# Low-Latency Adaptive Streaming over TCP

ASHVIN GOEL
University of Toronto
CHARLES KRASIC
University of British Columbia
and
JONATHAN WALPOLE
Portland State University

20

Media streaming over TCP has become increasingly popular because TCP's congestion control provides remarkable stability to the Internet. Streaming over TCP requires adapting to bandwidth availability, but unforunately, TCP can introduce significant latency at the application level, which causes unresponsive and poor adaptation. This article shows that this latency is not inherent in TCP but occurs as a result of throughput-optimized TCP implementations. We show that this latency can be minimized by dynamically tuning TCP's send buffer. Our evaluation shows that this approach leads to better application-level adaptation and it allows supporting interactive and other low-latency applications over TCP.

Categories and Subject Descriptors: C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet (e.g., TCP/IP)*; D.4.4 [**Operating Systems**]: Communications Management—*Buffering and network communication*

General Terms: Measurements, Performance

Additional Key Words and Phrases: TCP, low latency streaming, multimedia applications

**ACM Reference Format:**

Goel, A., Krasic, C., and Walpole, J. 2008. Low-latency adaptive streaming over TCP. ACM Trans. Multimedia Comput. Commun. Appl. 4, 3, Article 20 (August 2008), 20 pages. DOI = 10.1145/1386109.1386113 http://doi.acm.org/10.1145/1386109.1386113

## 1. INTRODUCTION

Media streaming applications are increasingly using TCP as their transport protocol because TCP, the most common transport protocol on the Internet today, offers several benefits for media streaming. It provides congestion-controlled delivery, which is largely responsible for the remarkable stability of the Internet despite an explosive growth in traffic, topology and applications. TCP handles flow control and packet losses so that applications do not have to explicitly perform packet loss recovery. This issue is especially important because the effects of random packet loss can quickly become severe. For instance,

Authors' addresses: A. Goel, Electrical and Computer Engineering, University of Toronto, 10 King's College Rd., Toronto, ON M5S 3G4, Canada; email: ashvin@eecg.toronto.edu; C. Krasic, University of British Columbia, 2366 Main Mall, Vancouver BC V6T 1Z4, Canada; email: krasic@cs.ubc.ca; J. Walpole, Computer Science, Portland State University, PO Box 751, Portland, OR 97207; email: walpole@pdx.edu.

loss of the header bits of a picture typically renders the whole picture and possibly a large segment of surrounding video data unviewable. Thus, media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [Rizzo 1997] that potentially have high bandwidth and processing overhead even when packets are not lost.

The main challenge in using TCP for media streaming is that the application must adapt media quality in response to TCP's estimate of current bandwidth availability. Otherwise, the stream can be delayed indefinitely when the available bandwidth is below the stream's bandwidth needs. Non-adaptive streaming protocols compensate for this problem with large buffering, but this approach still requires that users choose the appropriate media quality (e.g., for a 56-kb/sec, 1.5-mb/sec or 10-mb/sec connection) and the client-side buffer size (e.g., 20 sec). Both these choices are hard because a static value does not reflect the dynamic nature of bandwidth availability.

Adaptive streaming applications solve the varying bandwidth problem by adapting the quality of the media based on available bandwidth using techniques such as prioritized data dropping and dynamic rate shaping [Rejaie et al. 1999; Feng et al. 1999; Krasic et al. 2003]. The effectiveness of the quality adaptation depends on the delay in the bandwidth feedback. In particular, the application can adapt its transmission requirements to the available bandwidth more accurately with low delay feedback. Unfortunately, TCP can introduce significant latency at the application level. Therefore, the application must make adaptation decisions far in advance of data transmission, which makes the adaptation unresponsive and perform poorly as the available bandwidth varies over time.

In this article, we show that the latency introduced by TCP at the application level is not inherent in TCP. Instead, a significant part of the latency occurs on the sender side of TCP as a result of throughput-optimized TCP implementations. Then, we develop an *adaptive buffer-size tuning* technique that reduces this latency. This technique helps streaming applications because it dramatically reduces the end-to-end latency experienced by streaming applications, especially when the network is loaded. The reduced latency improves the responsiveness of adaptation and hence the quality observed by the application. Our experiments also show reduced variance in throughput so that the streaming media has smoother quality. Consequently, our approach enables low-latency media streaming over TCP and benefits interactive applications such as conferencing as well as applications that transmit prioritized data over the same connection. For example, in a pervasive computing [Kozuch and Satyanarayanan 2002] or an interactive environment (e.g., a remote VNC desktop [RealVNC Limited 2002]), data could be transmitted with high priority while an ftp or another long-lived flow could be transmitted with low priority on a virtual connection. Similarly, media control operations such as the sequence of start play, fast forward, and restart play can be more responsive because the network and the end-points have low delay in the data path.

We demonstrate these benefits by extensively evaluating our scheme using a real adaptive-streaming application. While buffer-size tuning reduces latency, it also reduces network throughput. We explore the reasons for this effect and then propose a simple enhancement that allows trading latency and network throughput. Our approach changes the TCP implementation but does not change the TCP protocol and is thus attractive in terms of deployment.

The next section presents related work in the area. Section 3 analyzes the ways in which TCP introduces latency. Then, Section 4 introduces our adaptive send-buffer technique for reducing TCP latency. Section 5 explains how this technique affects TCP throughput and then extends our approach to allow trading between latency and throughput. Then, Section 6 describes our implementation. Section 7 presents the benefits of our approach for a real adaptive streaming application. Finally, Section 8 justifies our claims about the benefits of our buffer-size tuning approach for low-latency streaming over TCP.

## 2. RELATED WORK

Researchers in the multimedia and networking community have proposed several alternatives to TCP [Allman et al. 1999] for media streaming [Stewart et al. 2000; Floyd et al. 2002; Kohler et al. 2006]. These alternatives aim to provide TCP-friendly congestion control for media streams without providing reliable data delivery and thus avoid the latency introduced by packet retransmissions. Unfortunately, the effects of packet loss on media streaming are nonuniform and can quickly become severe. For instance, loss of the header bits of an *I*-frame in an MPEG movie can render a large segment of surrounding video data unviewable. Thus, media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [Rizzo 1997] that potentially have high bandwidth and processing overhead. The benefit of FEC schemes for loss recovery is that they often have lower latency overhead as compared to ARQ schemes such as employed in TCP. Thus, Nonnenmacher et al. [1998] explores introducing FEC as a transparent layer under an ARQ scheme to improve transmission efficiency.

DCCP [Kohler et al. 2006] implements congestion-controlled, unreliable flow of datagrams suitable for use by streaming media applications. It features reliable handshakes and acknowledgments but it does not retransmit data packets. It is a packet stream protocol that requires application-level framing similar to our use of application-level framing over TCP (see Section 6.3). The goals of DCCP are very similar to our goals, but we do not require a new transport protocol. The main difference between DCCP and our approach is related to packet dropping. With buffer tuning, we minimize the negative impacts of packet dropping and, in addition, we use ECN to minimize packet dropping. DCCP drops packets instead of retransmitting them. This can be beneficial for delayed packets but handling dropped packets can introduce significant complexity at the application level. DCCP provides multiple different congestion control algorithms that could be combined with buffer tuning to further improve latency.

Popular interactive streaming applications include Voice over IP (VoIP) products such as Microsoft NetMeeting [NetMeeting]. NetMeeting provides reasonable voice quality over a best effort network but was originally implemented over UDP because the delays introduced by TCP were considered unacceptable. This article shows that TCP buffer tuning yields acceptable delays, especially for quality adaptive applications. Skype [Skype], a popular VoIP application, uses several UDP and TCP ports for communication especially in restricted conectivity scenarious such as with NATs.

The feasibility of TCP-based media streaming for stored media (e.g., movies) has been studied by several researchers. These approaches typically require adaptation of the media quality in response to TCP's estimate of bandwidth availability. Generally, the trade-off in the adaptation is short-term improvement in video quality versus long-term smoothing of quality. Rejaie et al. [1999] uses layered video and adds or drops video stream layers to perform long-term coarse grained adaptation, while using a TCP-friendly congestion control mechanism to react to congestion on short-time scales. Feng and Krasic [Feng et al. 1999; Krasic et al. 2003] use priority-based streaming, which allows a more flexible implementation of quality adaptation.

Various researchers have explored the used of TCP-friendly congestion control schemes [Bansal et al. 2001] such as TCP-Friendly Rate Control (TFRC) [Floyd et al. 2000] and AIMD with different linear constants from TCP [Yang and Lam 2000] to reduce variation in media quality. These schemes can be combined with our buffer tuning approach to further improve the latency and jitter of streaming applications.

Our send-buffer adaptation approach is similar to the buffer tuning work by Semke et al. [1998]. Semke tunes the send buffer size to between two to four times the congestion window of TCP to improve the throughput of a high bandwidth-delay connection that is otherwise limited by the send buffer size. The buffer size value is chosen to limit small, periodic fluctuations in buffer size. This paper shows that

a connection can achieve throughput close to TCP throughput by keeping the send buffer size slightly larger than the congestion window and also achieve significant reduction in latency.

Active queue management and explicit congestion notification (ECN) [Ramakrishnan et al. 2001] have been proposed for improving the packet loss rates of TCP flows. Salim and Almed [2000] show ECN has increasing throughput advantage with increasing congestion levels and ECN flows have hardly any retransmissions. Feng [Feng et al. 1997] shows that adaptive active queue management algorithms (Adaptive RED) and more conservative end-host mechanisms can significantly reduce loss rates across congested links. In this article, we show that ECN can be combined with our buffer-tuning technique to further improve latency.

Many differentiated network services have been proposed for low-latency streaming. These schemes are complementary to our adaptive tuning approach. Hurley and Le Boudec [1999] provides a low-delay alternative best-effort (ABE) service that trades high throughput for low delay. The ABE service drops packets in the network if the packets are delayed beyond their delay constraint. In this model, the client must recover from randomly dropped packets. Further, unlike with TCP, the server does not easily get back-pressure feedback information from the network in order to make informed quality adaptation decisions.

For interactive applications, ITU G.114 [ITU 1993] recommends 150 ms as the upper limit for one-way delay for most applications, 150 to 400 ms as potentially tolerable, and above 400 ms as generally unacceptable delay. The one-way delay tolerance for video conferencing is in a similar range, 200 to 300 ms.

## 3. TCP INDUCED LATENCY

In this section, we examine the various ways in which latency is introduced in a streaming application. The end-to-end latency in a streaming application consists of two main latencies: (1) *application-level* latency, that is, latency at the application level on the sender and the receiver side, and (2) *protocol latency*, which we define as the time difference from a write on the sender side to a read on the receiver side at the application level, that is, socket write to socket read latency. These latencies should be minimized so that the latency tolerance of a streaming application can be met.

In this article, we will mainly deal with reducing protocol latency although our evaluation considers application-level latencies also. TCP introduces protocol latency in three ways: *packet retransmission*, *congestion control* and *sender-side buffering*. To understand these latencies, we briefly describe TCP's transmission behavior.

TCP is a window-based protocol. It uses a window size, which is the maximum number of unacknowledged and distinct packets in flight in the network at any time. TCP stores the size of this current window in the variable CWND. When an acknowledgment (or ACK) arrives for the first packet that was transmitted in the current window, the window is said to have opened up, and then TCP transmits a new packet from a buffer on its sending side. This *send buffer* keeps copies of packets that are already in flight so that packets dropped by the network can be retransmitted. Given a network round-trip time (RTT), the throughput of a TCP stream is roughly CWND/RTT, because CWND packets are sent by TCP every round-trip time.

Since TCP is normally used in a best-effort network, such as the Internet, it must estimate bandwidth availability. To do so, it probes for additional bandwidth by slowly increasing CWND, and hence its transmission rate, by one packet every RTT. Eventually, the network drops a packet for this TCP stream, and TCP perceives this event as a congestion event. At this congestion event, TCP drops its CWND value by half to reduce its transmission rate. With this brief description of TCP, next we examine each of the ways in which TCP introduces latency.

*Packet Retransmission.* When packets are dropped in the network, they have to be retransmitted by TCP. Due to the round-trip time (RTT) needed for the congestion event feedback, these retransmitted packets are delayed by at least one RTT (multiple drops can cause multiple RTT delays). Further, since TCP is an in-order protocol, the receiving side does not deliver packets that arrive out-of-order to the application until the missing packets have been received. Hence, a retransmitted packet adds at least an additional RTT delay for CWND consecutive packets.

*Congestion Control.* TCP congestion control reduces the CWND value and thus its transmission rate in response to congestion feedback. Normally, TCP infers a network congestion event when it notices that a packet has been dropped. Hence, congestion control adds RTT delay for packets in flight, as discussed above. In addition, packets that are buffered on the sending side in the send buffer are delayed further since the sending rate (CWND) has decreased. For example, if CWND is halved, then the first lost packet is delayed by at least $1\frac{1}{2}$ RTT (at least one RTT for the dropped packet + $\frac{1}{2}$ RTT due to CWND reduction). The subsequent buffered packets suffer a delay of $\frac{1}{2}$ RTT due to CWND reduction. This problem is reduced by the rate-halving algorithm [Mathis et al. 1999] that paces packet transmissions when CWND is reduced. With this algorithm, the initial buffered packets are delayed by a small fraction of $\frac{1}{2}$ RTT (+ RTT due to packet dropping) when CWND is large.

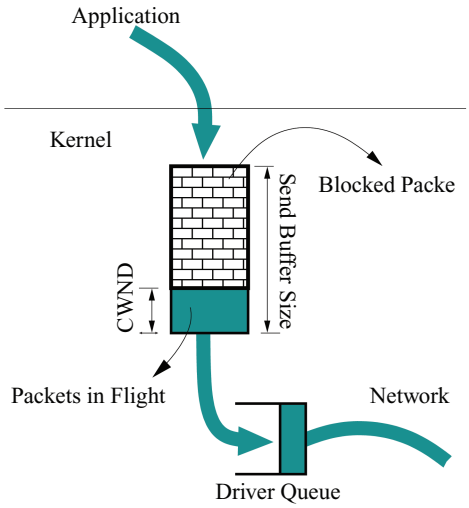
To reduce the delay associated with packet dropping, an explicit congestion notification (ECN) can be employed [Floyd and Jacobson 1993]. With ECN, routers use active queue management and explicitly inform TCP of impending congestion by setting an ECN bit on packets that would otherwise have been dropped by the router. This ECN bit is received by the receiver and then returned in an ACK packet to the sender. The TCP sender considers the ECN bit as a congestion event and reduces CWND before packets are dropped in the network due to congestion. In essence, ECN allows TCP congestion control to operate without packet dropping. Assuming few packets are dropped, TCP congestion control with ECN (together with rate-halving) introduces latencies that are a small fraction of RTT.

*Sender-Side Buffering.* The last component of protocol latency is caused by buffering on the sender side. TCP transmits application packets from a *fixed size* send buffer, as shown in Figure 1. Since TCP is a reliable protocol, it keeps copies of the CWND packets in flight in this buffer for retransmission. As described above, these packets can introduce latency (in the order of an RTT) due to packet dropping or congestion control.

Note that the TCP throughput is proportional to CWND and CWND can never exceed the send buffer size. Hence, TCP uses a large send buffer to ensure that this buffer does not restrict TCP throughput. Unfortunately, a large send buffer can introduce significant latency in a TCP stream. Consider the following example: the TCP send buffer in most current Unix kernels is at least 64 KB. For a 300 kbps (high quality) video stream, a full send buffer contributes 1700 ms of delay. This delay increases for a smaller bandwidth (low quality) stream or when the stream faces increasing competition since the stream bandwidth goes down. By comparison, the round trip time (RTT) generally lies between 50–100 ms for coast-to-coast transmission within North America. Next, we propose a technique to minimize the send buffer latency.

## 4. ADAPTIVE SEND-BUFFER TUNING

Figure 1 shows that the first CWND packets in the send buffer are in flight or have been transmitted. When an application sends a new packet into the send buffer, the packet is not transmitted until it becomes the first packet after the CWND packets in flight and an ACK arrives to open the window. We refer to such a packet as a *blocked packet* because it contributes to latency in a TCP stream, as shown in Figure 1.




TCP's send buffer stores the CWND packets that are currently being transmitted in case they are needed for retransmission. These packets generally do not introduce any latency while the blocked packets can add significant latency.

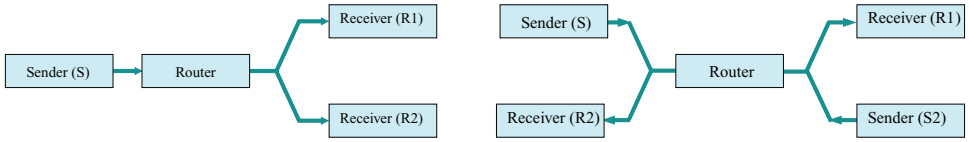
Fig. 1. TCP's send buffer.

It should be clear from Section 3 that the blocked packets can cause far more latency than packet dropping or congestion control. In particular, for a standard video stream, these packets can cause latencies that are greater than 10–20 RTT while packet dropping and congestion control cause latencies typically less than 2 RTT. Furthermore, the latter number goes down to a fraction of an RTT when rate halving and ECN (which minimizes packet dropping) are used.

The discussion above shows that large latencies are not inherent in the TCP protocol but are mainly introduced by the large send buffer in TCP implementations. Hence, we focus on reducing sender-side buffering to improve latency. To avoid storing blocked packets, the send buffer should store no more than CWND packets. Furthermore, the send buffer should not be smaller than CWND because a buffer smaller than CWND packets is guaranteed to limit throughput. In this case, CWND gets artificially limited by the buffer size rather than a congestion (or flow control) event in TCP. Hence, tuning the send buffer size to follow CWND should minimize latency without significantly affecting stream throughput. Since CWND changes dynamically over time, we call this technique adaptive send-buffer tuning and a TCP connection that uses this technique is a MIN_BUF TCP flow [Goel et al. 2002].

A MIN_BUF TCP stream blocks an application from writing data to a socket when there are CWND packets in the send buffer. Later, the application is allowed to write data to the socket when at least one new packet can be admitted in the send buffer. Consider the operation of a MIN_BUF TCP stream. The send buffer will have at most CWND packets after an application writes a packet to the socket. MIN_BUF TCP can immediately transmit this packet since this packet lies within TCP's window. After this transmission, MIN_BUF TCP will wait for the arrival of an ACK for the first packet in the current window. When the ACK arrives, TCP's window opens up by at least one packet and thus a packet can be admitted in the send buffer. Once again the application can write a packet to the send buffer, which can be transmitted immediately without introducing any latency. Hence, as long as packets are not dropped or CWND is nondecreasing, MIN_BUF TCP will minimize latency in the TCP stack.

In essence, MIN_BUF TCP moves the latency due to blocked packets to the application level. The application then has much greater control over sending time-critical data. For example, the application may decide to drop stale data or send a recently generated higher-priority packet.




(a) Forward path congestion topology (b) Reverse path congestion topology

Note that the figure on the right has congestion on both the forward and reverse paths.

Fig. 2. Network topology.

### 4.1 Evaluation

This section provides our initial results on latency improvements due to MIN_BUF streams. We evaluate the protocol latency of MIN_BUF and TCP streams under varying and heavy network load. These experiments were performed on a Linux 2.4 test-bed that simulates WAN conditions by introducing delay at an intermediate Linux router in the test-bed. To emulate a heavily loaded network environment, we run experiments with varying numbers of long-lived TCP streams, bursts of short-lived TCP streams, and a constant bit rate (CBR) stream, such as a UDP stream. All TCP flows use rate halving.

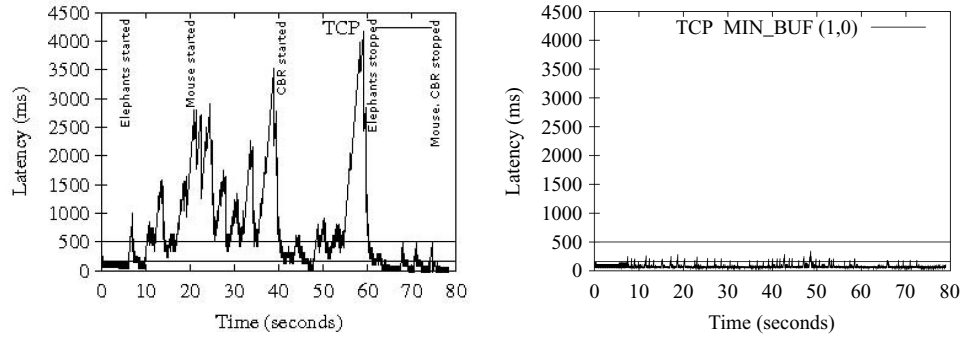
All our experiments use a single-bottleneck "dumbbell" topology and FIFO scheduling at the bottleneck. The network topology is shown in Figure 2. Figure 2(a) shows the topology used to evaluate MIN_BUF under forward path congestion, while Figure 2(b) shows the topology used when congestion occurs in the reverse path also. Each box is a separate Linux machine. The latency and throughput measurements are performed for a single stream originating at the sender *S* and terminating at the receiver *R1*. In Figure 2(a), the sender *S* generates cross traffic for both receivers *R1* and *R2*. In Figure 2(b), the sender *S* generates cross traffic for receiver *R1* and the second sender *S2* generates cross traffic along the reverse path to receiver *R2*. The router runs `nistnet` [NISTnet], a network emulation program that allows the introduction of additional delay and bandwidth constraints in the network path.

The protocol latency in all the experiments described below is measured by recording the application write time for each packet on the sender *S* and the application read time for each packet on the receiver *R1*. All the machines are synchronized to within one millisecond of each other using NTP. For the results shown below, we chose the round-trip time to be 100 ms since the median RTT between west-coast and east-coast sites in North America is approximately 100 ms [Huffaker et al. 2001]. The router queue length is chosen so that bandwidth is limited to 30 Mb/sec.

Figure 3 shows the results of a run using the forward path congestion topology. This figure compares the latency of a standard TCP and a MIN_BUF TCP[1] stream. The experiment is run for about 80 seconds with load being introduced at various different time points in the experiment. The standard TCP or MIN_BUF TCP long-lived stream being measured is started at $t = 0$ s. We refer to this stream as the *latency* stream. For evaluation, we inject heavy and varying cross traffic. At $t = 5$ s, 15 other long-lived (*elephant*) streams are started, 7 going to receiver R1 and 8 going to receiver R2. At $t = 20$ s, each receiver initiates 40 simultaneous short-lived (*mouse*) TCP streams. A mouse stream is a repeating short-lived stream that starts the connection, transfers 20 KB of data, ends the connection, and then repeats this process continuously [Iannaccone et al. 2001]. The number of mouse streams was chosen so that the mouse streams would get approximately 30 percent of the total bandwidth. At $t = 40$ s, CBR traffic that consumes 10 percent of the bandwidth is started. At $t = 60$ s, the elephants are stopped and then the mice and the CBR traffic are stopped at $t = 75$ s.

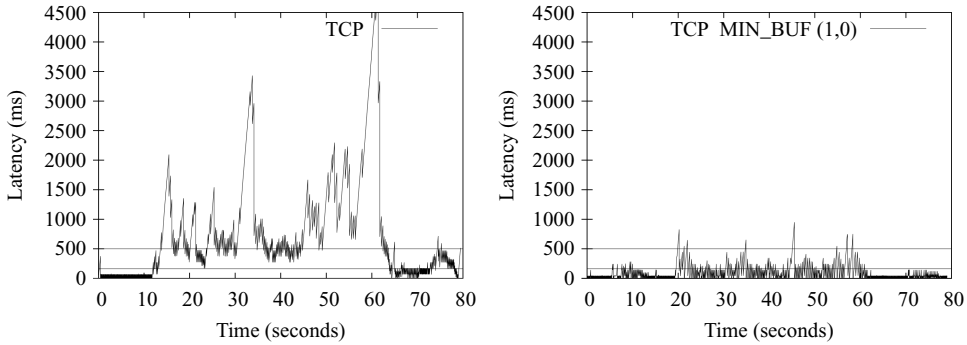
Figure 3 shows the protocol latency of *each* packet in the latency stream as a function of packet receive time. The figure shows two horizontal lines at the 160-ms and at the 500-ms thresholds. The first

[1]The significance of the (1, 0) notation after MIN_BUF will be explained in the next section. For the moment, it can be ignored.




These figures show the protocol latency of packets plotted as a function of packet receive time. For comparison, the horizontal lines on the figures show the 160 ms and 500 ms latency threshold.

Fig. 3. Protocol latencies of TCP and MIN_BUF TCP with forward path congestion topology.




These figures show the protocol latency of packets plotted as a function of packet receive time. For comparison, the horizontal lines on the figures show the 160 ms and 500 ms latency threshold.

Fig. 4. Protocol latencies of TCP and MIN_BUF TCP with reverse path congestion topology.

threshold approximates the requirements of interactive streaming applications such as video conferencing (the ITU suggests 200 ms for end-to-end latency [ITU 1993]), whereas the 500-ms threshold, while somewhat arbitrary, is chosen to represent the requirements of media-streaming control operations such as the sequence consisting of start play, fast forward, and restart play.

Figure 4 compares the protocol latency of TCP and MIN_BUF TCP when the experiment is performed using the reverse path congestion topology in which both the forward and the reverse paths are congested. The graphs in Figures 3 and 4 show that the MIN_BUF TCP stream has significantly lower protocol latency than a standard TCP stream. Reverse path congestion can cause acknowledgments to be dropped and as a result MIN_BUF TCP performs slightly worse in this topology. We conducted an extensive set of experiments with a smaller bandwidth limitation (10 Mb/sec) at the router and with smaller round-trip times (25 ms and 50 ms). As expected, the protocol latency increased with smaller available bandwidth and decreased with smaller round-trip times. However, the latency of MIN_BUF TCP flows in all these configurations is significantly lower than corresponding TCP flows and comparable to the latency shown in the two figures. We provide the latency distribution results across multiple runs later in Section 5.1, after we discuss our approach in more detail. More details about these experiments are also available in our previous paper [Goel et al. 2002].

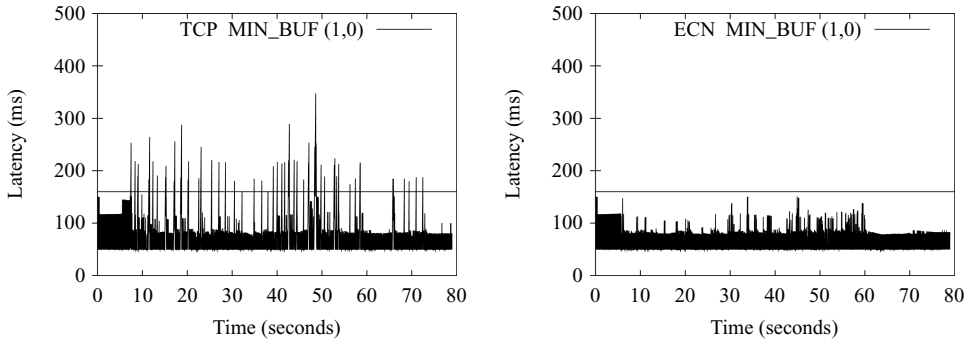



Fig. 5. Protocol latencies of MIN_BUF TCP and MIN_BUF TCP with ECN.

We also ran the same experiment with MIN_BUF TCP enabled with ECN. For ECN, we use derivative random drop (DRD) active queue management, which is supported in `nistnet`. DRD is a RED variant that is implemented efficiently in software. The *drdmin*, *drdmax* and *drdcongest* parameters of DRD were chosen to be 1.0, 2.0 and 2.0 times the bandwidth-delay product, respectively. DRD marks 10-percent packets with the ECN bit when the queue length exceeds *drdmin,* progressively increasing the percentage until packets are dropped when the queue length exceeds *drdcongest*. Unlike RED, DRD does not average queue lengths. Figure 5 shows the protocol latency for MIN_BUF TCP and MIN_BUF TCP with ECN enabled. The left graph in Figure 5 zooms into the right graph of Figure 3. The right graph in Figure 5 shows that MIN_BUF TCP with ECN has lower latency than without ECN. Each of the spikes in the left graph of Figure 5 is a packet dropping event (note that the network is heavily loaded) that causes an additional round-trip time delay (100 ms in these experiments) that MIN_BUF TCP with ECN does not suffer. The smaller spikes seen with ECN occur because congestion control decreases CWND. However, as explained in Section 3, these congestion control spikes are small because they occur without packet dropping. Note that TCP with ECN but without MIN_BUF drops few packets but still suffers large delays because of blocked packets.

### 4.2 Implications for Adaptive Applications

MIN_BUF TCP removes sender-side buffering latency from the TCP stack so that low-latency applications are allowed to handle buffering themselves. This approach allows applications to adapt their bandwidth requirements to maintain low latency through data scalability techniques such as frame dropping, priority data dropping and dynamic rate shaping [Rejaie et al. 1999; Feng et al. 1999; Krasic et al. 2003]. More precisely, the benefit of MIN_BUF TCP streaming is that the sending side application can wait longer before making its adaptation decisions. Hence, it has more control and flexibility over what data should be sent and when it should be sent. For instance, if MIN_BUF TCP doesn't allow the application to send data for a long time, the sending side can drop low-priority data. Then it can send higher-priority data, which will arrive at the receiver with low delay (instead of committing the low-priority data to a large TCP send-buffer early and then losing control over quality adaptation and timing when that data is delayed in the send buffer). In our model, applications are typically written using nonblocking write socket calls so that the sending side can do other work such as media encoding and make adaptation decisions when the network is busy.

Note that low-latency applications that do not adapt their sending behavior or are simply bandwidth limited but do not have any any latency requirements (e.g., ftp) will not benefit from MIN_BUF TCP.

## 5. EFFECT ON THROUGHPUT

Until now, we have compared the protocol latency of MIN_BUF TCP and standard TCP flows. However, another important issue for MIN_BUF TCP streams is the throughput of these streams compared with standard TCP. The size of the send buffer in standard TCP is large because it helps TCP throughput. The MIN_BUF approach will impact network throughput when standard TCP could have sent a packet but there are no new packets in MIN_BUF TCP's send buffer. This condition occurs because when an ACK arrives, standard TCP has a packet in the send buffer that it can send immediately while MIN_BUF TCP has to wait for the application to write the next packet before it can send it. With this loop, system latencies such as preemption and scheduling latency can affect MIN_BUF TCP throughput.

These adverse affects on MIN_BUF TCP throughput can be reduced by adjusting the buffer size so that it is slightly larger than CWND. To understand how much the buffer size should be increased, we need to consider events in TCP which cause new packets to be transmitted. There are four such events: ACK arrival, delayed ACK arrival, CWND increase and back-to-back ACK arrivals due to ACK compression [Shenker et al. 1991]. These events and the way MIN_BUF TCP can handle them are described below.

*ACK Arrival.* When an ACK arrives for the first packet in TCP window, the window moves forward and admits a new packet in the window, which is then transmitted by TCP. In this case, the MIN_BUF TCP send buffer should buffer one additional packet so that it can immediately send this packet.

*Delayed ACK.* To save bandwidth in the reverse direction, most TCP implementations delay ACKs and send, by default, one ACK for every two data packets received. Hence, each ACK arrival opens TCP's window by two packets. To handle this case, MIN_BUF TCP should buffer two instead of one additional packets.

*CWND Increase.* During steady state, when TCP is in its additive increase phase, TCP probes for additional bandwidth by increasing its window by one packet every round-trip time. Hence, TCP increments CWND by 1. At these times, the ACK arrival allows releasing two packets. With delayed ACKs, three packets can be released. Hence, MIN_BUF TCP should buffer three additional packets to deal with this case and delayed ACKs.

TCP can also use a byte-counting algorithm [Allman 2003] in which the congestion window is increased based on the number of bytes acknowledged by the arriving ACKs. The byte-counting algorithm improves performance by mitigating the impact of delayed ACKs on the growth of CWND. A byte counting TCP behaves similar to regular TCP unless its L limit parameter is chosen to be more aggresive than the default one MSS value [Allman 2003]. For instance, byte counting TCP effectively increments CWND by one MSS sized packet every RTT. Hence the discussion in this section applies to byte counting TCP also.

*ACK Compression.* At any time CWND TCP packets are in transit in the network. Due to a phenomenon known as ACK compression [Shenker et al. 1991] that can occur at routers, ACKs can arrive at the sender in a bursty manner. In the worst case, the ACKs for all the CWND packets can arrive together. To handle this case, MIN_BUF TCP should buffer 2 * CWND packets (CWND packets in addition to the first CWND packets). Note that the default send buffer size can often be much larger than 2 * CWND and thus we expect lower protocol latency even in this case. If we take CWND increase into account with ACK compression then MIN_BUF TCP should allow buffering 2 * CWND + 1 packets to achieve throughput comparable to TCP.

*Dropped ACK.* When the reverse path is congested, ack packets can be dropped. Then later acks can acknowledge more than two packets. This case is similar to ACK compression.

To study the impact of the send-buffer size on throughput and latency, we add two parameters A > 0 and B ≥ 0 to MIN_BUF TCP streams. With these parameters, the send buffer is limited to A * CWND + B packets at any given time.[2] We use the A * CWND + B function because increasing the first parameter (A) allows handling any bandwidth reduction that can be caused by ACK compression, as described above, while progressively increasing the second parameter (B) allows taking ACK arrivals, delayed ACKs and CWND increase into account. Furthermore, when A ≥ 2 and B ≥ 1, the timing with which TCP sends and acknowledges packets is unaffected by our approach, and as a result the throughput achieved by MIN_BUF TCP should be comparable to TCP.

The parameters A and B represent a trade-off between latency and throughput. In general, we expect that for every additional CWND blocked packets, output latency will increase by a network round-trip time since a packet must wait for an additional CWND ACKs before being transmitted. From now on, we call a MIN_BUF TCP stream with parameters A and B, a MIN_BUF(A, B) stream. Hence, the original MIN_BUF TCP stream which limited the send-buffer size to CWND packets is a MIN_BUF(1, 0) stream.

### 5.1 Evaluation

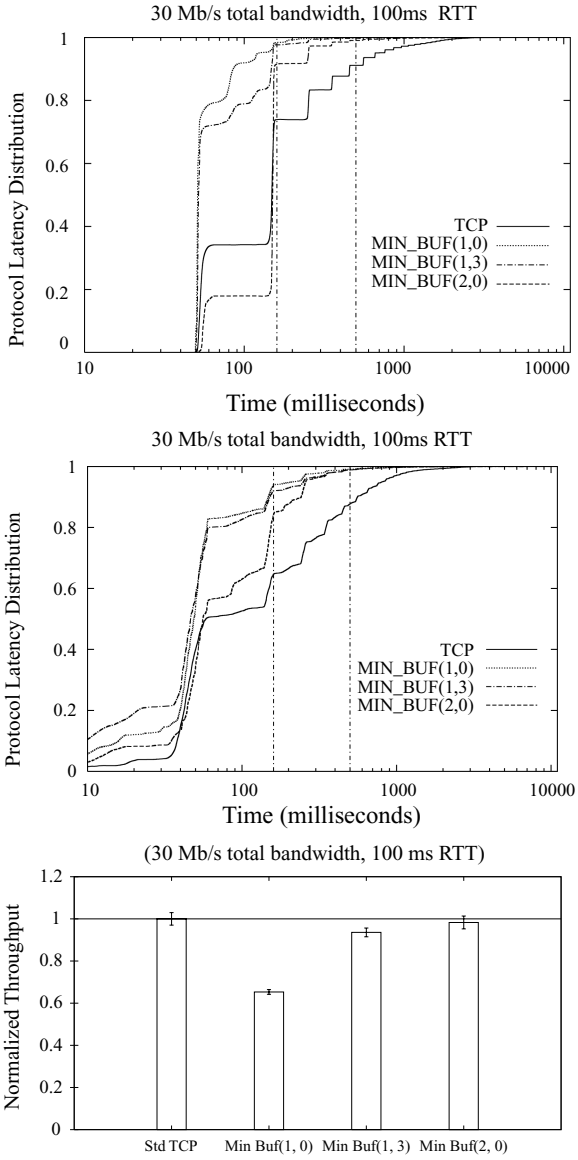
Figure 6 shows the protocol latency and throughput of TCP and three MIN_BUF configurations: MIN_BUF(1, 0), MIN_BUF(1, 3) and MIN_BUF(2, 0). The MIN_BUF(1, 0) stream is the original stream with the least protocol latency. We chose a MIN_BUF(1, 3) stream to take ACK arrivals, delayed ACKs and CWND increase into account. Finally, we chose a MIN_BUF(2, 0) stream because it takes ACK compression and dropped ACKs into account and we expect it to have throughput close to TCP, as explained earlier.

The top two graphs in Figure 6 show the protocol latency distribution when the experiments are performed using the forward and the reverse path congestion topologies. This distribution is the percentage of packets that arrive at the receiver within a delay threshold. The vertical lines show the 160 ms and 500 ms delay thresholds. The first threshold represents the requirements of interactive applications such as video conferencing while the second threshold represents the requirements of media control operations. Each experiment was performed eight times and the results show the latencies accumulated over all the runs. The figures show that TCP has a much longer tail than any of the MIN_BUF TCP flows. Note that the x-axis, which shows the protocol latency in milliseconds, is on a log scale. In the forward path topology, the MIN_BUF(1, 0) and MIN_BUF(1, 3) streams have less than 2% packets outside the 160 ms threshold while MIN_BUF(2, 0) and TCP streams have about 10% and 30% packets outside this threshold. With reverse path congestion, acknowledgments can be dropped, leading to slightly increased delays. However, MIN_BUF(1, 0) and MIN_BUF(1, 3) streams still drop less than 10% packets within the 160 ms threshold while TCP drops over 40% packets.

We conducted an extensive set of experiments for the four MIN_BUF configurations and with a smaller bandwidth limitation (10 Mb/sec) at the router and with smaller round-trip times (25 ms and 50 ms). The results of these experiments are similar to the latency results shown in Figure 6 and hence not presented here. More details about these experiments are available in our previous paper [Goel et al. 2002].

The normalized throughput of these flows is shown in the bottom graph of Figure 6. The numbers shown are the mean and 95% confidence interval over 8 runs. The graph shows that, as expected, the MIN_BUF(2, 0) flow receives throughput close to standard TCP (within the confidence intervals). The MIN_BUF(1, 0) flow receives the least throughput because TCP has no new packets in the send buffer

[2]With the byte counting algorithm [Allman 2003], the send buffer is limited to (A * CWND + B) * (maximum segment size (or MSS)) bytes.




The experiments were performed with a 30 Mb/sec limit and with 100 ms round-trip delay. The top two graphs shows the protocol latency distributions for the forward and the reverse path congestion topologies. The bottom graph shows the normalized throughput of the different MIN_BUF TCP configurations.

Fig. 6. Protocol latency distribution and throughput of TCP in three MIN_BUF TCP configurations.

that can be sent after each ACK is received and hence TCP must ask the application to write the next packet to the send buffer before it can proceed with the next transmission. The MIN_BUF(1, 3) flow receives about 95 percent of TCP throughput. The three blocked packets in the send buffer are able to handle delayed ACKs and CWND increase. These graphs show that the MIN_BUF(1, 3) flow presents a good compromise between achieving low latency and good throughput. Experiments with a lower

Table I. Major CPU Costs of TCP and MIN_BUF TCP

| | TCP | MIN_BUF(1,0) |
|---|---|---|
| write | 2.33 seconds | 2.67 seconds |
| poll | 0.45 seconds | 3.55 seconds |
| total CPU time | 4.20 seconds | 11.50 seconds |

bandwidth limitation of 10 Mb/sec and shorter RTT of 50 and 25 ms show similar results and hence are not presented here.

### 5.2 System Overhead

The MIN_BUF TCP approach reduces protocol latency compared to TCP flows by allowing applications to write data to the kernel at a fine granularity. However, this approach can cause higher system overhead because more system calls are invoked to transfer the same amount of data. To quantify this overhead, we profiled the CPU usage of TCP and MIN_BUF(1,0) flows for the experiment shown in Figure 3. Profiling the event-driven application showed that the two main costs on the sender side were the `write` and the `poll` system calls for both standard TCP and MIN_BUF TCP flows. Table I shows the total time spent in the kernel (system time) in `write` and `poll` for these flows and the total CPU time (user and system time).

Table I shows that the MIN_BUF TCP flow has slightly more overhead for write calls. MIN_BUF TCP writes one packet at a time to the socket while standard TCP writes several packets at a time before the application is allowed to write next time. In particular, TCP in Linux allows the application to write only after a third of the send buffer has been drained. Hence, standard TCP amortizes context switching overhead over larger writes.

Table I also shows that MIN_BUF TCP has significantly more overhead for poll calls. With standard TCP, poll is called after the application writes a third of the send buffer because, as explained above, TCP wakes the application only after draining a third of the buffer. With MIN_BUF TCP, poll is called after every write. The default send buffer size in TCP is 64 Kb. A third of that size is 21 Kb, which allows 14 packets of MSS size (1448 bytes). Hence, in our application, with standard TCP, poll should be called after every 14 writes, while with MIN_BUF(1,0) it is called every time. We measured the number of calls to poll over the entire experiment for TCP and MIN_BUF(1,0) flows and found the ratio of these numbers is 12.66, which is close to the expected value of 14. The slight discrepancy occurs because when TCP increases CWND, then MIN_BUF TCP can send two packets in a single write, hence the ratio of writes is slightly less than 14.

The last row of the Table I shows that the MIN_BUF(1, 0) TCP flows are approximately three times more expensive in CPU time compared to standard TCP flows. This overhead occurs as a result of fine-grained writes that are allowed by MIN_BUF TCP flows. These fine-grained writes occur as a result of low-latency streaming but their benefit is that applications have much finer control over latency. To reduce the overhead of MIN_BUF(1, 0), larger values of the MIN_BUF parameters can be used. The larger values will help amortize the poll and write overheads at the cost of increased latency.

## 6. IMPLEMENTATION

We have implemented the MIN_BUF TCP approach with a small modification to the TCP stack on the sender side in the Linux 2.4 kernel [Goel et al. 2002]. This modification can be enabled per socket by using a new SO_TCP_MIN_BUF option, which limits the send buffer size to $A * \text{CWND} + \text{MIN}(B, \text{CWND})$ segments (segments are packets of maximum segment size or MSS) at any given time. The send buffer size is at least CWND because $A$ must be an integer greater than zero and $B$ is zero or larger.

With the send-buffer modification, an application is blocked from sending when there are $A * \text{CWND} + \text{MIN}(B, \text{CWND})$ packets in the send buffer. In addition, the application is woken up when at least one packet can be admitted in the send buffer. By default $A$ is one and $B$ is zero, but these values can be made larger with the SO_TCP_MIN_BUF option.

### 6.1 Sack Correction

The previous discussion about the send buffer limit applies for a non-SACK TCP implementation. For TCP SACK [Mathis et al. 1996], we make a *sack correction* by adding an additional term *sacked_out* to $A * \text{CWND} + \text{MIN}(B, \text{CWND})$. The sacked_out term (or an equivalent term in other OSs) is maintained by a TCP SACK sender and is the number of selectively acknowledged packets. With TCP SACK, when selective acknowledgments arrive, the packets in flight are no longer contiguous but lie within a CWND+sacked_out packet window. We make the sack correction to ensure that the send buffer limit includes this window and is thus at least CWND+sacked_out. Without this correction, TCP SACK is unable to send new packets for a MIN_BUF flow and assumes that the flow is application limited. It can thus reduce the congestion window multiple times after the arrival of selective acknowledgments.

### 6.2 Alternate Application-Level Implementation

It is conceivable that the objectives of the send-buffer modifications can be achieved at the application level. Essentially the application would stop writing data when the socket buffer has a fill level of $A * \text{CWND} + \text{MIN}(B, \text{CWND})$ packets or more. The problem with this approach is that the application has to poll the socket fill level. Polling is potentially both expensive in terms of CPU consumption and inaccurate since the application is not informed immediately when the socket-fill level goes below the threshold.

### 6.3 Application Model

Applications that use MIN_BUF TCP should explicitly align their data with packets transmitted on the wire (application level framing) [Clark and Tennenhouse 1990]. This alignment has two benefits: (1) it minimizes any latency due to coalescing or fragmenting of packets below the application layer and (2) it ensures that low-latency applications are aware of the latency cost and throughput overhead of coalescing or fragmenting application data into network packets. For alignment, an application should write maximum segment size (MSS) packets on each write. TCP determines MSS during stream startup but the MSS value can change due to various network conditions such as routing changes [McCann et al. 1996]. A latency-sensitive application should be informed when TCP determines that the MSS has changed. Currently, we detect MSS changes at the application level by querying TCP for the MSS before each write. Another more efficient option, currently unimplemented, would be to return a write error on an MSS change for a MIN_BUF socket.

In our experiments, we use the TCP_CORK socket option in Linux, which ensures that TCP sends data only when there are at least MSS bytes of data available. Thus, application data is never broken up by TCP into less than MSS sized packets. This option improves throughput by always sending maximum sized packets but does not affect protocol latency since applications have already aligned their data with MSS sized packets.

If applications such as audio streaming have packets smaller than MSS bytes, then the CORK option should not be used since it introduces additional delay. Note that when packets smaller than MSS bytes are transmitted, the stream has lower throughput than a stream using the TCP_CORK socket option, but the latency improvements due to MIN_BUF are unaffected because MIN_BUF simply minimizes buffering in the send buffer. Finally, if the network is not saturated (e.g., when using a single stream

of audio with no competing traffic), then the send buffer will not fill up and TCP and MIN_BUF flows will exhibit similar latencies.

## 7. APPLICATION-LEVEL EVALUATION

The previous section evaluated the protocol latency due to TCP under heavy network load. This section evaluates the timing behavior of a *real* live streaming application and shows how MIN_BUF TCP helps in improving application-level end-to-end latency. In particular, our experimental results will show that MIN_BUF TCP improves the end-to-end latency distribution of video frames and hence the number of frames that arrive within a given deadline. In addition, we show that MIN_BUF TCP has lower variance in throughput compared to TCP, which allows streaming video with smoother quality.

We ran these experiments over TCP and MIN_BUF TCP on a best-effort network that does not guarantee bandwidth availability. With a nonadaptive media application, data will be delayed, possibly for long periods of time, when the available bandwidth is below the application's bandwidth requirements. Hence, we need to use an *adaptive* media application that can adapt its bandwidth requirements based on currently available bandwidth. For this purpose, we have chosen an open-source adaptive streaming application called Qstream [Krasic et al. 2003].

We need to understand the *adaptive media format* and the *adaptation mechanism* used in Qstream to analyze the components of application-level end-to-end latency. Qstream uses an adaptive media format called scalable MPEG (SPEG). SPEG [Krasic et al. 2003] is a variant of MPEG-1 that supports layered encoding of video data that allows dynamic data dropping. In a layered encoded stream, data is conceptually divided into layers. A base layer can be decoded into a presentation at the lowest level of quality. Extension layers are stacked above the base layer where each layer corresponds to a higher level of quality in the decoded data. For correct decoding, an extension layer requires all the lower layers.

Qstream uses an adaptation mechanism called *priority-progress streaming* (PPS). For the purposes of this work, the key idea in the PPS adaptation mechanism is an *adaptation period*, which determines how often the sender drops data. Within each adaptation period, the sender sends data packets in priority order, from the highest priority to the lowest priority. The priority label on a packet exposes the layered nature of the SPEG media so that higher layers can be sent and the display quality improved with increases in resource availability. Hence, the highest priority data has the base quality while lower priority data encodes higher quality layers. The data within an adaptation period is called an adaptation window. Data dropping is performed at the end of the adaptation period where all unsent data from the adaptation window is *dropped* and the server starts processing data for the next adaptation period. Consequently, the data that is transmitted within an adaptation period determines the quality of the presentation for that adaptation period. On the receiver side, data packets are collected in priority order for each adaptation period and then reordered in time order before they are displayed.[3] Note that Qstream displays data in real-time even when it drops lower priority data. If the available bandwidth is low or the sender side is blocked from sending data for long periods of time, then the sender drops entire adaptation windows, which we call *dropped windows*.

The minimum expected latency at the application level at the sender and the receiver sides is a function of the adaptation period. In particular, the sender must wait for an adaptation period to perform data prioritization before it can starting sending the adaptation window. Similarly, the receiver must wait for an adaptation period to receive and reorder an adaptation window. Figure 7 shows all the components of the end-to-end latency experienced in Qstream. Note that capture, protocol and display latencies occur in the kernel while the rest of the latencies occur at the application level. In our

[3]PPS uses time-stamps on data packets to perform the reordering.

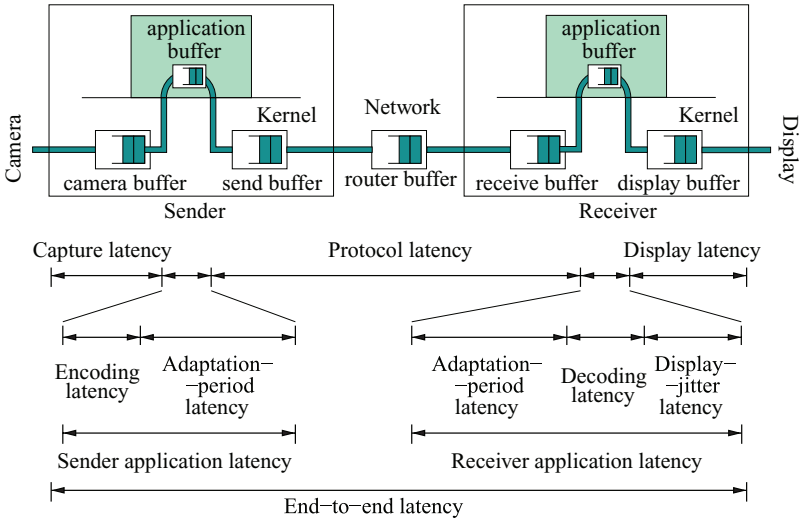



Fig. 7. Breakup of end-to-end latency in Qstream.

experience, capture and display latencies are relatively small and hence our focus on protocol latency helps to significantly reduce kernel latencies.

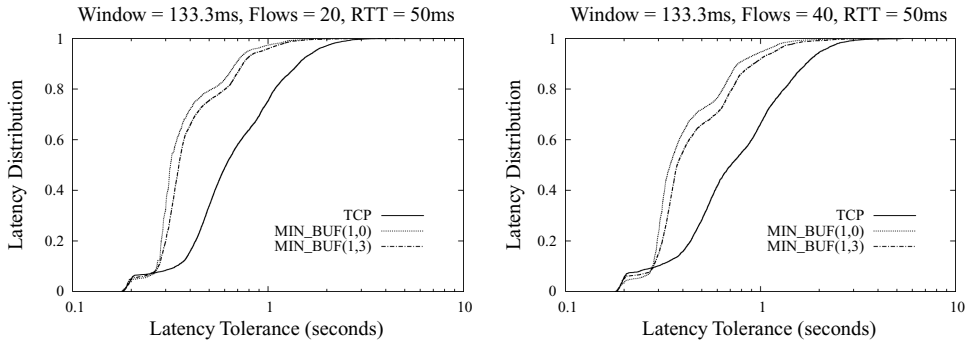
### 7.1 Evaluation Methodology

We use three metrics *latency distribution, sender throughput* and *dropped windows* to evaluate the performance of the Qstream application running under MIN_BUF TCP versus TCP. The latency distribution is a measure of the end-to-end latency, shown in Figure 7, experienced by the adaptation windows during an experimental run. It measures the ratio of the number of *entire* windows that arrive within an application's end-to-end latency requirements (henceforth called *latency tolerance*) to the number of windows transmitted by the sender. Note that Qstream displays entire or partial windows that arrive within the latency tolerance and hence this metric is a conservative estimate of goodput (i.e., throughput that is useful) especially when the windows are large. The sender throughput is the amount of data that is transmitted by the sender. Finally, dropped windows is the number of *entire* adaptation windows that are dropped by the *sender*. Qstream drops an entire adaptation window when the first packet of this window arrives after the latency tolerance and hence none of the packets of the window can be displayed in time. Note that the number of windows dropped by the sender across runs can be different even if the sender throughput is the same because the amount of data sent in each window (and hence the quality of a window) is variable. For the same value of sender throughput, a smaller value of dropped window indicates that more windows were transmitted and the stream had smoother quality.

The experiments are performed on our Linux 2.4 test-bed that simulates WAN conditions by introducing a known delay at an intermediate Linux router in the test-bed. Experiments are run under varying network load and the cross-traffic and the network topology is similar to the traffic and the topology described in Section 4.1.

### 7.2 Results

First, we compare the latency distribution of flows using TCP versus MIN-BUF TCP. For these experiments, the round-trip time is 50 ms and the bandwidth capacity of the link is 10 Mb/s. Figure 8




The bandwidth limit for all the experiments is 10 Mbps, the round trip time is 50 ms. Note that for latency tolerance of less than a second, the MIN_BUF flows have significantly higher number of windows that arrive in time compared to TCP.

Fig. 8. Latency distribution (adaptation window = 133 ms).

Table II. Throughput of TCP and MIN_BUF TCP

| | 20 flows | 40 flows |
|---|---|---|
| TCP | $0.76 \pm 0.11$ | $0.56 \pm 0.04$ |
| MIN_BUF(1,0) | $0.60 \pm 0.05$ | $0.47 \pm 0.04$ |
| MIN_BUF(1,3) | $0.72 \pm 0.04$ | $0.54 \pm 0.03$ |

The throughput of the MIN_BUF(1,0) and the MIN_BUF(1,3) flows is approximately 80% and 95% of TCP. The experiments are run with 20 and 40 competing flows.

compares the results for TCP, MIN_BUF(1, 0) and MIN_BUF(1, 3) flows. It shows the latency distribution of each of the flows when the adaptation window period is 4 frames or 133.3 ms (camera captures data at 30-frames a second). The figures show that as long as the application's latency tolerance is less than a second, significantly more windows arrive in time for MIN_BUF flows compared to a TCP flow. For example, 80% windows arrive in less than 500 ms for MIN_BUF flows while only 40% windows arrive within the same time for a TCP flow when there are 20 competing flows.

We conducted experiments with different competing loads. The two graphs in Figure 8 show the results of experiments with 20 and 40 competing long-lived flows. These figures show that with increasing load, the percent of transmitted packets that arrive in time is only marginally affected for MIN_BUF flows (clearly the total amount of throughput achieved by the video flow is lower in the second case, as shown in Table II later).

Note that Figure 8 shows the end-to-end latency that includes kernel as well as application-level latencies and in this experiment, the Qstream application itself introduces a latency of 2 times the adaptation window or 266 ms. To reduce this latency, we ran the experiments above with a smaller adaptation window of 2 frames (or 66.6 ms). Figure 9 shows the latency distribution in this case. It shows that with MIN_BUF, 80% of packets arrive within 300–400 ms when there are 20 competing flows. This figure shows that the latency tolerance can be made tighter when the adaptation window is made smaller for any desired percent of timely window arrivals. The trade-off, as discussed below, is that the video quality varies more as the window is made smaller. We performed experiments with other values of the round-trip time and the bandwidth capacity but those results are similar and hence not shown here.

Next, we compare the sender throughput of TCP and MIN_BUF flows. We show one set of results (averaged over 5 runs) when the adaptation window is 2 frames (66 ms) since the size of the adaptation window does not affect the sender throughput much. Table II shows that the throughput achieved by

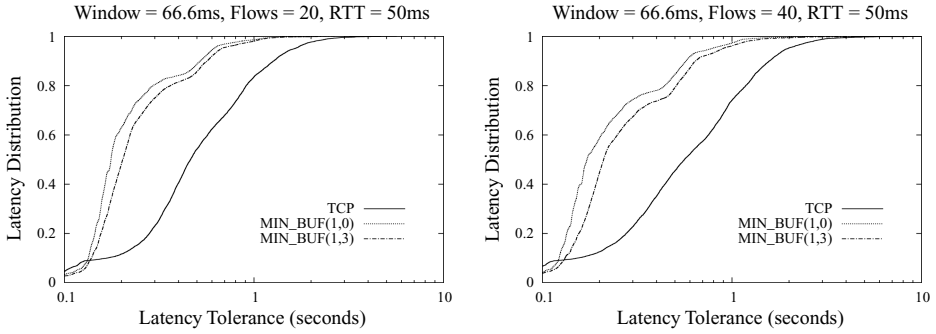



Fig. 9. Latency distribution (adaptation window = 66 ms).

Table III. Windows Dropped by TCP and MIN_BUF TCP

| | 4 frames | 2 frames |
|---|---|---|
| TCP | $45 \pm 3\%$ | $58 \pm 5\%$ |
| MIN_BUF(1,0) | $28 \pm 2\%$ | $37 \pm 2\%$ |
| MIN_BUF(1,3) | $27 \pm 2\%$ | $34 \pm 3\%$ |

The adaptation window is 4 frames and 2 frames. MIN_BUF drops fewer entire adaptation windows and thus provides smoother video quality. Reducing the adaptation window reduces the smoothness.

MIN_BUF(1,0) is close to 80% of TCP while the throughput achieved by MIN_BUF(1,3) is close to 95% of TCP with 20 or 40 competing flows. These numbers are close to the relative throughput achieved by MIN_BUF flows in the micro-benchmarks presented in Figure 6 and confirm that MIN_BUF(1,3) flows compete closely with TCP flows while significantly improving end-to-end latency.

The last metric we compare for TCP and MIN_BUF flows is the number of dropped windows over the course of the experiment, which provides a rough estimate of the variation in video quality over time and hence the responsiveness of adaptation. Table III shows the ratio of adaptation windows dropped at the *sender* to the number of adaptation windows generated at the server, expressed as a percentage, with 20 competing flows. The sender drops windows when it is unable to transmit data for an entire window period. This table shows that MIN_BUF flows drop fewer entire windows because these flows can send data in a less bursty manner, which results in smoother video quality since more frames are transmitted. Table III also shows that when the adaptation window period is small, then more windows are dropped as a percent of total number of windows because an idle transmission period is more likely to lead to a smaller window being dropped. Since larger numbers of windows are dropped with a smaller adaptation window, the video quality fluctuates more over time.

### 7.3 Discussion

The adaptation period allows control over the delay and the quality stability trade-off. A large adaptation period reduces the frequency of quality fluctuations because there are at most two quality changes in each adaptation period within Qstream and the window is more likely to be at least partially transmitted. However, a large adaptation period increases end-to-end latency as shown in the graphs above. For a low-latency streaming application, the user should have control over Qstream's adaptation period so that the desired latency and quality trade-off can be achieved.

## 8. CONCLUSIONS

The dominance of the TCP protocol on the Internet and its success in maintaining Internet stability has led to several TCP-based stored media-streaming approaches. The success of TCP-based streaming led us to explore the limits to which TCP can be used for low-latency network streaming. Low latency streaming allows building adaptive streaming applications that respond to changing bandwidth availability quickly and sufficiently low-latency streaming makes interactive applications feasible.

This paper shows that low-latency streaming over TCP is feasible by tuning TCP's send buffer so that it keeps just the packets that are currently in flight. Compared to packet dropping, TCP retransmissions or TCP congestion control, this approach has the most significant effect on TCP induced latency at the application level. In addition, we showed that a few extra packets (blocked packets) help to recover much of the lost network throughput without increasing protocol latency significantly. Our adaptive buffer tuning approach can be used by any application that prioritizes data. To validate our approach, we evaluated the performance of an adaptive streaming application that prioritizes data based on layered media encoding. Our results show that TCP buffer tuning yields significant benefits in terms of reducing end-to-end latency and variation in media quality.

### ACKNOWLEDGMENTS

We greatly appreciate the detailed feedback from our anonymous reviewers. The ideas in this paper were refined during several discussions with Kang Li, Wu-chang Feng and Wu-chi Feng. We wish to thank the members of the Quasar group at OGI, Portland who provided comments on initial drafts of the paper.

### REFERENCES

ALLMAN, M. 2003. TCP congestion control with appropriate byte counting (ABC). Internet RFC 3465.

ALLMAN, M., PAXSON, V., AND STEVENS, W. 1999. TCP congestion control. Internet RFC 2581.

BANSAL, D., BALAKRISHNAN, H., FLOYD, S., AND SHENKER, S. 2001. Dynamic behavior of slowly-responsive congestion control algorithms. In *Proceedings of the ACM SIGCOMM*. ACM, New York.

CLARK, D. D., AND TENNENHOUSE, D. L. 1990. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM*. ACM, New York. 200–208.

FENG, W., KANDLUR, D. D., SAHA, D., AND SHIN, K. S. 1997. Techniques for eliminating packet loss in congested TCP/IP networks. Tech. Rep. CSE-TR-349-97, Univ. Michigan. Nov.

FENG, W., LIU, M., KRISHNASWAMI, B., AND PRABHUDEV, A. 1999. A priority-based technique for the best-effort delivery of stored video. In *Proceedings of the SPIE Multimedia Computing and Networking Conference*. 286–300.

FLOYD, S., HANDLEY, M., AND KOHLER, E. 2002. Problem statement for DCP. Work in progress, IETF Internet Draft draft-floyd-dcp-problem-00.txt, expires Aug 2002.

FLOYD, S., HANDLEY, M., PADHYE, J., AND WIDMER, J. 2000. Equation-based congestion control for unicast applications. In *Proceedings of the ACM SIGCOMM*. ACM, New York. 43–56.

FLOYD, S. AND JACOBSON, V. 1993. Random early detection gateways for congestion avoidance. *ACM/IEEE Trans. Netw. 1*, 4 (Aug.), 397–413.

GOEL, A., KRASIC, C., LI, K., AND WALPOLE, J. 2002. Supporting low latency TCP-based media streams. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*. 193–203.

HUFFAKER, B., FOMENKOV, M., MOORE, D., AND CLAFFY, K. C. 2001. Macroscopic analyses of the infrastructure: Measurement and visualization of internet connectivity and performance. In *Proceedings of the workshop on Passive and Active Measurements (PAM2001)*.

HURLEY, P. AND LE BOUDEC, J. Y. 1999. A proposal for an asymmetric best-effort service. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*. 132–134.

IANNACCONE, G., MAY, M., AND DIOT, C. 2001. Aggregate traffic performance with active queue management and drop from tail. *ACM Comput. Commun. Rev. 31*, 3 (July), 4–13.

ITU. 1993. *Transmission Systems and Media, General Recommendation on the Transmission Quality for an Entire International Telephone Connection; One-Way Transmission Time*. Geneva, Switzerland. Recommendation G.114, Telecommunication Standardization Sector of ITU.

KOHLER, E., HANDLEY, M., AND FLOYD, S. 2006. Datagram congestion control protocol (DCCP). Internet RFC 4340.

KOZUCH, M. AND SATYANARAYANAN, M. 2002. Internet Suspend/Resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*. 40–48.

KRASIC, C., WALPOLE, J., AND FENG, W. 2003. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*. 112–121.

MATHIS, M., MAHDAVI, J., FLOYD, S., AND ROMANOW, A. 1996. TCP selective acknowledgment options. Internet RFC 2018.

MATHIS, M., SEMKE, J., MAHDAVI, J., AND LAHEY, K. 1999. Rate-halving algorithm for TCP congestion control. `http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt`.

MCCANN, J., DEERING, S., AND MOGUL, J. 1996. Path MTU discovery for IP version 6. Internet RFC 1981.

NETMEETING. Windows Netmeeting. `http://www.microsoft.com/netmeeting`, viewed in Jun 2002.

NISTNET. The NIST network emulation tool. `http://www.antd.nist.gov/itg/nistnet`, viewed in Jun 2002.

NONNENMACHER, J., BIERSACK, E. W., AND TOWSLEY, D. 1998. Parity-based loss recovery for reliable multicast transmission. *ACM/IEEE Trans. Netw. 6*, 4, 349–361.

RAMAKRISHNAN, K., FLOYD, S., AND BLACK, D. 2001. The addition of explicit congestion notification (ECN) to IP. Internet RFC 3168.

REALVNC LIMITED. 2002. Realvnc. `http://www.realvnc.com`.

REJAIE, R., HANDLEY, M., AND ESTRIN, D. 1999. Quality adaptation for congestion controlled video playback over the internet. In *Proceedings of the ACM SIGCOMM*. ACM, New York. 189–200.

RIZZO, L. 1997. Effective erasure codes for reliable computer communication protocols. *ACM Comput. Commun. Rev. 27*, 2 (Apr.), 24–36.

SALIM, J. H. AND ALMED, U. 2000. Performance evaluation of explicit congestion notification (ECN) in IP networks. Internet RFC 2884.

SEMKE, J., MAHDAVI, J., AND MATHIS, M. 1998. Automatic TCP buffer tuning. In *Proceedings of the ACM SIGCOMM*. ACM, New York. 315–323.

SHENKER, S., ZHANG, L., AND CLARK, D. 1991. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of the ACM SIGCOMM*. ACM, New York. 133–147.

SKYPE. Skype. `http://www.skype.com/`.

STEWART, R., XIE, Q., MORNEAULT, K., SHARP, C., SCHWARZBAUER, H., TAYLOR, T., RYTINA, I., KALLA, M., ZHANG, L., AND PAXSON, V. 2000. Stream control transmission protocol. Internet RFC 2960.

YANG, Y. R., AND LAM, S. S. 2000. General aimd congestion control. Tech. Rep. TR-2000-09, University of Texas at Austin. Austin, TX, May.

Received June 2005; revised November 2005, August 2006, August 2007; accepted October 2007