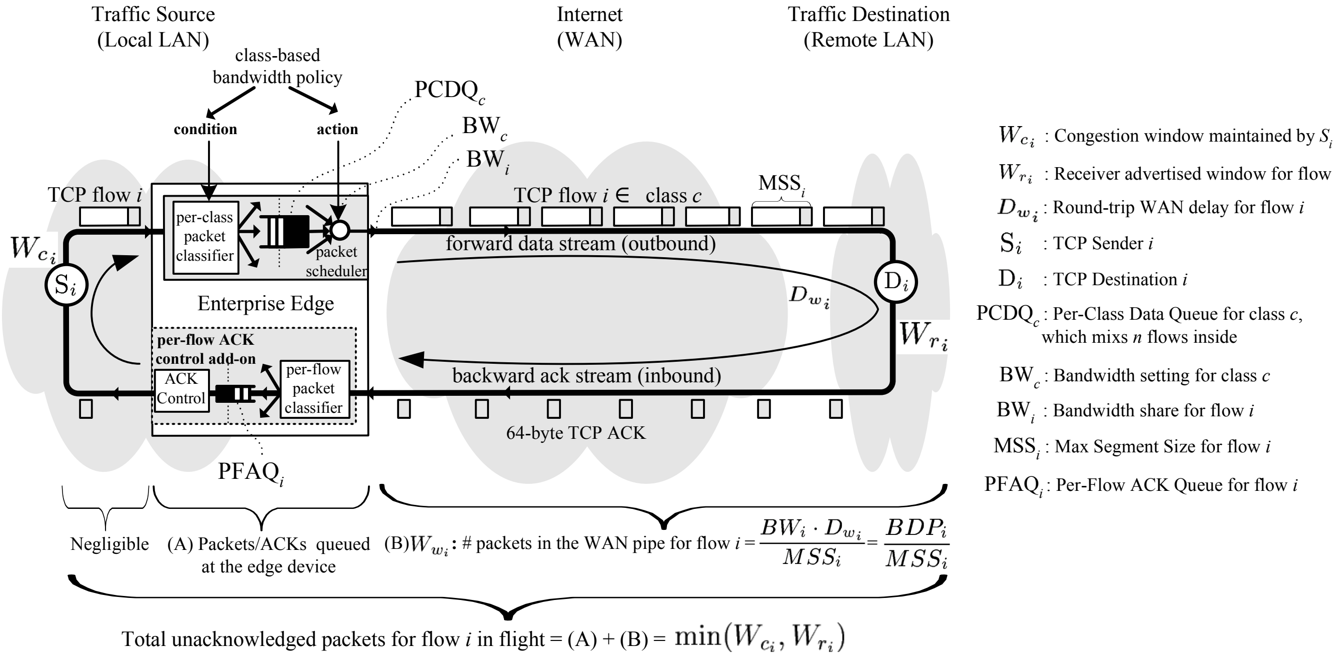
Traffic Source
(Local LAN)
Internet
(WAN)
Traffic Destination
(Remote LAN)
class-based
bandwidth policy
condition
action
PCDQ$_c$
BW$_c$
BW$_i$
TCP flow i
TCP flow i ∈ class c
MSS$_i$
W_{c_i}
S$_i$
per-class packet classifier
packet scheduler
forward data stream (outbound)
Enterprise Edge
D_{w_i}
D$_i$
W_{r_i}
per-flow ACK control add-on
ACK Control
per-flow packet classifier
backward ack stream (inbound)
64-byte TCP ACK
PFAQ$_i$
Negligible
(A) Packets/ACKs queued at the edge device
(B)W_{w_i}: # packets in the WAN pipe for flow i = $\frac{BW_i \cdot D_{w_i}}{MSS_i} = \frac{BDP_i}{MSS_i}$
Total unacknowledged packets for flow i in flight = (A) + (B) = $\min(W_{c_i}, W_{r_i})$
W_{c_i} : Congestion window maintained by S_i
W_{r_i} : Receiver advertised window for flow i
D_{w_i} : Round-trip WAN delay for flow i
S$_i$: TCP Sender i
D$_i$: TCP Destination i
PCDQ$_c$: Per-Class Data Queue for class c, which mixs n flows inside
BW$_c$: Bandwidth setting for class c
BW$_i$: Bandwidth share for flow i
MSS$_i$: Max Segment Size for flow i
PFAQ$_i$: Per-Flow ACK Queue for flow i

Fig. 1. General Bandwidth Management Model For Class-based Outgoing TCP Traffic

4. *Robustness under Various TCP implementations* (§IV-A.4).

Bandwidth of a TCP Flow

TCP throughput modelling has been extensively studied in [12] and [13]. Without considering packet losses for simplicity, the bandwidth (or rate) of a TCP flow can be measured in various time scales as shown in Eq.1. For a TCP flow, choosing its RTT (D_{w_i} plus delays at $PCDQ_c$ and $PFAQ_i$) as the measuring time interval can establish a relation with TCP windows as in Eq.2. Excluding the packets queued at the edge gateway (A in Fig.1), Eq.2 is transformed to Eq.3. Apparently, the bandwidth of a TCP flow can be affected by either shrinking the window size (the TCP rate control approach) or stretching the RTT (the PostACK and per-flow queuing approaches).

$$\begin{aligned} BW_i &= \frac{Bytes\ Sent}{Time\ Interval} && (1)\\ &= \frac{TCP\ window}{RTT} = \frac{\min(W_{c_i}, W_{r_i}) \cdot MSS_i}{D_{w_i} + PCDQ_c^{delay} + PFAQ_i^{delay}} && (2)\\ &= \frac{Bytes\ in\ WAN}{Round\ Trip\ WAN\ Delay} = \frac{W_{w_i} \cdot MSS_i}{D_{w_i}} && (3) \end{aligned}$$

As shown in Fig.1, if the WAN pipe of flow i is full, each bandwidth sample of flow i measured at the end of each D_{w_i} will approximate BW_i; otherwise, the flow is under-utilizing its bandwidth share. Additionally, the more evenly the packets are distributed across the D_{w_i}, the less the fluctuations among the consecutive measured bandwidth samples.

Problem Statement: How to keep flow i to BW_i $(= BW_c/n)$ with optimizations to the performance metrics

As explained in §I-A, any flow $i \in$ class c should obtain $BW_i(= BW_c/n)$. With the optimizations to the above performance metrics, the gateway can have low buffer/latency/cost while keeping high goodput, fairness, and robustness.

C. Organization of This Work

The following sections are organized as follows: The next section reviews TCP sender behaviors and previous works (§II). The PostAck approach is presented in §III. Subsequently, the effectiveness of the schemes are verified through prototype experiments, simulations, and live experiments (§IV). Finally conclusions are given in §V.

II. Background and Related Works

The following sections assume that readers are familiar with TCP congestion control schemes.

A. History of Existing Schemes

Karandikar et al. [8] sponsored by Packeteer [11] propose the TCP Rate control (TCR, a strange acronym named in [8]) approach. While TCR is popular among many commercial implementations [1], it remains only partially studied. TCR is only compared with RED and ECN, which are merely congestion control schemes without keeping per-flow states as TCR does. Additionally, not a single loss in the TCR performance study may hide its deficiencies compared with per-flow queuing[2]. Because better understanding of TCR is helpful in presenting the PostACK algorithm, we review the TCR algorithm in details.

[2] Per-flow queuing (PFQ) assigns each TCP flow to a queue to isolate the bandwidth share. The scheduling algorithm can be any, such as weighted fair queuing [14] (WFQ). In this paper, PFQ results are obtained by simply applying a token bucket shaper to each flow.

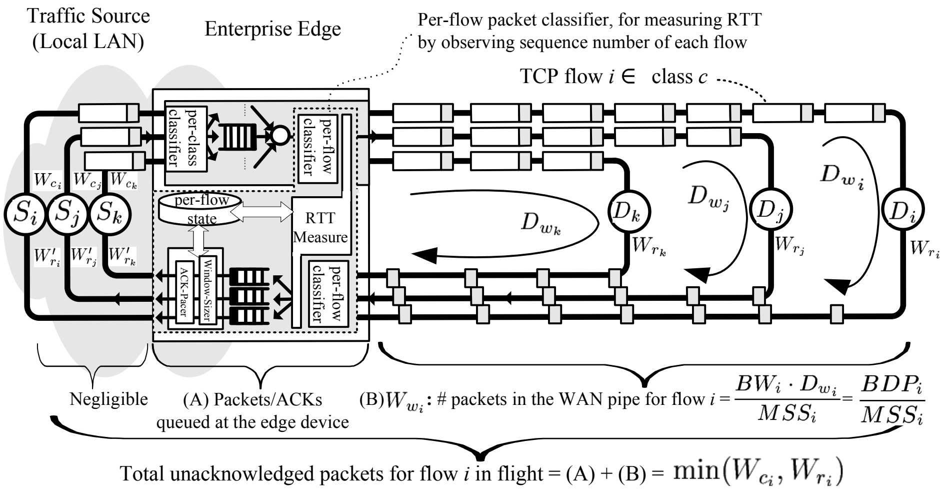
Traffic Source
(Local LAN)
Enterprise Edge
Per-flow packet classifier, for measuring RTT
by observing sequence number of each flow
TCP flow $i \in$ class c
per-class classifier
per-flow classifier
per-flow state
RTT Measure
ACK-Pacer
Window-Sizer
per-flow classifier
W_{c_i} W_{c_j} W_{c_k}
S_i S_j S_k
W'_{r_i} W'_{r_j} W'_{r_k}
D_{w_k} D_{w_j} D_{w_i}
D_k D_j D_i
W_{r_k} W_{r_j} W_{r_i}
Negligible
(A) Packets/ACKs
queued at the edge device
(B)W_{w_i}: # packets in the WAN pipe for flow $i = \frac{BW_i \cdot D_{w_i}}{MSS_i} = \frac{BDP_i}{MSS_i}$
Total unacknowledged packets for flow i in flight = (A) + (B) = $\min(W_{c_i}, W_{r_i})$

Fig. 2. TCP Rate Control ACK Control Model For Managing Outgoing TCP Traffic

B. Prior-Art ACK Control Approach: TCR

1) Algorithm Review: Figure 2 displays the TCR [8] ACK control model that exercises window-sizing and ACK-pacing. If W_{w_i} denotes the number of packets in the WAN pipe for flow i with BW_i $(= BW_c/n)$, then

$$W_{w_i} = \frac{BDP_i}{MSS_i} = \frac{BW_i \cdot D_{w_i}}{MSS_i}, \tag{4}$$

where the equation can be used as follows:

1. *Window-sizing*: Because normally a TCP sender S_i expands its W_{c_i} to speedup its rate, window-sizing tries to slow it down by locking the TCP window $(= min(W_{r_i}, W_{c_i}))$ using the modified W_{r_i}. Window-sizing periodically measures the D_{w_i} by observing the sequence numbers, and then rewrites the W_{r_i} in each ACK with BDP_i bytes (W_{w_i} packets). Thus flow i is expected to just fill up its WAN pipe without overflowing excessive packets to the PCDQ.
2. *ACK-pacing*: To evenly spread W_{w_i} of packets across the WAN pipe, the inter-ACK spacing time, Δ_i, can also be derived from Eq.4 as $\Delta_i = \frac{D_{w_i}}{W_{w_i}} = \frac{MSS_i}{BW_i}$. The ACK-pacing module then *clocks out* ACKs of flow i at intervals of $\frac{MSS_i}{BW_i}$. Thus the W_{w_i} packets from S_i are smoothly paced out and are most likely to be evenly distributed across the measured D_{w_i}.

2) Microscopic Behaviors of TCR-Applied Flows: To develop an efficient ACK-pacing, TCR can be implemented with a single timer for each class instead of for each flow. The timer times out at intervals of $\frac{MSS_i}{BW_i}$ and releases all n ACKs back to the n senders at a time. If window-sizing is *absent*, the reaction of releasing an ACK to a sender depends on the congestion control phase the sender is in:

1. *TCP Senders in Slow-Start Phase:* In TCP slow-start phase, CWND advances by one whenever an ACK acknowledges the receipt of a full-size data segment. So generally *every ACK* released by the edge gateway in this condition will trigger out two new data packets into the corresponding $PCDQ$.
2. *Full-CWND ACKed TCP Senders in Congestion-Avoidance Phase:* In TCP congestion-avoidance phase, CWND advances by one whenever an ACK acknowledges the whole window (CWND) of data packets. So generally each ACK will trigger out one new data packet but *the last ACK of each CWND round*[3] can trigger out two new data packets into the corresponding $PCDQ$.
3. *TCP Senders Exited from Fast Recovery Phase:* Acknowledgements of successfully retransmitted packets may bring the sender out of the fast recovery phase, causing the W_c to reset to $ssthresh$ $(= \frac{1}{2}min(W_c, W_r)$, where W_c herein means the largest W_c before congestion occurs) . The reset action can trigger a burst of packets into the $PCDQ$.

Because the edge gateway cannot accurately identify which sender is in which phase, simultaneous releasing n ACKs to the n flows of class c may result in unfairness. Some flows may respond multiple packets while some flows may only respond one. By window-sizing, TCR can enforce that each ACK will respond exactly one packet no matter in which phase the TCP sender is because the sending window is then bounded by the W_r instead of W_c.

3) Expected Side Effects of TCR Approach: Measuring round-trip WAN delay and modifying the TCP ACK header are expected to have at least three side effects:

1. *Halved-BDP Side Effect: Lower Throughput* Since TCR shrinks the RWND of flow i to its *WAN pipe size* (BDP_i bytes or W_{w_i} packets, which is smaller than W_{r_i}), a single loss can trigger the sender to halve its window down to $\frac{1}{2}W_{w_i}$ (which is smaller than $\frac{1}{2}W_{r_i}$) rather than to $\frac{1}{2}\min(W_{c_i}, W_{r_i})$ packets. Thus the performance degrades even under slight WAN packet losses.
2. *Tiny-Window Side Effect: Less Compatibility and even Lower Throughput* For flows with small BDP (either

[3] CWND round herein means the data packets of the same round that can advance the CWND by one when all of them are ACKed, in congestion avoidance phase.

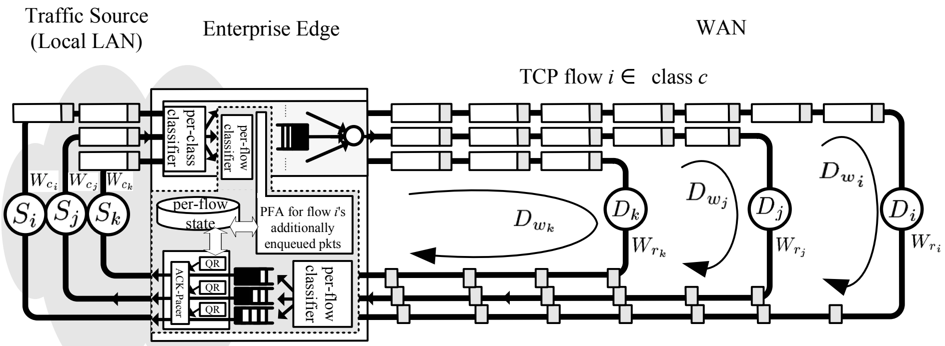

Fig. 3. Efficient PostAck Implementation For Managing Outgoing TCP Traffic

BW_i or D_{w_i} is too small), window-sizing may shrink their RWNDs to the situation that no more than three unacknowledged data packets are in the WAN pipe. As such any single loss resorts to a retransmission timeout (RTO) rather than using fast retransmit (also stated in RFC 3042 [5]). Some classical Berkeley-derived operating systems employ a coarse-grained timer (500ms), which can cause a 1-second idle to retransmit the packet [4]. This significantly degrade the TCR-applied flows. Many enterprises installing heterogeneous OSes may encounter such problems. A recent benchmark[15] among TCR-employed vendors also demonstrate this phenomenon.

III. Alternative ACK Control Approach: PostAck

PostAck is designed to be more intelligent both in retaining previous TCR benefits and eliminating its deficiencies. *Without measuring the WAN delay and shrinking the RWND* in TCP ACKs, PostAck can avoid the side effects of TCR.

A. Motivation: Delaying the ACKs Instead of Data Packets

As indicated in the problem statement, each flow should obtain a bandwidth share of $BW_i = BW_c/n$. Recall that in Fig.1 the RTT consists of D_{w_i}, the queuing delays at $PCDQ_c$ and $PFAQ_i$, and the neligible round-trip LAN delay. Generally the delay at $PFAQ_i$ approaches zero while the forward-data-packet queuing delay for TCP is large. Imagine that a Per-Flow Queuing (PFQ) is placed within the class c to enforce that each $BW_i = BW_c/n$ ($i \in c$). Thus, the number of data packets of flow i queued before the packet scheduler in Fig.1, $PCDQ_i^{qlen}$, is $min(W_{c_i}, W_{r_i}) - (BDP_i/MSS_i)$, namely all unacknowledged packets excluding the packets in the WAN pipe. To achieve BW_i, each queued data packet should wait for a period of $(PCDQ_i^{qlen} * MSS_i)/BW_i$. Imagine that the packet scheduler in the forward direction were absent. By delaying each ACK for the same interval ($(PCDQ_i^{qlen} * MSS_i)/BW_i$), the bandwidth of flow i will also approach its target bandwidth BW_i. The effects of delaying the data packets in the forward direction by the packet scheduler is identical to delaying the ACKs in the reverse direction since a TCP sender only measures RTT, which consists of bidirectional delays. Gradually increasing the delay of ACKs would not cause Retransmission TimeOuts (RTO) because a TCP sender can adapt the RTO to the newly measured RTTs. In summary, the target bandwidth, BW_i, which keeps only BDP_i/MSS_i packets in the WAN pipe, can be achieved through queuing excessive packets. Either queuing the data packets or the ACKs have the same effects on rate shaping.

B. Efficient Implementation for PostACK

Although the concept of PostACK is quite different from that of TCR, PostAck can also be efficiently implemented as an on-off variant of ACK-pacing. Namely it can also employ a per-class timer and has $O(1)$ per-packet processing time complexity, which is as efficient as TCR. Recall that the ACK-pacing interval ($\Delta_i = \frac{MSS_i}{BW_i}$) can be derived without estimating the RTT. So PostACK implemented as an on-off variant of ACK-pacing does not need to measure the WAN delay.

We first recall that TCR achieves the fairness among the n flows within the class c by using a per-class timer to simultaneously release n ACKs to the n TCP senders. Window-sizing forces each ACK to trigger out only one data packet. So n senders are expected to send n data packets into the $PCDQ_c$. Since PostACK do not modify the W_r, when using ACK-pacing, among the n ACKs released to the n TCP senders on a ACK-pacing timeout of class c, slow-start TCP sender $i \in c$ will be triggered out two data packets while congestion-avoidance TCP sender $j \in c$ may be triggered out one or two data packets as discussed in §II-B.2. Thus, flow i and j may not get the same share of bandwidth during this round of ACK-pacing (the interval between two consecutive ACK-pacing timeouts) because during this time interval only n data packets in $PCDQ_c$ can be scheduled out. To retain fairness among flows, whenever seeing k ($k > 1$) data packets of flow i entering the edge gateway after releasing an ACK of flow i, PostACK stops the pacing of flow i's ACK for the next $k - 1$ times. During this silent period, flow i's feedback ACKs still come in from the WAN pipe and get queued, resulting in the delaying of ACKs. Intelligent stopping and resuming ACK-pacing of flow $i \in$ class c guarantee that $BW_i = BW_c/n$.

C. Algorithm

To determine the number of ACK-pacing timeouts to skip for flow i (the $k-1$ in the above example), Per-Flow Accounting (PFA in Fig.3 and $i.out$ in Fig.4) of *additionally enqueued packets* is introduced. Whenever PFA finds additionally enqueued packets of flow i, Queue Relocator (line 03 in Fig.4 and QR in Fig.3) quench the pacing of flow i's ACK to relocate the queuing delay at $PCDQ_c$ to the $PFAQ_i$. As implied in Fig.4, $i.out$ is always non-negative because a sender always emits a packet into the $PCDQ_c$ first (i.e. $i.out = i.out + 1$ in Fig.4) before its corresponding ACK is released (i.e. $i.out = i.out - 1$ in Fig.4). Similar to TCR, generally PostACK expects one data packet (i.e. $i.out = i.out + 1$) after releasing an ACK of flow i (i.e. $i.out = i.out - 1$). However, if two data packets enters $PCDQ_i$ (i.e. $i.out = i.out + 1$ for two times) after releasing an ACK, one additionally enqueued data packet ($i.out > 1$) triggers the QR to stop the next ACK-pacing of flow i.

```
Initialization:
  ACK-pacing timeout interval for class c $\Delta_c = \frac{MSS_i}{BW_i}$
  ACK-pacing timeout function for class c = OnClassTimeout
  $i.out = 0$ /* number of additionally enqueued packets +1 */
Algorithm:
01  OnClassTimeout(class c){ /* per-class timer for ACK-pacing */
02      for each flow $i$ in class $c${
03          if ($i.out > 1$) /* QR: skip this pacing of ACK for flow $i$ */
04              $i.out = i.out - 1$
05          else
06              release an ACK
07              $i.out = i.out - 1$
10  }
11  PCDQ_Enqueue(pkt m, class c){ /* enqueue m to $PCDQ_c$ */
12      $i$=PerFlowClassify(m) /* find m's state information */
13      $i.out = i.out + 1$
14      /* original PCDQ_Enqueue code for $m$ and $c$ goes here */
15  }
```

Fig. 4. Efficient PostAck Implementation: On-Off Variant of ACK-Pacing

IV. Implementations and Experimental Results

We have implemented PostACK and TCR into Linux kernel 2.2.17, together with a practical emulation testbed. The per-flow queuing is achieved by assigning a token bucket policer (available in Linux kernel 2.2.17) to each TCP flows. We hereby describe the implementations and experimental results.

A. Numerical Results

1) Buffer Requirement at the Edge Gateway: This section demonstrates the same effectiveness of PostACK and TCR in saving the buffer space. Figure 5 quantifies the goodput degradation due to buffer overflow at the edge gateway. For a 500KB/s class, pure CBQ requires a huge buffer (243 packets at 32 flows) to achieve the target goodput. For PostAck and TCR, only a reasonable buffer (< 10 packets) is needed. Buffer overflow can cause high retransmission ratio[4] (up to 22% when 32 flows competing for a 1-packet FIFO), which consumes a considerable amount of LAN bandwidth.

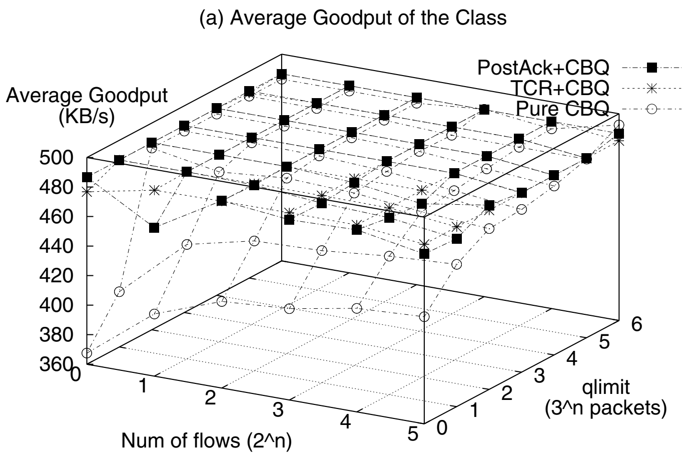

Fig. 5. Buffer requirement of a 500KB/s Class
Testbed Configurations: WAN delay=50ms, class bandwidth=500KB/s, default RWND=32KB.

2) WAN Loss Behaviors (Sensitivity to Internet Loss): Figure 6 compares the goodput degradation due to packet losses in WAN. Under normal WAN loss (below 4% random loss), PostAck obviously outperforms TCR but remains the same degree of TCP-friendliness with PFQ. Microscopic view on the bandwidth fluctuations in 0.5% random WAN loss (Fig.6(a)(b)(c)) proves the benefit of PostACK against TCR. Figure 6(d) shows the average goodput over 10-30 seconds. PostACK can have 10% improvement against TCR under 1% packet loss rate. Under heavily congested conditions (beyond 4% random loss), no significant difference can be found among the three. This is because the throughput is not bottlenecked by the configured rate at the edge gateway anymore. The CWND becomes too small to achieve its target rate.

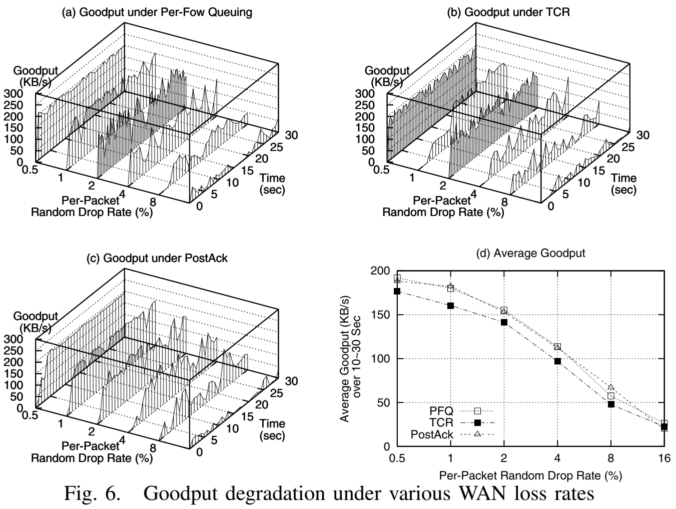

Fig. 6. Goodput degradation under various WAN loss rates
Configuration: A flow bottlenecked by a 200KB/s class runs under various random packet loss rates set by the WAN emulator.

[4] The figure for the corresponding retransmission ratio is omitted due to space limitation.

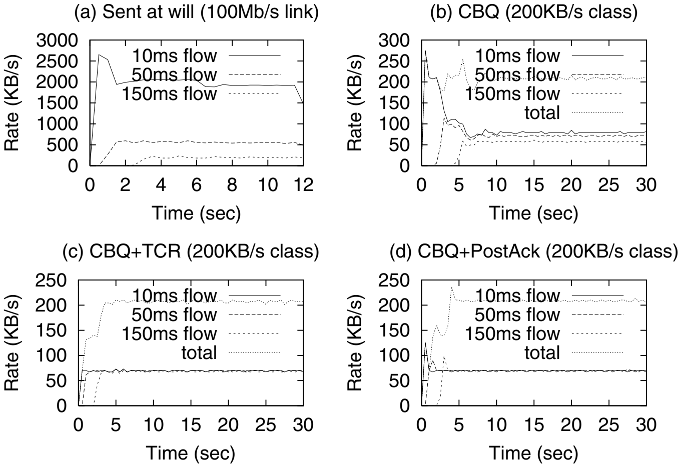

Fig. 7. Fairness among flows in 200KB/s class

Configuration: Three paths are configured as different WAN delays (10ms, 50ms, and 150ms) in wan-emu. In (a) each flow sends packets at its own will; in (b)(c)(d) a 200KB/s class contains the three flows. These figures are measured at the TCP sender.

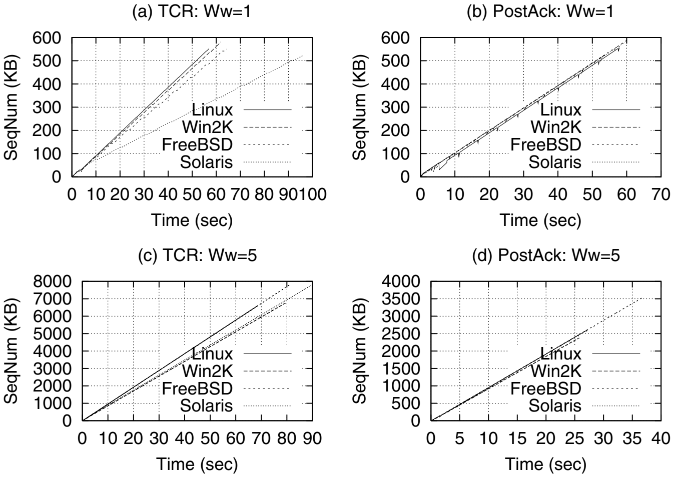

Fig. 8. Robustness under Various TCP sender implementations: TCR vs. PostAck

Testbed Configurations: round-trip WAN delay=50ms, MSS=1500 bytes. In (a)(b), periodic drop rate in WAN=1/40, class bandwidth=10KB/s and thus $W_w < 4$; In (c)(d), periodic drop rate=1/100, class bandwidth=100KB/s and thus $W_w > 4$. Linux 2.2.17, FreeBSD 4.0, Solaris 8, and Windows 2000 are tested.

3) Fairness Among Flows in One Class (Flow Isolation): This section investigates the effectiveness of ACK control modules in resolving the unfairness among TCP flows with heterogeneous WAN delays. Test configurations are described in Fig.7. Figure 7(a) demonstrates the classical problem: throughput of a TCP flow is inversely proportional to its RTT. However, when the three flows share a 200KB/s class in a FIFO PCDQ (Fig.7(b)), the unfairness among the 10ms/50ms/100ms flows is alleviated. This is because the RTT measured by flow i (RTT_i) equals to $D_{w_i} + \sum_i PCDQ_i^{delay}$. The shared PCDQ's queuing delay, $\sum_i PCDQ_i^{delay}$, dominates the RTT_i so that the flows are almost fair. Both TCR (Fig.7(c)) and PostAck (Fig.7(d)) can further eliminate the little unfairness. Note that these figures are measured at TCP sender side so each peak corresponds to the phase of pumping traffic to the edge gateway. The peaks in PostAck are relatively lower than those in CBQ since whenever a PostAck-applied flow gets queued at the PCDQ, the QR in PostACK skip the flow's ACK-pacing. So the peak diminishes immediately.

4) Robustness under Various TCP Implementations: This section tests the robustness of TCR and PostAck under major TCP implementations. The test methodology is self-contained in Fig.8. In Fig.8(a)(b), the bandwidth policy constrains the unacknownledged packets in WAN to one ($W_w = 1$). The *Tiny-Window Side Effect* of TCR occurs in Fig.8(a). Linux takes the finest timer on measuring the RTT and the RTO fires faster than other systems. So Linux sender has the best performance. Solaris keeps a coarse-grained timer and performs badly. Under the condition that five unacknowledged packets ($W_w = 5$) can pipeline in the WAN pipe (Fig.8(c)(d)), goodputs of the TCP flow under Window 2000 or Solaris are still slightly lower than the others. in a recent benchmark, TCR employed by PacketShaper also reveals this phenomenon [1]. In contrast, PostAck (Fig.8(b)(d)) can keep the target rate regardless of TCP implementations.

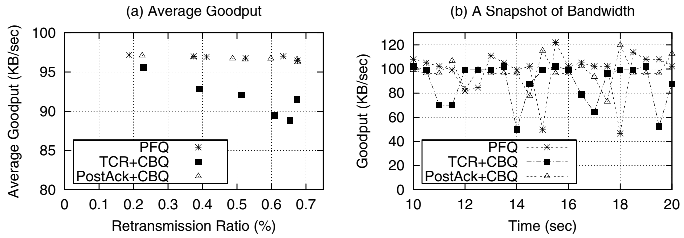

Fig. 9. OurSite-to-UCLA Live Experiments

Configuration: The live experiments were conducted for seven times. Each time a 100KB/s TCP flow is separately run over each scheme for 60 seconds. Data from 10 to 50 seconds is analyzed. The loss rate ranges from 0.002 to 0.007, with corresponding WAN delays ranging from 102 ms to 168 ms.

5) Live WAN Experiments: So far the results are obtained from the tcp-masq over the wan-emu testbed. This section tries to seek empirical validation from live WAN experiments between our site and UCLA. The focus is on how a single TCR/PostACK-applied flow pumping traffic across the Pacific is degraded by the lossy WAN. The test parameters are contained in the Fig.9. The results (Fig.9(a)) confirm that throughput of TCR-applied flows suffer even under slight Internet losses. Figure 9(b) displays the rate fluctuations of one data set. The PostAck scheme has the smallest degree of rate fluctuation.

6) Scalability: The primary overhead in PostAck/TCR is ACK-Pacing, which uses a kernel timer for each class to pace out ACKs. Since TCR in Packeteer's PacketShaper can support 20,000 flows [8] in 1999 (now it is upgraded to at least P-III 600MHz), the kernel timer scales well in modern computers. In fact, the overhead of the per-class timer does not increase as the number of flows, n, sharing the class increases. When n increases, the target bandwidth of flow i, BW_i ($= \frac{BW_i}{n}$), decreases such that the ACK-pacing interval becomes larger, causing the timer to even less busy. Therefore, scalability depends mostly on the bandwidth of the class rather than the number of flows sharing the same class. For PostACK, the stopping/resuming operations in Fig.4 does not introduce any new processing overhead but only skip the flows that send more than expected.

V. Conclusions

This study re-evaluates possible TCP rate shaping approaches, including the TCP Rate control (TCR), the proposed PostAck, and the Per-Flow Queuing (PFQ) approaches, to shape TCP traffic at the organizational edge gateways. Specifically, this study demonstrates the throughput vulnerability (a degradation of 10% shown in §II-B.3) and incompatibility (Solaris' poor RTO) of TCR, which exercises window-sizing and ACK-pacing techniques. Window-sizing is especially widespread among vendors [1] but with only partially studied. An alternative robust and simple approach, PostAck, is hereby proposed (§III) to combine the virtues of TCR (good fairness, low buffer/cost/latency) and PFQ (better performance under loss) without the drawbacks of TCR. All numerical results can be re-produced through our open sources [15]. Notice that PostACK/TCR is not limited to only apply on CBQ, but can also apply on any queuing-based link-sharing mechanisms. However, this study customizes PostACK/TCR to work for CBQ because CBQ is the most popular link-sharing mechanism.

Table I summarizes the pros and cons among them. Notice that under WAN without any loss, PostACK can also achieve perfect fairness, as PFQ and TCR can, if the measuring time scale lasts for several RTTs. But if we measure the bandwidth with a very fine-grained time scale, PostACK's fairness is slightly degraded. However, in lossy WAN environments, several found side-effects of TCR question its perfect fairness. Honestly speaking, ACK control has always been a cool hack, but not a deep solution. This study is perhaps most interesting as a big picture of how much you can shape TCP traffic transparently, especially in lossy WAN environments. Hence, the comparison sometimes shows tradeoffs among the schemes. In lossy environments, TCR does not always work perfectly both in their commercial implementation [1] and our tcp-masq implementation. PostACK can be an alternative.

TABLE I
Comparison of PFQ, TCR, and PostACK: Performance Metrics

Metrics	PFQ	TCR	PostACK
Goodput (under lossy WAN)	High	Slightly Lower	High
Fairness (fine-grained)	Good	Good	Slightly Lower
Fairness (under lossy WAN)	Good	Degraded	Similar to PFQ
Buffer Requirement	High	Low	Low
Data Packet Latency	Large	Little	Little
ACK Packet Latency	Little	Little	Large
Robustness (under lossy WAN)	High	Poor	High
Stability (under lossy WAN)	High	Slightly Lower	High

Table II compares the implementation complexities among the three. Note that PFQ cannot be $O(1)$ when using a fine-grained packet scheduler such as WFQ. They all require $O(N)$ space where N denotes the number of TCP flows passing through.

TABLE II
Comparison of PFQ, TCR, and PostACK: Complexity

Complexity	PFQ	TCR	PostACK
Classification	per-flow	per-flow	per-flow
Time	$O(1)$	$O(1)$	$O(1)$
Space	$O(N)$	$O(N)$	$O(N)$
RTT Measurement	No	Yes	No
Header Modification	No	Yes	No
Checksum Recalculation	No	Yes	No

Some bandwidth management vendors use GPL-licensed open sources, such as Linux kernel, but never do they open their modifications. Currently tcpmasq is under the process of applying GPL license. After all, open source should be open.

Acknowledgments

We thank Alan Cox at RedHat for his kind suggestions on attaching multiple Linux kernel virtual device drivers onto a network interface when designing the WAN emulator, and Prof. Mario Gerla for providing the TCP sinks in UCLA for live WAN experiments.

References

[1] H. Y. Wei and Y. D. Lin, *A Survey and Evaluation of Bandwidth Enforcement Techniques over Edge Gateways*, IEEE Communications Surveys and Tutorials, Vol.5, No. 2, 2003.

[2] Y. D. Lin, H. Y. Wei, and S. T. Yu, *Integration and Performance Evaluation of Security Gateways: Mechanisms, Implementations, and Research Issues*, IEEE Communications Surveys and Tutorials, Vol. 4, No. 1, 2002.

[3] S. Floyd, and V. Jacobson, *Link-sharing and resource management models for packet networks*, IEEE/ACM Transactions on Networking, Vol. 3, No. 4, pp.365-386, 1995.

[4] W. R. Stevens, *TCP/IP Illustrated Volume 1 - The Protocols*, Addison-Wesley, pp.289, 1994.

[5] M. Allman, H. Balakrishnan, and S. Floyd, *Enhancing TCP's Loss Recovery Using Limited Transmit*, RFC 3042, Jan. 2001.

[6] J. Mahdavi, *Enabling High Performance Data Transfers on Hosts*, "http://www.psc.edu/networking/perf_tune.html".

[7] S. Floyd, and V. Jacobson, *Random early detection gateways for congestion avoidance*, IEEE/ACM Transaction on Networking Vol. 1, No. 4, pp.397-413, Aug. 1993.

[8] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer, *TCP Rate Control*, ACM Computer Communication Review, Vol. 30, No. 1, Jan. 2000.

[9] D. Lin and R. Morris, *Dynamics of Random Early Detection* ACM SIGCOMM'97, Cannes, France, Sep. 1997.

[10] S. Floyd, and K. Fall, *Promoting the use of end-to-end congestion control in the Internet* IEEE/ACM Transaction on Networking, Vol. 7, No. 4, pp.458-472, Aug. 1999,

[11] Packeteer, Inc., "http://www.packeteer.com".

[12] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, *Modelling TCP Throughput: A Simple Model and its Empirical Validation*, ACM SIGCOMM'98, Vancouver, British Columbia, Sep. 1998.

[13] E. Altman, K. Avrachenkov, and C. Barakat, *A Stochastic Model of TCP/IP with Stationary Random Losses*, ACM SIGCOMM'00, Stockholm, Sweden, Aug. 2000.

[14] A. Demers, S. Keshav and S. Shenker, *Analysis and simulation of a fair queuing algorithm*, ACM SIGCOMM'89, Austin, TX USA, Sep. 1989.

[15] *Open Source: tcp-masq and wan-emu Linux kernel patches*, "http://speed.cis.nctu.edu.tw/bandwidth/opensource/" "http://speed.cis.nctu.edu.tw/wanemu/"