Assessing and Improving TCP Rate Shaping over Edge Gateways

Huan-Yun Wei, Shih-Chiang Tsao, and Ying-Dar Lin

Abstract—Computers installed with commercial/open-source software have been widely employed as organizational edge gateways to provide policy-based network management. Such gateways include firewalls for access control, and bandwidth managers for managing the narrow Internet access links. When managing the TCP traffic, pass-through TCP flows can introduce large buffer requirements, large latency, frequent buffer overflows, and unfairness among flows competing for the same queue. So, how to allocate the bandwidth for a TCP flow without the above drawbacks becomes an important issue. This study assesses and improves TCP rate shaping algorithms to solve the above problems through self-developed implementations in Linux, testbed emulations, live Internet measurements, computer simulations, modeling, and analysis. The widely deployed TCP Rate control (TCR) approach is found to be more vulnerable to Internet packet losses and less compatible to some 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 minimizing the buffer requirement up to 96 percent. PostACK also has 10 percent goodput improvement against TCR under lossy WAN environments. A further scalable design of PostACK can scale up to 750Mbps while seamlessly cooperating with the link-sharing architecture. 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.

Index Terms—Bandwidth management, TCP, rate enforcement, window-sizing, ACK-pacing, scheduling, queuing, packet scheduler, testbed.

1 Introduction

Policy-based networking is a plan of an organization to achieve its resource-sharing objectives. Many policy-based gateways have been installed at the LAN-WAN interconnected edges to enforce their organizational policies. Such gateways include Firewall, Virtual Private Network (VPN), Network Address Translation (NAT), Content Filtering (CF), Intrusion Detection System (IDS), and Bandwidth Management (BM). Nowadays, computers (especially x86-compatible) installed with commercial or open-source software such as Linux have been the most widely used platform to provide high performance services [1], [2]. From the bandwidth management aspect, since end-to-end Internet QoS such as DiffServ [3] is still under experiment, enterprises seek to at least manage their inbound and outbound traffic on the expansive but narrow Internet access links. Thus, their important, interactive or mission-critical traffic such as voice over IP (VoIP), e-business, and ERP (Enterprise Resource Planning) flows are not blocked by less-important traffic such as FTP. A policy rule usually consists of *condition* and *action* fields that define specific actions for specific conditions. For bandwidth policy rules, 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." The urgent demand for such gateways encourages many commercial or open-source implementations.

An intuitive example: A 125kbps access link is partitioned into a 90kbps VoIP class and a 35kbps FTP class. If there is no voice call, FTP sessions can occupy the entire 125kbps link. Whenever a 30kbps VoIP session starts, the bandwidth manager allocates 30kbps for the VoIP class until the 90kbps is used up by the three 30kbps voice calls. Administrators can set the minimum bandwidth for each FTP flow to be 10kbps. When the FTP class contains only 35kbps, the bandwidth manager allocates about 11.6kbps for the first three FTP sessions. Any newly initiated FTP sessions will be blocked by the bandwidth manager since the minimum bandwidth for each FTP session is 10kbps now. If a voice call leaves, the FTP class can obtain another 30kbps and become a 65kbps class. So, a newly initiated FTP session is allowed to join the FTP class. The four FTP sessions fairly share the 65kbps class and satisfy the administrative 10kbps minimum session bandwidth guarantee.

After quantitatively evaluating three kinds of policies among eight major players [1] in the market, we summarize a general bandwidth management model in Section 1.1. The objectives and contributions are then described in Section 1.2.

1.1 General Bandwidth Management Model

Most surveyed bandwidth management gateways [1] can control both inbound and outbound traffic. For simplicity, the following general model (Fig. 1) focuses on the control of outbound TCP traffic. Inbound control of TCP traffic is

• *The authors are with the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan 300. E-mail: {hywei, weafon, ydlin}@cis.nctu.edu.tw.*

Manuscript received 26 July 2002; revised 23 June 2003; accepted 21 July 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117027.

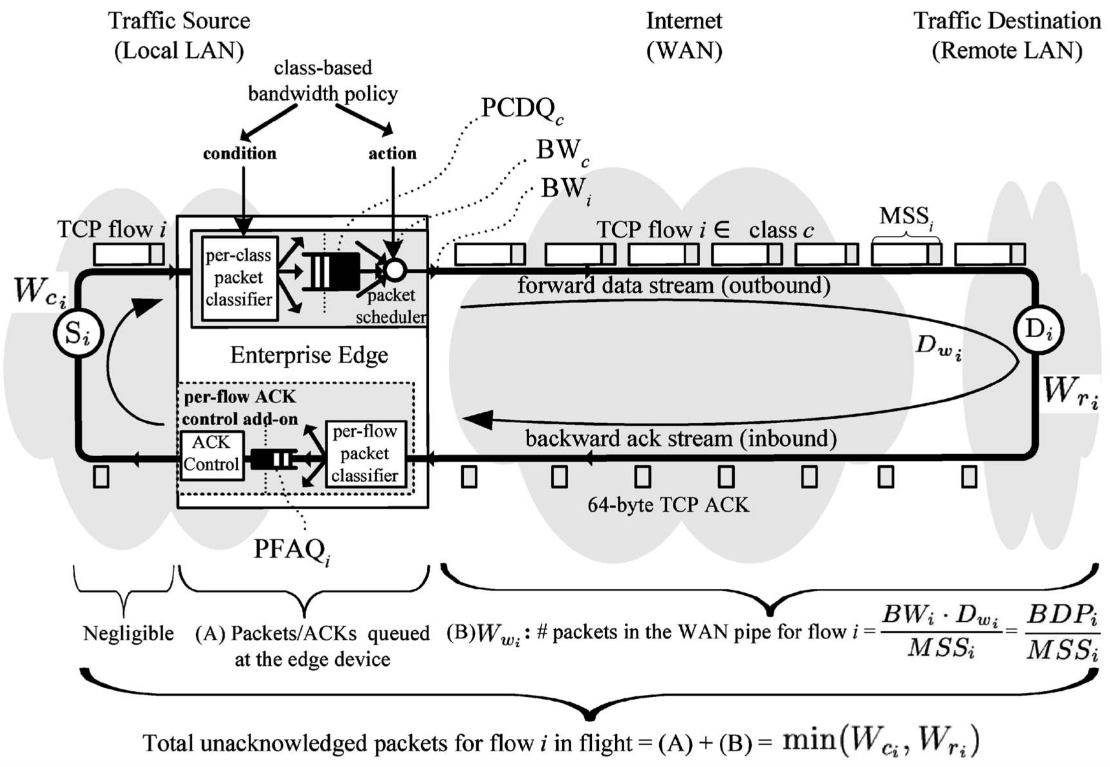

Traffic Source (Local LAN)
class-based bandwidth policy
condition
action
Internet (WAN)
Traffic Destination (Remote LAN)
PCDQ$_c$
BW$_c$
BW$_i$
TCP flow i
per-class packet classifier
packet scheduler
TCP flow $i \in$ class c
MSS$_i$
forward data stream (outbound)
W_{c_i}
S$_i$
D$_i$
Enterprise Edge
D_{w_i}
per-flow ACK control add-on
ACK Control
per-flow packet classifier
W_{r_i}
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) = $\min(W_{c_i}, W_{r_i})$

Fig. 1. General bandwidth management model for class-based outgoing TCP traffic. Terms: S_i: TCP sender i; D_i: TCP destination i; W_{c_i}: congestion window maintained by S_i; W_{r_i}: receiver advertised window announced by D_i; D_{w_i}: round-trip WAN delay for flow i; $PCDQ_c$: per-class data queue for class c, inside which mixes N flows; $PFAQ_i$: per-flow ACK queue for flow i, inside which queues the ACKs for flow i; BW_c: bandwidth settings for class c; BW_i: bandwidth share for flow i $(= BW_c/N)$; MSS_i: max segment size for flow i.

discussed in Appendix A. 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 PCDQ are scheduled out to the WAN pipe 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) [4], 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 resulting in large latency, frequent buffer overflows, and unfairness among the competing TCP flows within the same class. This is due to the mismatch between the growing TCP window and the fixed bandwidth delay product [5], [7] (BDP) of the flow. The microscopic details will be analyzed in Section 2.4.
2. *Guarantee bandwidth for each flow within a class:* Traditionally, RED [8] can be used to alleviate the unfairness among competing TCP flows within a class. However, RED is less effective to achieve perfect fairness [1], [9], [10]. 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, no weighted fairness can be set among the flows in the class. Since flows are dynamically created, it is not practical to assign some specific rate on the fly to some dynamically created flow. In Fig. 1, 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.

1.2 Objectives and Contributions

Guided by the above demand, our objectives and contributions lie in assessing and improving possible approaches, namely, the TCP Rate Control, Per-Flow Queuing, and the proposed PostACK, to solve the problem defined in this section.

1.2.1 Problem Statement

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

1. *Buffer requirement at the edge gateway*, which implies cost and latency (Section 5.2.1).
2. *Vulnerability of goodput under lossy environments* (average goodput[1] under packet losses (Section 5.2.2)).

1. The reason is minor when applying RED at edge gateways because the retransmissions only consume LAN bandwidth.

3. *Fairness among flows in one class* (flow isolation within one class (Section 5.2.3)).
4. *Robustness under Various TCP implementations* (Section 5.2.4).

1.2.2 Underlying Assumptions

As described in Section 1.1, a TCP-unaware packet scheduler is always needed to deal with unresponsive flows. Moreover, a qualified network administrator should not mix TCP flows with TCP-unfriendly [11] (i.e., unresponsive) flows in the same class (i.e., queue). Additionally, the LAN bandwidth is big so that the delay and frame loss rate in LAN are negligible compared to those in WAN. Bulk data transfer is assumed during the analysis. Encryption/decryption beyond the IP layer are performed after/before bandwidth management, respectively, so that the gateway can differentiate the flows. Importantly, TCP flows should be able to reach their target rates, namely, they are bottlenecked by their configured rates at the edge gateway rather than the receiver advertised window sizes ($W_{r_i} > W_{w_i}$). For simplicity, all figures and discussions assume that TCP receivers will instantly reply with an ACK for each successful received data packet.

1.3 Organization of This Work

The following sections are organized as follows: The next section reviews TCP sender behaviors and previous works (Section 2). The PostACK approach is presented in Section 3. Next, the microscopic behaviors and goodput analysis of TCP over these schemes are modeled (Section 4.1). Subsequently, the effectiveness of the schemes is verified through prototype experiments, simulations, and live experiments (Section 5). Section 6 designs a scalable PostACK for gigabit networks. Finally, conclusions are given in Section 7. Some analytical works, proofs, and discussions are included in the appendices.

2 Background and Related Works

This section briefly reviews the behaviors and throughput of a TCP sender. Subsequently, we survey previous approaches in solving the problem.

2.1 Brief Review of TCP Sender Behaviors

The design philosophy of TCP aims at *reliably* and *cooperatively* [14] utilizing network resources. As for reliability, TCP senders carefully avoid overflowing their receivers' buffer and retransmit lost packets which are not acknowledged within a timeout. As for cooperation, TCP senders infer network congestion by detecting packet loss events and trade off their goodputs for network stability. To satisfy both of them, each TCP sender keeps two window values, receiver advertised window (RWND or W_r) and congestion window (CWND or W_c), indicating its receiver's buffer capacity (flow control) and the current network capacity (congestion control), respectively. So, each TCP sender does not have unacknowledged data more than the min(RWND, CWND). RWND is advertised by the receiver in TCP ACK packets and ranges widely among operating systems [7]. CWND, which is kept by the sender, increases exponentially during the slow-start phase and linearly during the congestion-avoidance phase to probe available bandwidth until packet losses occur. Loss behavior differs among TCP versions, mainly on how the CWND is shrunken and raised or on how the lost packets are accurately retransmitted . Fall and Floyd [15] give a good overview and problems on Tahoe, Reno, NewReno, and SACK versions. Vegas [16] and FACK [17] are also famous for their elaborate designs. The four TCP congestion control algorithms, slow start, congestion avoidance, fast retransmit, and fast recovery are formally defined in [18] as basic requirements of an Internet host. The TCP sender implementation in Linux kernel 2.2.17, which constitutes most of our traffic source when evaluating related schemes, is a joint implementation of NewReno, SACK, and FACK. The following sections assume that readers are familiar with TCP congestion control schemes.

2.2 Bandwidth of a TCP Flow

TCP throughput modeling has been extensively studied in [12] and [13]. But, they are too complex to be used for discussion in this paper. Without considering packet losses for simplicity, the bandwidth (or rate) of a TCP flow can be measured in various time scales as shown in (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 (2). Excluding the packets queued at the edge gateway (A in Fig. 1), (2) is transformed into (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).

$$BW_i = \frac{Bytes\ Sent}{Time\ Interval} \tag{1}$$

$$= \frac{TCP\ window}{RTT} = \frac{\min(W_{c_i}, W_{r_i}) \cdot MSS_i}{D_{w_i} + PCDQ_c^{delay} + PFAQ_i^{delay}} \tag{2}$$

$$= \frac{Bytes\ in\ WAN}{Round\ Trip\ WAN\ Delay} = \frac{W_{w_i} \cdot MSS_i}{D_{w_i}}. \tag{3}$$

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 underutilizing its bandwidth share. Additionally, the more evenly the packets are distributed across the D_{w_i}, the fewer the fluctuations among the consecutive measured bandwidth samples.

2.3 History of Existing Schemes

Several packet scheduling algorithms have been proposed and formally analyzed by Stiliadis and Varma [19]. Floyd and Jacobson [4] investigate the hierarchical link sharing among bandwidth classes. Bennett and Zhang [20] further propose a theoretical-proven link-sharing architecture that can simultaneously support real-time traffic. These schemes are TCP-unaware approaches.

Since a TCP sender is clocked by its feedback ACKs, most TCP-aware works tend to *regulate* the ACKs to actively control the TCP senders. The approaches are motivated by two concepts: window-sizing and ACK-pacing. Window-sizing

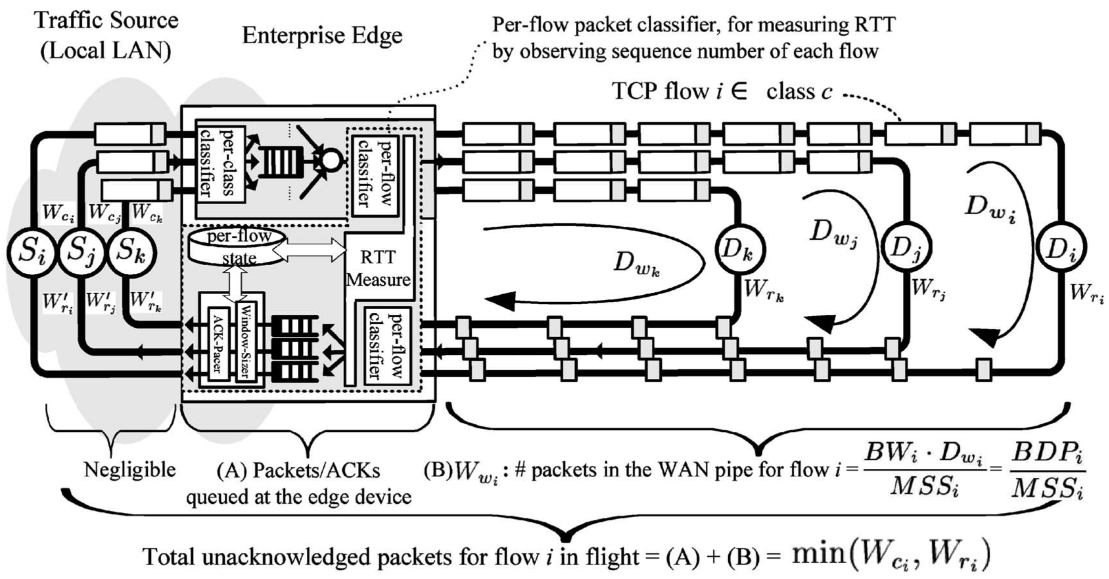

Traffic Source (Local LAN)
Enterprise Edge
Per-flow packet classifier, for measuring RTT by observing sequence number of each flow
TCP flow $i \in$ class c
per-class classifier
per-flow classifier
per-flow state
RTT Measure
ACK-Pacer
Window-Sizer
W_{c_i} W_{c_j} W_{c_k}
W'_{r_i} W'_{r_j} W'_{r_k}
S_i S_j S_k
D_k D_j D_i
D_{w_k} D_{w_j} D_{w_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.

determines how much to send while ACK-pacing decides when to send. Many works employ these concepts, but only typical examples are outlined here.

2.3.1 *Window-Sizing*

TCP Vegas [16] employs a sender-based window-sizing scheme that adapts its CWND to the BDP by a fine-grained RTT measurement. Kalampoukas et al. [21] propose a gateway-based window-sizing scheme that reduces the buffer overflow at interconnected gateways by modifying the RWND in TCP ACKs. Spring et al. [22] present a receiver-based window-sizing approach that makes each bulk-data transfer advertise a small RWND.

2.3.2 *ACK-Pacing*

Zhang et al. [23] suggest sender-based pacing to alleviate the ACK-compression phenomenon due to cross traffic. Narváez and Siu [24] propose a gateway-based scheme that regulates ACKs to make the sender adapt to the ABR explicit rate. Aggarwal et al. [25] then summarize sender-based and receiver-based approaches.

2.3.3 *Hybrid (Window-Sizing Plus ACK-Pacing)*

Karandikar et al. [9] sponsored by Packeteer [26] propose a edge-gateway approach, the TCP rate control (TCR, a strange acronym named in [9]), that combines window-sizing and ACK-pacing. 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, next we assess the TCR algorithm in detail.

2. Goodput means effective throughput, which excludes the throughput consumed by retransmissions.

2.4 Prior-Art ACK Control Approach: TCR

2.4.1 *Algorithm Review*

Fig. 2 displays the TCR [9] 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 speed up 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 (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}.

2.4.2 *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 the 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 the 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:* Acknowledgments 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 of n ACKs to the n flows of class c may result in unfairness. Some flows may respond to multiple packets while some flows may only respond to one. By window-sizing, TCR can enforce that each ACK will respond to 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.

2.4.3 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 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 [6]). Some classical Berkeley-derived operating systems employ a coarse-grained timer (500ms), which can cause a 1-second idle to retransmit the packet [5]. This significantly degrades the TCR-applied flows. Many enterprises installing heterogeneous OSs may encounter such problems. A recent benchmark [28] among TCR-employed vendors also demonstrates this phenomenon.
3. *Inaccurate D_{w_i} Estimation Side Effect: Less Fairness*: In [9], the WAN delay is assumed to be a constant. So, the TCR approach can effectively adjust the window size as described. However, the D_{w_i} of a flow i can vary dramatically. An increase of measured D_{w_i} indicates an increase in queuing rather than an increase in end-to-end distance. Then, the misleading D_{w_i} causes the TCR to raise the modified W_{r_i}, thereby causes a burst of traffic into its corresponding $PCDQ$ that results in unfairness among flows within the class. Our TCR implementation uses the exponential weighted moving average (EWMA) as in thr TCP RTT measurement to smooth the burst.

3. 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 [27] (WFQ). In this paper, PFQ results are obtained by simply applying a token bucket shaper to each flow.

3 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.

3.1 Motivation: Delaying the ACKs instead of Data Packets

As assumed in Section 1.2, ideally, 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 was 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 Time-Outs (RTO) because a TCP sender can adapt the RTO to the newly measured RTTs. Without considering any implementation details, the PostACK algorithm is shown in Fig. 3. The TCP window ($min(W_{c_i}, W_{r_i})$) can be estimated by watching the data stream and the ACK stream.

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. While queuing data packets has many drawbacks (Section 1), queuing the ACKs has many advantages:

1. *Low buffer requirement:* Buffer requirement for TCP at edge gateways can be minimized up to 96 percent

```
Initialization:
  $n$ = # TCP flows in class $c$ with bandwidth $BW_c$
  $BW_i = BW_c/n$
Algorithm:
  if $(min(W_{c_i}, W_{r_i}) < (BDP_i/MSS_i))${
    /* flow i is under-utilizing its share */
    do nothing
  }else{
    /* flow i is over its share */
    $PCDQ_i^{qlen} = min(W_{c_i}, W_{r_i}) - (BDP_i/MSS_i)$
    Delay each feedback ACK for $(PCDQ_i^{qlen} * MSS_i)/BW_i$
  }
```

Fig. 3. Basic Post/ACK Algorithm.

($= \frac{1{,}500-64}{1{,}500}$) since an ACK takes only 64 bytes, while a data packet takes roughly 1,500 bytes if the path MTU so permits. Zero buffer is feasible because ACKs can be artificially generated by the edge gateway. However, it improves little and introduces additional overheads to record TCP timestamp/SACK options. So, our implementation chooses to merely queue the ACKs.

2. *Low data packet latency:* Low buffer requirement implies small data packet latency at the gateway. So, a newly created flow can establish the connection faster because the shared $PCDQ_c$ and the flow's initial $PFAQ_i$ are empty. Thus, users may obtain a faster response when establishing a connection.
3. *Fairness among Flows within a Class:* Each flow in class c can be enforced to its target bandwidth BW_i ($= BW_c/n$).
4. *Higher Goodputs than TCR-applied Flows:* Without measuring the D_{w_i} and shrinking the W_r, PostACK does not have the Halved-BDP side effect (Section 2.4.3).
5. *More Robust under Various TCP Implementations:*Same as above.

While queuing the ACKs may sound good, it is not practical due to the following two challenges:

1. *Computational Inefficiency:* The algorithm in Fig. 3 may not be computationally efficient because it requires accurately delaying each ACK for some time. The most straightforward method is to employ a timer for each TCP flow to shape its ACKs. If there are N flows passing through the edge gateway, the kernel timers require per-packet $O(logN)$ complexity.
2. *Bottlenecked by $PCDQ_c$:* Since a packet scheduler is assumed to be always present at the forward direction to manage non-TCP traffic, $PCDQ_c$ and $PFAQ_i$ are two shaping points of flow i. Because the n flows sharing the $PCDQ_c$ should obtain BW_c/n, packets of flow i will be bottlenecked by its $PCDQ_c$ first rather than by its $PFAQ_i$. Obviously, the $PCDQ_c$ will grow up before $PFAQ_i$ can queue the ACKs since $PCDQ_c$ is the first shaping point of flow i.

The above challenges question the deployment of the basic PostACK algorithm in Fig. 3. However, an efficient and elegant PostACK does exist. To cope with the above challenges, the implementation should avoid having per-packet $O(logN)$ complexity or being bottlenecked by the $PCDQ$. Moreover, it should neither require the information of CWND nor measure the RTT of a flow to estimate its BDP.

3.2 Efficient PostACK Implementation

3.2.1 Motivations to Overcome the Challenges

Although the concept of PostACK is completely different from that of TCR, PostACK can also be efficiently implemented as an on-off variant of ACK-pacing. Namel,y 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.

To overcome the second challenge, 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 such that 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 an 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 Section 2.4.2. 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 $BW_i = BW_c/n$.

3.2.2 Efficient Implementation: Relocating the Queuing Delay

To determine the number of ACK-pacing timeouts to skip for flow i (the $k - 1$ in the above example), Per-Flow Accounting ($i.out$ in Fig. 4 and PFA in Fig. 5) of *additionally enqueued packets* is introduced. Whenever PFA finds additionally enqueued packets of flow i, $QueueRelocator$ (line 03 in Fig. 4 and QR in Fig. 5) quench the pacing of flow i's ACK ($PFAQ_i$ in Fig. 5) to relocate the queuing at $PCDQ_c$ to the $PFAQ_i$. Namely, when the first shaping point ($PCDQ_c$) discovers that flow i is over its share, instead of queuing data packets at the first shaping point, the PostACK queues additional packets in $PFAQ_i$ as ACKs by temporarily quenching the pacing of flow i's ACKs. As implied in Fig. 4, $i.out$ is always nonnegative 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.,

```
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:
OnClassTimeout(class c){ /* per-class timer for ACK-pacing */
    for each flow $i$ in class $c${
        if ($i.out > 1$) /* QR: skip this pacing of ACK for flow $i$ */
            $i.out = i.out - 1$
        else
            release an ACK
            $i.out = i.out - 1$
}
PCDQ_Enqueue(pkt m, class c){ /* enqueue m to $PCDQ_c$ */
    $i$=PerFlowClassify(m) /* find m's state information */
    $i.out = i.out + 1$
    /* original PCDQ_Enqueue code for $m$ and $c$ goes here */
}
```

Fig. 4. Efficient PostAck implementation: on-off variant of ACK-pacing.

$i.out = i.out + 1$) after releasing an ACK of flow i (i.e., $i.out = i.out - 1$). However, if two data packets enter $PCDQ_i$ (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.

4 Modeling and Analysis of ACK Control Schemes

This section analyzes the microscopic behaviors of TCR/PostACK-applied TCP flows. A packet-level model, time series snapshot (TSS) model is presented to model the CWND evolution. This facilitates the modeling of TCP goodput (i.e., effective throughput) and buffer requirement at the edge gateway.

4.1 Time-Series Snapshots Model

4.1.1 Description of Time Series Snapshots (TSS) Model

The behaviors of a TCP flow i bounded by bandwidth BW_i are modeled in a queuing model (Fig. 6a). The model indicates the packets ready to be sent by the sender, queued at the edge gateway, and sent by the edge. Fig. 6b displays a sample snapshot of the model taken at the end of a D_{w_i} (assumed to be a constant). Each tiny diamond stands for a packet, with the bounding rectangle specifying the packet size in the vertical axis and the time at which the packet is sent by the edge gateway within that D_{w_i} in the horizon axis. Each diamond is also accompanied by a number indicating its sequence within its CWND round (defined in Section 2.4.2). Packets one to four can be scheduled out within the D_{w_i} and are evenly distributed across the D_{w_i} if the scheduler clocks out packets using a fine granularity. Packets five to eight are not allowed to be forwarded within the D_{w_i} and, thus, are queued at the edge; packets with numbers above eight are queued by the sender if the window size ($= \min(W_{c_i}, W_{r_i})$) is eight ($W_{c_i} = 11, W_{r_i} = 8$). Queued packets (i.e., numbered 5 to 11) are aligned to the right of the D_{w_i}.

The time series of snapshots (TSS) model (Fig. 6c) is comprised of consecutive snapshots. TSS is a packet-level model analogous to the model in [12], but contains extra details about each packet, such as the time each packet is sent within an D_{w_i}, as well as its location (at sender, edge gateway, or WAN pipe). Packets with the same shape belong to the same CWND round, as shown in Fig. 6b. Since bulk data transfers are assumed herein, the packet size should generally be MSS. Once packets with the same shape (i.e., the same CWND round) are completely scheduled out and the last of their corresponding ACKs returns, the CWND is advanced with a full-size packet. If W_{c_i} grows beyond W_{w_i}, packets with the same shape cannot be completely scheduled out during a single D_{w_i} and, thus, excessive packets are overflowing to the edge gateway for sending in the next D_{w_i}. Packets queued at the edge in previous D_{w_i}s are darkened.

4.1.2 Behaviors of PostACK-Applied TCP Flows

Because PostACK emulates the queuing delay of per-flow queuing (PFQ) in the reverse ACK direction, the PostACK-applied TCP source behaviors are essentially the same with PFQ. In the congestion-avoidance phase, PostACK/PFQ-applied TCP source i raises its window *linearly* until W_{w_i} is reached. After that, a *curved* increasing phase slows down the evolution of W_{c_i}, as modeled in Fig. 6c. Continuous

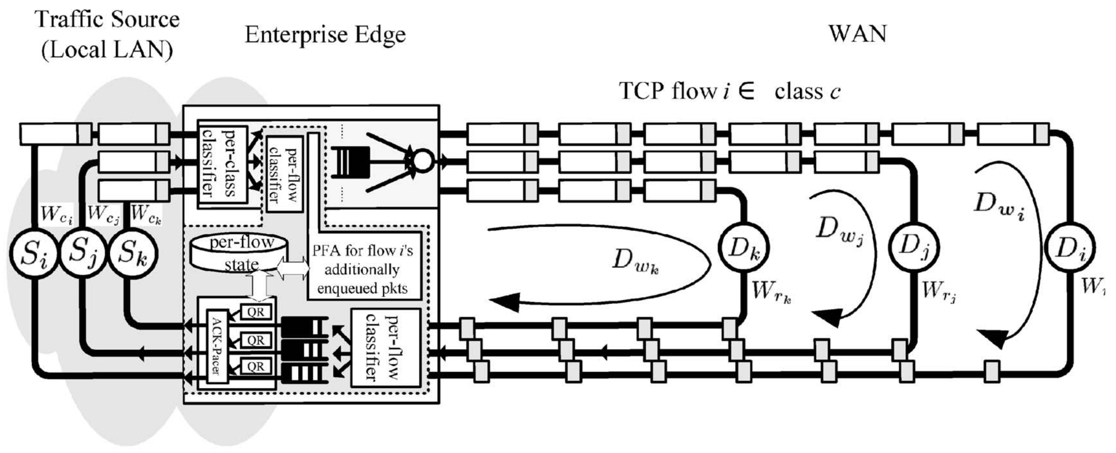


Fig. 5. Efficient PostACK implementation for managing outgoing TCP traffic. Terms: QR: queue relocator; PFA: per-flow accounting.

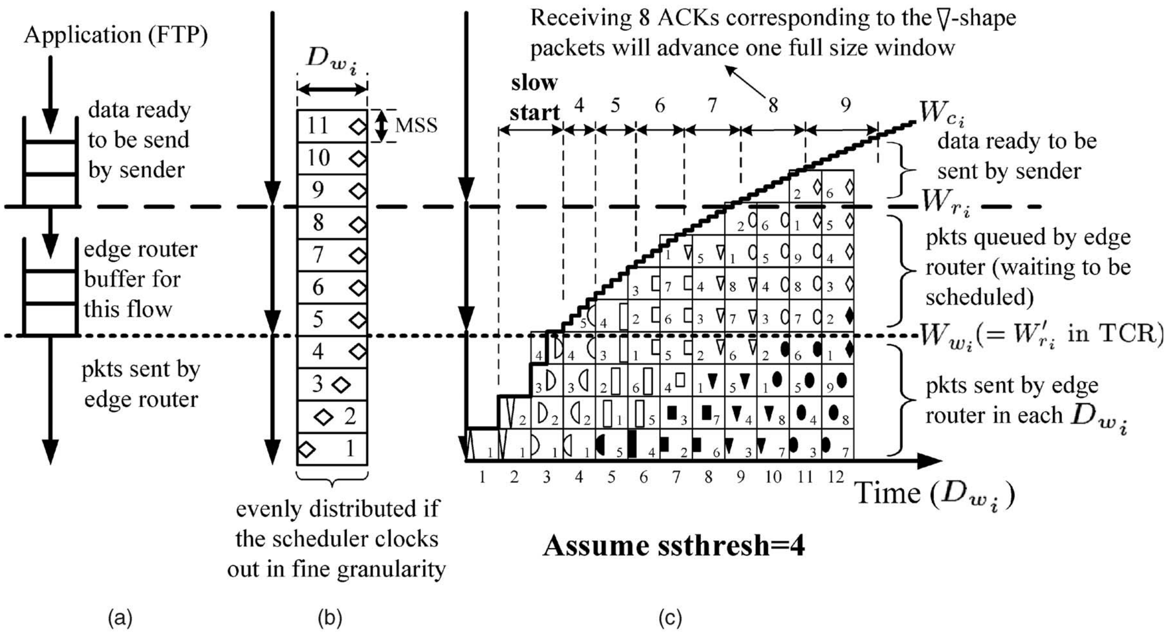

Application (FTP)
data ready to be send by sender
edge router buffer for this flow
pkts sent by edge router
D_{w_i}
MSS
evenly distributed if the scheduler clocks out in fine granularity
Receiving 8 ACKs corresponding to the ∇-shape packets will advance one full size window
slow start
W_{c_i}
data ready to be sent by sender
W_{r_i}
pkts queued by edge router (waiting to be scheduled)
$W_{w_i}(= W'_{r_i}$ in TCR)
pkts sent by edge router in each D_{w_i}
Time (D_{w_i})
Assume ssthresh=4
(a)
(b)
(c)

Fig. 6. Microscopic view of the TSS model. (a) Queuing model. (b) Snapshot at the end of a DW_i. (c) Time series of snapshots (TSS) model.

increasing of W_{c_i} but bounded by a fixed W_{w_i} results in the change from linear phase into curved phase. Between W_{w_i} and $\min(W_{c_i}, W_{r_i})$ is exactly what the gateway buffers for the flow. Thus, a large advertised W_{r_i} [7] or a small W_{w_i} result in high buffer requirement at the gateway, i.e., $\min(W_{c_i}, W_{r_i}) - W_{w_i}$ packets. The edge gateway may encounter severe buffer overflows under such conditions if the gateway ignore the above phenomenon. Formally speaking, (2) indicates that, when W_{c_i} grows beyond W_{w_i}, PFQ stretches the $PCDQ_i^{delay}$ by queuing the data packets. PostACK minimizes the buffer requirement by relocating the queuing delay of $PCDQ_i^{delay}$ to the reverse ACKs. Zero buffer is also feasible (Section 3.2).

4.1.3 *Behaviors of TCR-Applied TCP Flows*

In the TSS model (Fig. 6c), window-sizing resizes the W_{r_i} to W'_{r_i} $(= W_{w_i})$ and thus reduces the buffer requirement and latency at the edge. According to (2), as the window grows, TCR shrinks the W_{r_i} to W_{w_i} such that $PCDQ_i^{delay}$ and $PFAQ_i^{delay}$ both approximate zero.

4.2 Modeling the Throughput under Loss

The closed-form throughput for Per-Flow Queuing (PFQ) and TCR is derived here using the TSS model. Since PostACK does not shrink the window size, throughput of a PostACK-applied flow approximates that of a PFQ-applied one. Thus, only PFQ and TCR are considered. TCP modeling has received considerable attention, but the assumptions made differ significantly. Altman et al. [13] present a fairly good survey of the evolution and limitations of each model. Unfortunately, none of them address the performance of TCP under rate shaping gateways, though numerous bandwidth management gateways have been installed. Due to the limitation of space, the modeling and its verification using NS-2 simulations are briefly presented in Appendix C. The results confirm that the modeling closely approximates the simulations.

5 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 flow. We hereby describe the implementations and experimental results.

5.1 Tcp-masq and Wan-emu Open-Source Packages

Fig. 7 illustrates the protocol-stack view and bird's-eye view of the Tcp-masq and Wan-emu. All mechanisms are placed in kernel-space and configured by user-space tools. Memory disks and null devices (/dev/null) are used to avoid disk I/O overhead when logging/sniffing/storing traffic. Detailed descriptions about the tcp-masq and wan-emu testbed are self-contained in Fig. 7, Appendix D, and [28]. Because we focus on multiple TCP flows sharing a queue as assumed, the traffic load is always 100 percent due to TCP's aggressiveness.

5.2 Numerical Results

5.2.1 *Buffer Requirement at the Edge Gateway*

This section demonstrates the same effectiveness of PostACK and TCR in saving the buffer space. Fig. 8 quantifies the goodput degradation due to buffer overflow at the edge gateway. For a 500KB/s class, pure CBQ requires a huge buffer (3^5 packets at 2^5 flows) to achieve the same goodput (480kbps) as TCR and PostACK. The highest goodput is 480kbps because of TCP/IP header overheads. For PostACK and TCR, only a reasonable buffer (< 10 packets) is needed. Buffer overflow can cause high retransmission ratio (up to 22 percent when 2^5 flows compete for a 1-packet FIFO), which consumes a considerable amount of LAN bandwidth.

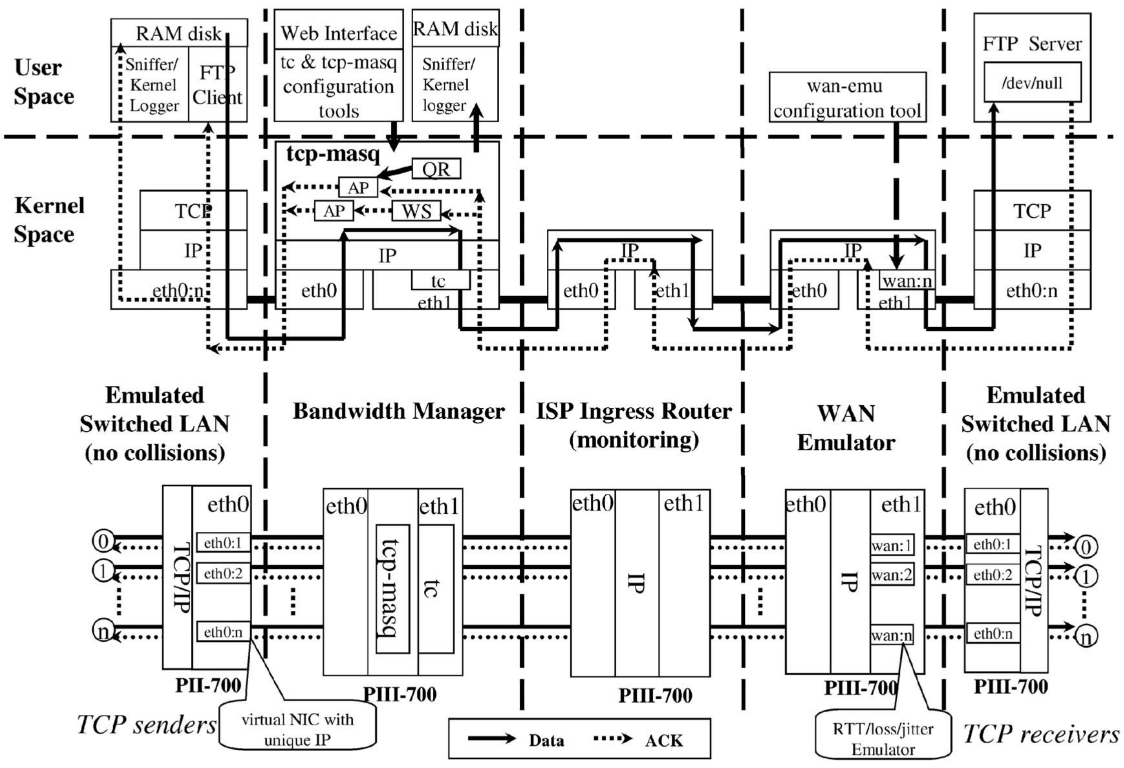


Fig. 7. Protocol-stack and bird's-eye views of Tcp-masq and Wan-emu. Description: The testbed topology emulates the well-konwn single bottleneck (or "dumbbell") emulation scenario.

5.2.2 Loss Behaviors (Sensitivity to Internet Loss)

Fig. 9 compares the goodput degradation due to packet losses in WAN. Under normal WAN loss (below 4 percent random loss), PostACK obviously outperforms TCR, but retains the same degree of TCP-friendliness with PFQ. A microscopic view of the bandwidth fluctuations in 0.5 percent random WAN loss (Fig. 9a, Fig. 9b, Fig. 9c) proves the benefit of PostACK against TCR. Fig. 9d shows the average goodput over 10-30 seconds. PostACK can have 10 percent improvement against TCR under 1 percent packet loss rate. Mathematical modeling for the improvement is analyzed in Appendix C. Under heavily congested conditions (beyond 4 percent 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.

5.2.3 Fairness among flows in One Class (Flow Isolation)

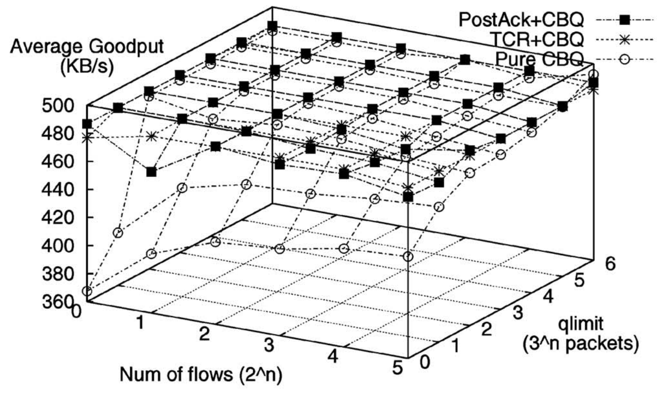
This section investigates the effectiveness of ACK control modules in resolving the unfairness among TCP flows with heterogeneous WAN delays. Test configurations are described in Fig. 10. Fig. 10a demonstrates the classical problem: Throughput of a TCP flow is inversely proportional to its RTT. However, when the three flows share a 200KB/s class in a FIFO PCDQ (Fig. 10b), the unfairness among the 10ms/50ms/100ms flows is alleviated. This is because the RTT measured by flow i (RTT_i) equals $D_{w_i} + \sum_i PCDQ_i^{delay}$. 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. 10c) and PostACK (Fig. 10d) 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.

5.2.4 Robustness under Various TCP Implementations

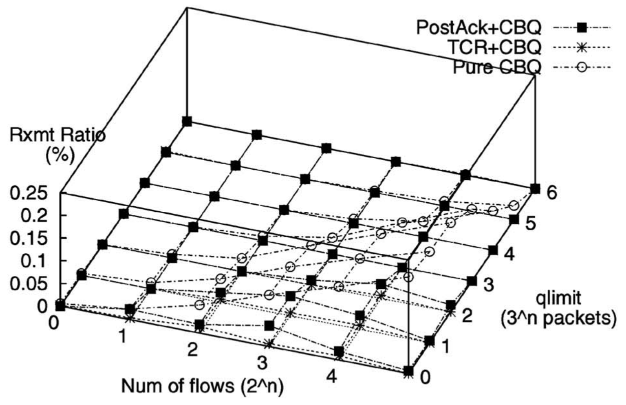
This section tests the robustness of TCR and PostACK under major TCP implementations. The test methodology is self-contained in Fig. 11. In Fig. 11a, Fig. 11b, the bandwidth policy constrains the unacknowledged packets in WAN to one ($W_w = 1$). The *Tiny-Window Side Effect* of TCR occurs in Fig. 11a. 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. 11c, Fig. 11d), 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. 11b, Fig. 11d) can keep the target rate regardless of TCP implementations.

5.2.5 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



(a)



(b)

Fig. 8. Buffer requirement of a 500KB/s class: The testbed is configured as WAN delay = 50ms, class bandwidth = 500KB/s, and default RWND = 32KB. (a) Average goodput of the class. (b) Retransmission ratio of the class.

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 Fig. 12. The results (Fig. 12a) confirm that throughput of TCR-applied flows suffers even under slight Internet losses. Fig. 12b displays the rate fluctuations of one data set. The PostACK scheme has the smallest degree of rate fluctuation.

5.2.6 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 could support 20,000 flows [9] in 1999 (it is now 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 be 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 do not introduce any new processing overhead, but only skip the flows that send more than expected.

Note that per-flow bandwidth management approaches such as TCR and PostACK should only be deployed at enterprise-side edges only. The access link speed is often too slow as some commercial TCR implementations are only equipped with a Pentium 133 MHz CPU [1]. However, some vendors also deploy their commercial TCR implementations for ISP edges. Actually, this may not be practical and could be only for marketing because 1) the ISP should only control the bandwidth among the subscribers rather than the bandwidth of TCP flows which belongs to organizational policies; 2) VPN tunnels using IPsec prevail so that everything beyond the IP header is encrypted by the enterprise and cannot be read/write by the ISP. For deployment at enterprise edge gateways, current PostACK and TCR scale well and are very practical solutions. However, the next section designs a scalable PostACK to overcome several challenges of PostACK and TCR.

6 Scalable Implementation of PostACK

The previous designs of PostACK/TCR require a timer for each bandwidth class. Maintaining multiple timers within a kernel system introduces overheads and side-effects. This section presents a zero-timer $O(1)$ PostACK with per-flow queuing that can eliminate the following drawbacks of PostACK:

1. *Only optimized for TCP traffic:* The above PostACK/TCR achieves the goal by regulating the ACKs. For flows without ACKs such as UDP and ICMP, PostACK and TCR are not usable.
2. *Nonscalable to gigabit networks:* When allocating a large bandwidth from a big WAN pipe, the timer can be too busy to pace out ACKs. Typically, modern operating systems use 1-10ms as one tick. For the most common largest packet size (1,500 bytes), such timer granularity can only achieve at most 1,500*8/0.001 bps. Though PostACK/TCR can be designed to pace out more ACKs at a time, the solution may degrade their fine-grained fairness.
3. *Inefficient bandwidth borrowing:* When sharing newly available bandwidth among classes within a link (interclass bandwidth borrowing) or among flows within a class (intraclass bandwidth borrowing), the timers to release ACKs should be adjusted according to dynamically borrow bandwidth from nonactive classes or flows, respectively. Otherwise, inefficient bandwidth borrowing will waste the newly available bandwidth.

Per-flow queuing (PFQ) within each bandwidth class is the most general approach to overcome the first drawback. However, pure PFQ still queues a large number of TCP packets. So, we apply the PostACK to each per-flow queue to minimize the buffer requirement. The PFQ can be simply implemented using the $O(1)$ deficit round robin (DRR) with equal quantum size for each queue to achieve the assumption of fairness among flows within a class (Section 1.2). Three design strategies to overcome the other drawbacks are:

1. *Round-robin within a class to achieve intraclass bandwidth borrowing:* Without having to detect if any flow joins/leaves the bandwidth class and then adjusts the ACK-pacing timer accordingly, round-robin servicing each per-flow queue can seamlessly distribute bandwidth among active flows.

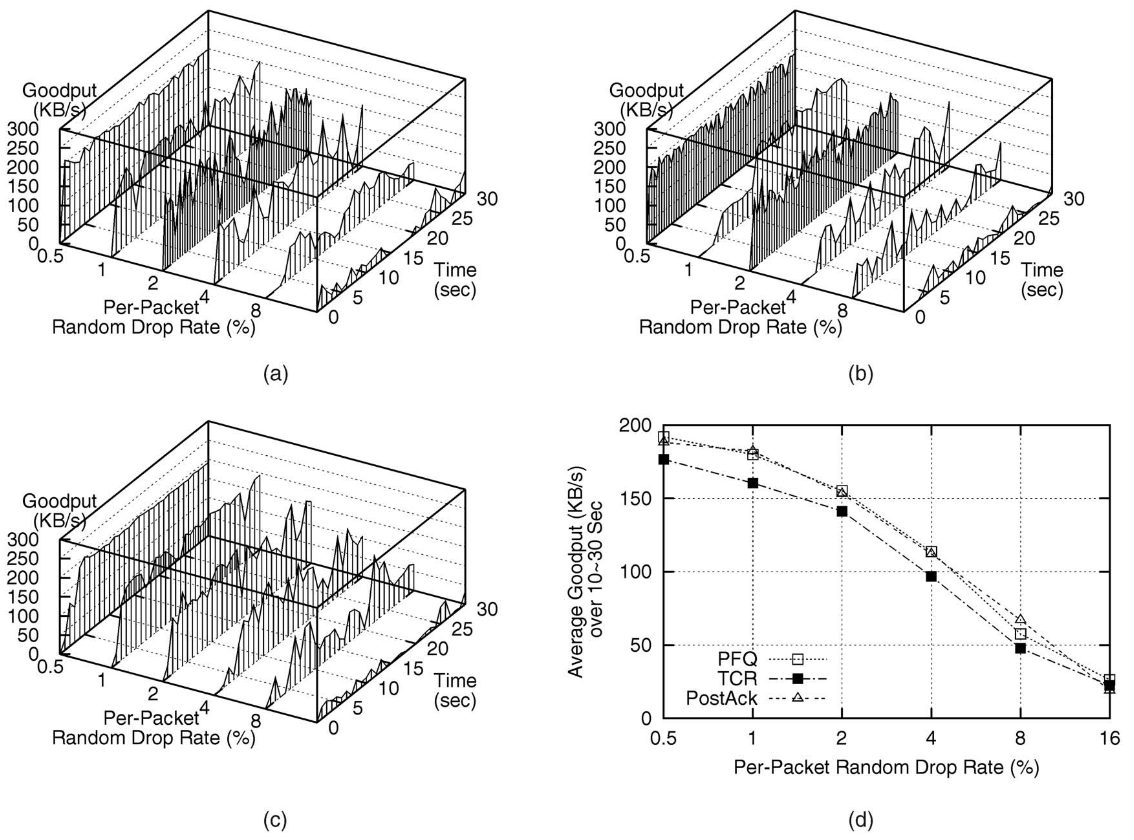


Fig. 9. Goodput degradation under various WAN loss rates. Configuration: A flow bottlenecked by a 200KB/s class runs under various random packet loss rates set by the WAN emulator. (a) Goodput under per-flow queuing. (b) Goodput under TCR. (c) Goodput under PostACK. (d) Average goodput.

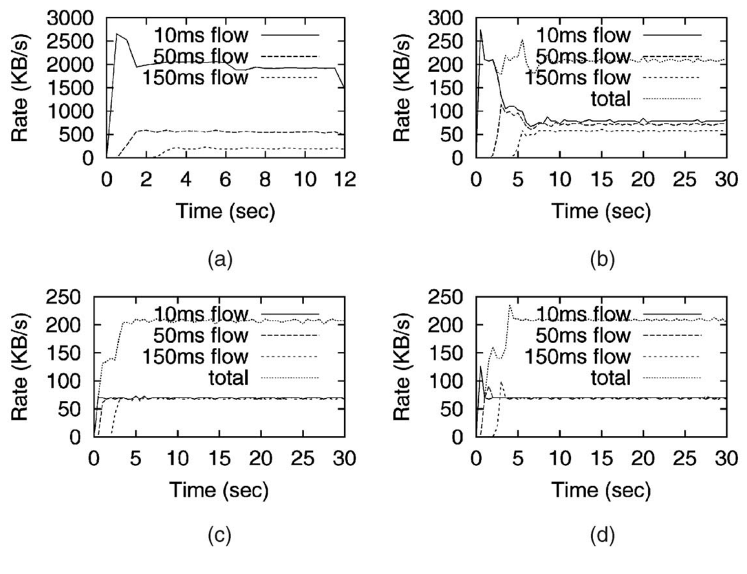


Fig. 10. Fairness among flows in 200kB/s class. Configuration: Three paths are configured as different WAN delays (10ms, 50ms, and 150ms) in Wan-emu. In (a) each flow sends packets at its own will; in (b), (c), (d) a 200KB/s class contains the three flows ((b) CBQ, (c) CBQ+TCR, (d) CBQ+PostACK). These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

2. *Keeping CBQ backlogged whenever possible to achieve interclass bandwidth borrowing:* Only when the bandwidth class c's PCDQ has packets can the class c compete for the extra newly available bandwidth.
3. *Everything clocked by PCDQ_Dequeue to scale:* Without employing extra timers, PostACK can scale with the link-sharing systems (e.g., CBQ).

Fig. 13 illustrates our 2-stage integration: 1) adding PFQ to the CBQ; 2) adding PostACK to the PFQ. The evolutions of the data path are self-contained in Fig. 13. Fig. 14 describes the algorithm of the above integration.

When the administrator sets a new policy to enforce per-flow bandwidth guarantee (Seciton 1.1.2) for each TCP flow in class c (Fig. 13), the PFQ_PCDQ_Enqueue(m,c) (Step 2.1

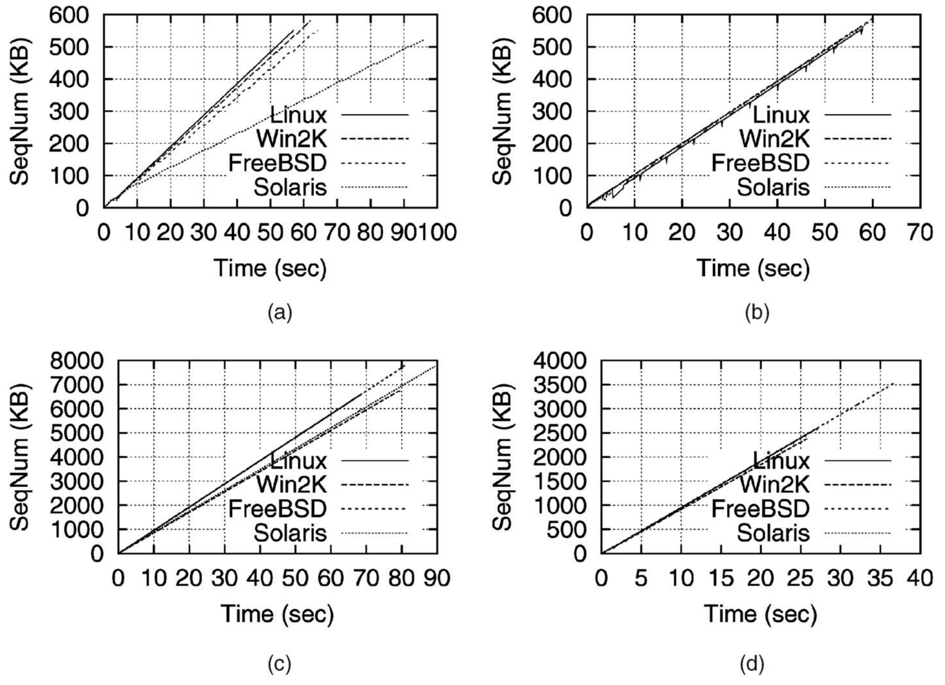


Fig. 11. Robustness under various TCP sender implementations: TCR versus PostACK. Testbed Configurations: round-trip WAN delay = 50ms, MSS = 1,500 bytes. In (a), (b), periodic drop rate in WAN = 1/40, class bandwidth = 10KB/s, and, thus, $W_w < 4$; in (c), (d), periodic drop rate = 1/100, class bandwidth = 100KB/s, and, thus, $W_w > 4$. Linux 2.2.17, FreeBSD 4.0, Solaris 8, and Windows 2000 are tested. (a) TCR: Ww = 1. (b) Post ACK: Ww = 1. (c) TCR: Ww = 5. (d) PostACK: Ww = 5.

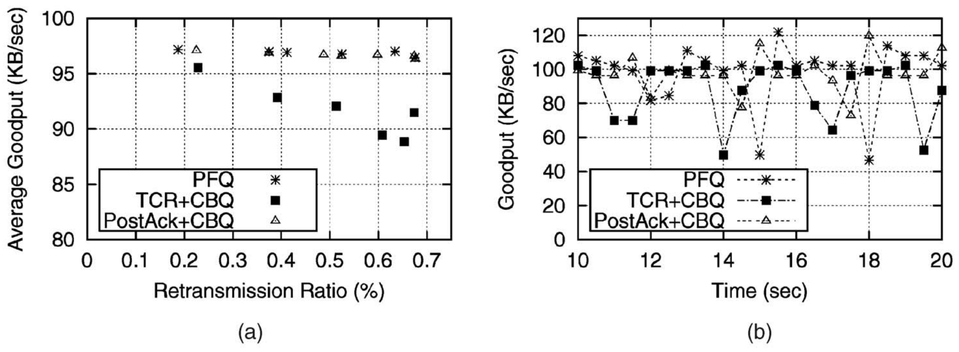


Fig. 12. Out site-to-UCLA live experiments. Configuration: The live experiments were conducted seven times. Each time a 100KB/s TCP flow is separately run over each scheme for 60 seconds. Data from 10 to 50 seconds is analyzed. The loss rate ranges from 0.002 to 0.007, with corresponding WAN delays ranging from 102 ms to 168 ms. (a) Average goodput. (b) A snapshot of bandwidth.

in Fig. 13) hooks before the PCDQ_Enqueue(m,c) to decide whether to enqueue packet m into c's PCDQ or m's PFQ. Step 2.1 ensures a backlogged PCDQ to compete for any newly available bandwidth released from other link-sharing (e.g., CBQ) classes. Subsequently, any PCDQ_Dequeue(c) kicks out one packet, but then immediately fetches another packet from the next DRR-selected PFQ into its PCDQ. So far, the PFQ can provide per-flow guarantee. In order to further minimize PFQ's buffer requirement, PostACK in Step 2.4 further checks the DRR-selected PFQ against a table to relocate the queuing delay: 1) If the queue length of the DRR-selected PFQ is becoming large ($qlen > 1$), PostACK does not release the flow's ACK to queue more ACKs in the reverse direction; 2) if $qlen == 1$, PostACK simultaneously releases the flow's data packet and ACK; 3) if $qlen == 0$ (such as when at the end of a transfer), PostACK releases its ACK.

The numerical results show that: 1) The scalable PostACK with PFQ effectively minimizes the buffer requirement as in previous PostACK results (Section 5.2). We do not show the results again because they are too similar. 2) PostACK integrated with CBQ in ALTQ 3.1 on NetBSD 1.5.2 equipped with Gigabit Ethernet cards can scale with the link-sharing mechanism up to 750Mbps.

7 Conclusions

This study 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 percent shown in Section 2.4.3) 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

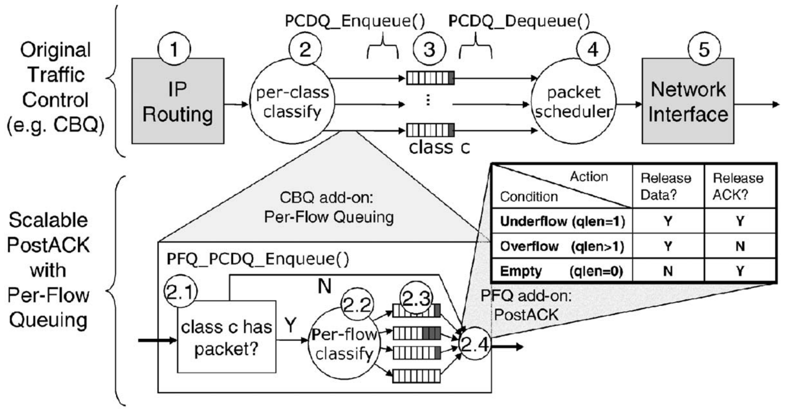


Fig. 13. Modularized integration of scalable PostACK with per-flow queuing into existing link-sharing architectures. (a) Original data path: Packets pass through Steps 1, 2, 3, 4, and 5 (e.g., Linus Kernel, ALTQ in BSD kernels). (2) PFQ data path: Packets folow Steps 1, 2, 2.1, 2.2, 2.3, 2.4, 3, 4, and 5. Step 2.4 is triggered by CBQ's PCDQ_Dequeue call and selects the next packet by the $O(1)$ deficit round-robin (DRR). (3) PostACK with PFQ data path: Step 2.4 further checks the state of the selected packets against the table to judge whether to relocate the queuing delay or not.

```
Initialization:
  SelectNextPFQ(c) select the next backlogged PFQ in class c
Algorithm:
01  PFQ_PCDQ_Enqueue(pkt m, class c){ /*enqueue to PCDQ or PFQ*/
02      if (PCDQ for class c is backlogged)
03          $i$=PerFlowClassify(m);
04          PFQ_Enqueue(m,$i$.PFQ);
05      else
06          PCDQ_Enqueue(m,c);
07  }
08  PCDQ_Dequeue(class c){ /* dequeue from $PCDQ_c$ */
09      /* original PCDQ_Dequeue code goes here */
10      After line 09 successfully dequeuing a packet {
11          $i$=SelectNextPFQ(c);
12          if ($i$.qlen>1) /*the PFQ is OVERFLOW*/
13              PCDQ_Enqueue(PFQ_Dequeue($i$),$c$);
14          else if ($i$.qlen==1) /*the PFQ is UNDERFLOW*/
15              PCDQ_Enqueue(PFQ_Dequeue($i$),$c$);
16              ReleaseACK($i$);
17          else if ($i$.qlen==0) /*the PFQ is EMPTY*/
18              ReleaseACK($i$);
20      }
21  }
```

Fig. 14. Scalable PostACK with PFQ Algorithm.

robust and simple approach, PostACK, is hereby proposed (Section 3) to combine the virtues of TCR (good fairness, low buffer/cost/latency) and PFQ (better performance under loss) without the drawbacks of TCR. Also, the throughput and buffer requirement of each scheme are modeled through the TSS model (Section 4.1). A further scalable design of PostACK can scale up to 750Mbps while seamlessly cooperating with the link-sharing architecture. All numerical results can be reproduced through our open sources [28]: 1) Tcp-masq: a modified Linux kernel that implements TCR and PostACK; 2) Wan-emu: a practical testbed for conducting LAN-to-WAN experiments with delay/jitter/loss emulations (Appendix D). Notice that PostACK/TCR is not limited to only applying on CBQ, but should 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 1 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 trade offs 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 2 compares the implementation complexities among the three. Note that PFQ cannot be $O(1)$ when using

TABLE 1
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

a fine-grained packet scheduler such as WFQ. They all require $O(N)$ space, where N denotes the number of TCP flows passing through.

Appendix A
Inbound TCP Traffic Control

Inbound traffic control, though important, is less effective since traffic flows have already *traversed the WAN access link* and *consumed the link bandwidth* before the edge gateway can control them. With queuing at the inbound direction, traffic can be shaped at its configured rate. However, since the traffic has arrived and consumed the WAN link bandwidth, shaping the traffic is less meaningful. TCR or PostACK can achieve inbound TCP traffic control by simply reversing their modules. However, they are also less effective because the evenly clocked-out ACKs have to traverse the dynamics in the Internet to trigger new data packets coming back. Their effectiveness still requires further study.

Appendix B
Modeling the Window Evolution in the TSS Model

the TSS model (Fig. 6c) facilitates the modeling of the CWND evolution. This can further analyze the per-flow throughput and buffer requirement of each scheme at the edge gateway. The curve in the TSS can be modeled as follows: Given the current CWND as w ($w \geq W_w$, $w = 1, 2 \dots$), the number of consecutive D_{w_i} without packet loss as x, and (4), we can write the following inequality according to the TSS (Fig. 6):

$$w + (w+1) + (w+2) + \cdots + (w+n-1) \leq x \cdot W_w,$$
$$n = 0, 1, 2 \dots,$$

TABLE 2
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

where n is the number of full-sized windows raised during x RTTs. Letting $g(w, x)$ be the CWND after x RTTs, we can derive $g(w, x)$ as

$$\begin{aligned} g(w,x) &= initial + largest\ possible\ n \\ &= w + \left\lfloor \frac{1 - 2w + \sqrt{(2w-1)^2 + 8xW_w}}{2} \right\rfloor . \end{aligned}$$

Appendix C
Modeling the Throughput of TCP over PFQ and TCR

This section models TCP throughput over PFQ and TCR with the given periodic loss rate p and W_w of a flow. Throughput of PostACK-applied flows are approximated by PFQ-applied flows since they both do not shrink W_r. Though TSS is analogous to the stochastic model in [12], deriving a closed-form throughput is difficult because the window evolution differs among cases. Since Altman et al. [13] have shown that the random loss events assumed in [12] imply a *deterministic* interval between loss events, we model the throughput by assuming a deterministic interval between congestion events. However, our modeling is verified through ns2 simulation with *random* loss.

Fig. 15 illustrates all possible steady-state cases of a PFQ/TCR-applied flow under periodic losses using the TSS model. We define the renewal point to be the time when the lost segment is ACKed. Let W_{c^k} be the congestion window at the end of the kth renewal cycle. The shaded areas correspond to the packets of the flow sent by the edge (described in Section 6). Each renewal cycle consists of three phases: the normal sending phase, buffer-draining phase, and retransmission phase, which are separated by the vertical dashed lines in Fig. 15. The ns2 simulation results confirm the accuracy of the modeling.

Given periodic loss rate p (a packet loss occurs every $\frac{1}{p}$ packets) and the W_w, we can derive to which among the four cases the W_{c^k} in its steady state belongs through the following claims, with the function $g(w, x)$ developed in Appendix B. For TCR, cases other than Fig. 15a are equivalent to Fig. 15d.

Claim 1. *If* $\frac{1}{p} \leq \frac{3}{8} W_w^2$ *and* $W_r \geq W_w \Longrightarrow E[W_c] \leq W_w$.

Short Proof. If $\frac{1}{p}$ packets cannot raise the W_c from $\frac{1}{2} W_w$ to W_w, a packet loss then halves the W_c down to $\frac{1}{2} W_w$ or lower, which will repeat the same process and result in a

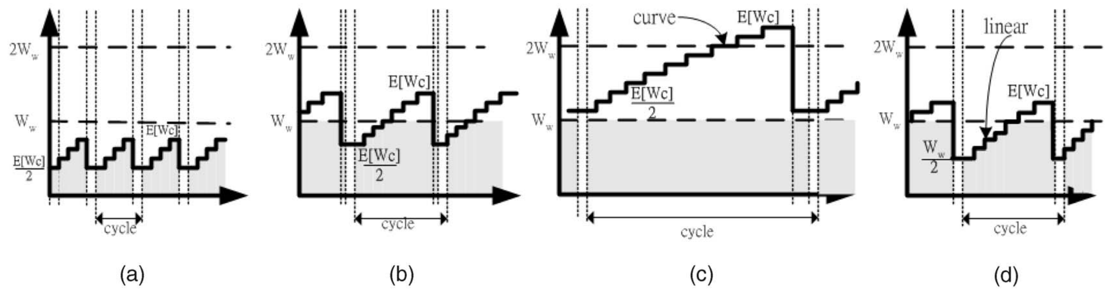


Fig. 15. Modeling the throughput: steady-state cases under periodic losses represented in the TSS model. (a) PFQ/TCR: Case 1. (b) PFQ: Case 2. (c) PFQ: Case 3. (d) TCR: Case 2.

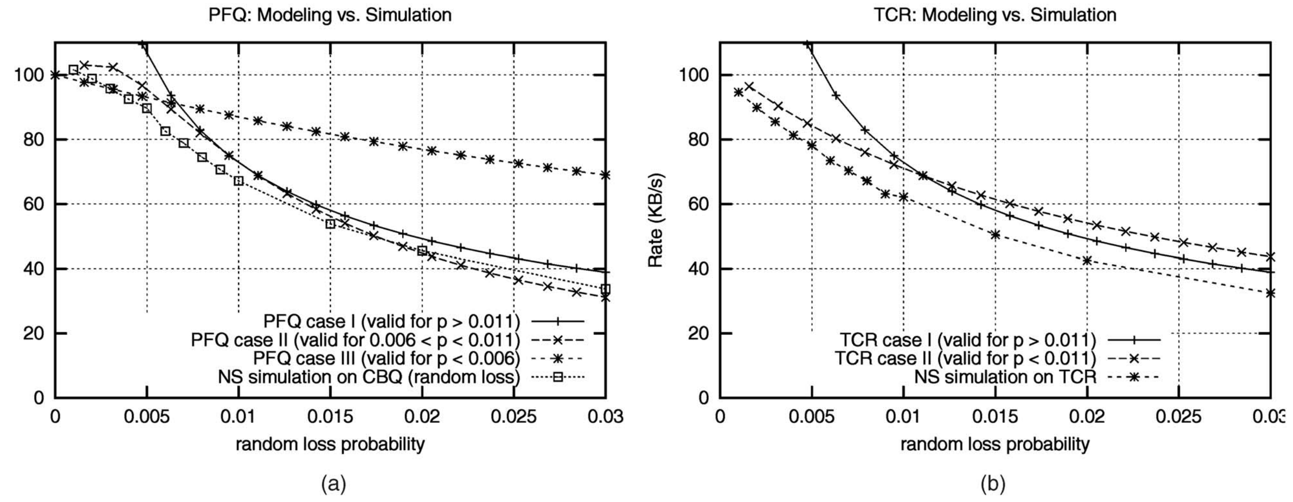


Fig. 16. Throughput of PFG and TCE: modeling versus simulation. Configuration: flow bandwidth = 100KB/s.

steady state indicated in Fig. 15a. The inequality can be arranged as $p \geq \frac{8}{3W_w^2}$. □

Claim 2. *If* $g(W_w, \frac{1/p}{W_w}) \leq 2W_w$ *and*

$$W_r \geq W_w \Longrightarrow W_w < E[W_c] \leq 2W_w.$$

Short Proof. If $\frac{1}{p}$ packets cannot raise the W_c from W_w to $2W_w$, a packet loss then halves the W_c below W_w, which will repeat the same process and result in a steady state indicated in Fig. 15b. The inequality can be arranged as $\frac{8}{3W_w^2} > p \geq \frac{2}{3W_w^2 - W_w}$. □

Claim 3. *If* $g(W_w, \frac{1/p}{W_w}) > 2W_w$ *and* $W_r \geq W_w \Longrightarrow E[W_c] > 2W_w$.

Short Proof. It is trivial from Claim 2 that this case will result in Fig. 15c. The inequality can be arranged as $p < \frac{2}{3W_w^2 - W_w}$. □

Given p and W_w for a flow, its steady state behavior can be determined through the above claims. The throughput T for a flow in each case can be derived by computing $(\frac{BytesSent}{Time})$ during a renewal cycle. In each renewal cycle, the shaded area is equal to the $\frac{1}{p}$ packets sent during the cycle. Each renewal cycle contains three phases, as described.

1. PFQ/TCR case 1: $\overline{W_c} < W_r$, and $\overline{W_c} < W_w$,

$$\frac{1}{p} = \left(\frac{\overline{W_c}}{2} + \overline{W_c}\right)\left(\frac{\overline{W_c}}{2}\right)\frac{1}{2} \Rightarrow \overline{W_c} = \sqrt{\frac{16}{3p}}$$

$$\Longrightarrow T = \frac{\frac{1}{p}}{\frac{\overline{W_c}}{2} + 1} = \frac{1}{\sqrt{\frac{2p}{3}} + p}.$$

2. PFQ case 2: $W_w < \overline{W_c} < 2W_w$, and $W_r > W_w$, let x be the period during normal sending phase and buffer-draining phase,

$$\begin{cases} g(W_w, x - (W_w - \frac{w}{2})) = w \\ \frac{1}{p} = \frac{(\frac{w}{2} + W_w)(W_w - \frac{w}{2})}{2} + \left[x - \left(W_w - \frac{w}{2}\right)\right]W_w + (w - W_w) \end{cases}$$

$$\Rightarrow \begin{cases} w = \frac{-2 + \sqrt{4 + \frac{24}{p} + 12W_w}}{3} \\ x = \frac{W_w}{2} + \frac{3}{2} + \frac{4}{3pW_w} + \frac{7}{9W_w} - (\frac{7}{9W_w} + \frac{1}{3})\sqrt{1 + 3W_w + \frac{6}{p}} \end{cases}$$

thus,

$$T = \frac{\frac{1}{p}}{x + 1} =$$

$$\frac{1/p}{\frac{W_w}{2} + \frac{3}{2} + \frac{4}{3pW_w} + \frac{7}{9W_w} - (\frac{7}{9W_w} + \frac{1}{3})\sqrt{1 + 3W_w + \frac{6}{p}} + 1}.$$

3. PFQ case 3: $\overline{W_c} > 2W_w$ and $\overline{W_c} > W_w$,

$$T = \frac{1}{\frac{1}{W_w} + p}.$$

4. TCR case 3: $W_r < \overline{W_c} < 2W_w$,

$$\frac{1}{p} = \left(\frac{W_w}{2} + W_w\right)\left(\frac{W_w}{2}\right)\frac{1}{2} + W_r{\cdot}x \Rightarrow x = \frac{1}{pW_w} - \frac{3}{8}W_w$$
$$\Rightarrow T = \frac{1/p}{\frac{W_w}{2} + x + 1} = \frac{1}{\frac{1}{W_w} + \frac{W_w}{8} + p}.$$

Fig. 16 verifies the modeling using NS-2 simulation. Each point is conducted on a single flow and the throughput is averaged over 2,000 seconds. Both Fig. 16a and Fig. 16b depict that the trends of simulation follows the valid portion of each case, but the modeling seems to over-estimate the expected throughput of PFQ and TCR by some constant factor. Elimination of the floor functions when solving the equations results in the mismatch. The TCP code in NS-2 takes a floor function such that the throughput is lower than that of the modeling.

No larger loss probability is demonstrated since, in those cases, the bottleneck no longer resides at the edge gateway. Their throughput can be characterized by classical models.

Appendix D

Descriptions of Our Wan-emu Testbed

Our testbed (Fig. 7) consists of cascaded machines with statically configured routes. IP-aliasing is used to emulate multiple competing senders and their receivers. Self-written WAN Emulator (a loadable Linux kernel virtual device driver) is to emulate the dynamics (delay, jitter, loss) of the Internet. A detailed description of the testbed is available at [1].

1. IP-aliasing support: In Linux, each network interface card (NIC) can emulate multiple virtual NICs, with each one having a unique IP address. With a proper routing table setup, we can direct flows destined to somewhere through some virtual NIC. Virtual NICs generate packets with their corresponding IP addresses such that the edge gateway will feel that outgoing packets are from different local hosts and incoming ACKs are from different remote hosts. What is more, packets are sent without link-layer collisions since only a single physical NIC is present at each side. By this we claim that it is a switched LAN-to-WAN testbed. Note that some operating systems, such as FreeBSD and Windows 2000, merely support aliased IP addresses but not aliased interface names.
2. Wan-emu virtual device driver: Each packet passing through is labeled with a timestamp indicating the time at which it is to be expelled. An interrupt is triggered every 1ms (tunable to 8192Hz in Linux) to examine how many packets are due and should be forwarded. Random/periodic loss rate and delay jitter are also implemented. Multiple Wan-emu devices can simultaneously be attached onto a NIC.

The tcp-masq runs over the ip-masq module (for NAT), thus tcp-masq can incorporate its fast per-flow classification. Furthermore, ip-masq is application-aware such that layer-7 protocols as FTP, CUSeeMe, RealAudio, and VDOLive traffic can also be identified for bandwidth management.

The CBQ used herein is from Linux kernel. Because the CBQ in Linux is not very accurate, a token bucket shaper is attached in each CBQ class. The PFQ used herein is achieved by applying a token bucket shaper to each flow. However, the scalable PostACK (Section 6) is implemented in ALTQ 3.1 on NetBSD 1.5.2 because the CBQ in ALTQ is very accurate.

Acknowledgments

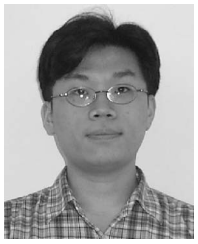
The authors thank Alan Cox at RedHat for his kind suggestions on attaching multiple Linux kernel virtual device drivers onto a network interface when designing the WAN emulator and Professor Mario Gerla for providing the TCP sinks at UCLA for live WAN experiments. This work was supported by MOE (Ministry of Education) Program of Excellence Research under project number 89-E-FA04-1-4 and L7 Networks Inc.

References

[1] H.Y. Wei and Y.D. Lin, "A Measurement-Based Survey and Comparison of Bandwidth Management Techniques," *IEEE Comm. Surveys and Tutorials,* to appear. http://speed.cis.nctu.edu.tw/bandwidth/BWSurvey.pdf, 2004.
[2] Y.D. Lin, H.Y. Wei, and S.T. Yu, "Integration and Performance Evaluation of Security Gateways: Mechanisms, Implementations, and Research Issues," *IEEE Comm. Surveys and Tutorials,* vol. 4, no. 1, Third Quarter 2002.
[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," RFC 2475, Dec. 1998.
[4] S. Floyd and V. Jacobson, "Link-Sharing and Resource Management Models for Packet Networks," *IEEE/ACM Trans. Networking,* vol. 3, no. 4, pp. 365-386, 1995.
[5] W.R. Stevens, *TCP/IP Illustrated Volume 1—The Protocols,* p. 289. Addison-Wesley, 1994.
[6] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit," RFC 3042, Jan. 2001.
[7] J. Mahdavi, "Enabling High Performance Data Transfers on Hosts," http://www.psc.edu/networking/perf_tune.html, 2003.
[8] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Networking,* vol. 1, no. 4, pp. 397-413, Aug. 1993.
[9] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer, "TCP Rate Control," *ACM Computer Comm. Rev.,* vol. 30, no. 1, Jan. 2000.
[10] D. Lin and R. Morris, "Dynamics of Random Early Detection," *Proc. ACM SIGCOMM '97,* Sept. 1997.
[11] S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Trans. Networking,* vol. 7, no. 4, pp. 458-472, Aug. 1999.
[12] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modelling TCP Throughput: A Simple Model and Its Empirical Validation," *Proc. ACM SIGCOMM '98,* Sept. 1998,
[13] E. Altman, K. Avrachenkov, and C. Barakat, "A Stochastic Model of TCP/IP with Stationary Random Losses," *Proc. ACM SIGCOMM '00,* Aug. 2000.
[14] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou, "Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP," *Proc. ACM SIGCOMM '02,* Aug. 2002.

[15] K. Fall and S. Floyd, "Simulation-Based Comparisons of Tahoe, Reno, and SACK TCP," *ACM Computer Comm. Rev.*, vol. 26, no. 3, pp. 5-21, July 1996.
[16] L.S. Brakmo, S.W. O'Malley, and L.L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," *Proc. ACM SIGCOMM '94,* Sept. 1994.
[17] M. Mathis and J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control," *Proc. ACM SIGCOMM '96,* Aug. 1996.
[18] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581, Apr. 1999.
[19] D. Stiliadis and A. Varma, "Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms," *IEEE/ACM Trans. Networking,* vol. 6, no. 5, pp. 611-624, Oct. 1998.
[20] J.C.R. Bennett and H. Zhang, "Hierarchical Packet Fair Queuing Algorithms," *Proc. ACM SIGCOMM '96,* Aug. 1996.
[21] L. Kalampoukas, A. Varma, and K.K. Ramakrishnan, "Explicit Window Adaptation: A Method to Enhance TCP Performance," *Proc. IEEE INFOCOM '98,* Apr. 1998.
[22] N.T. Spring, M. Chesire, M. Berryman, and V. Sahasranaman, "Receiver Based Management of Low Bandwidth Access Links," *Proc. INFOCOM '00,* Apr. 2000.
[23] L. Zhang, S. Shenker, and D.D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," *Proc. ACM SIGCOMM '91,* Sept. 1991.
[24] P. Narváez and K.Y. Siu, "An Acknowledge Bucket Scheme for Regulating TCP Flow over ATM," *Proc. IEEE GLOBECOM '97,* Nov. 1997.
[25] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," *Proc. INFOCOM '00,* Apr. 2000.
[26] Packeteer, Inc., http://www.packeteer.com, 2003.
[27] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queuing Algorithm," *Proc. ACM SIGCOMM '89,* Sept. 1989.
[28] "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/, 2002.

Huan-Yun Wei received the BS and PhD degrees in computer and information science from National Chiao Tung University, Hsinchu, Taiwan, in 1998 and 2003, respectively. His research interests include TCP rate shaping for enterprise edge devices, queuing analysis of TCP traffic shaping, high-speed packet classification, and performance analysis of network security gateways. He is especially interested in the design and implementation of FreeBSD/NetBSD/Linux kernels.



Shih-Chiang Tsao received the BS and MS degrees in computer and information science from National Chiao Tung University, Hsinchu, Taiwan, in 1997 and 1999, respectively. He worked as an associated researcher at Chung-Hwa Telecom from 1999 to 2003, mainly to capture and analyze switch performance. He is currently pursuing the PhD degree in computer and information science at National Chiao-Tung University. His research interests include TCP-friendly congestion control algorithms and fair queuing algorithms.

Ying-Dar Lin received the Bachelor's degree in computer science and information engineering from National Taiwan University in 1988 and the MS and PhD degrees in computer science from the University of California, Los Angeles in 1990 and 1993, respectively. His research interests include design, analysis, and implementation of network protocols and algorithms, wire-speed switching and routing, quality of services, network security, and content networking. Dr. Lin is a member of the ACM and IEEE. He is the founder and director of the Network Benchmarking Lab (NBL) which reviews the functionality, performance, conformance, and interoperability of networking products.