5-2002

Supporting Low-Latency TCP-Based Media Streams

Ashvin Goel
Oregon Graduate Institute of Science & Technology

Charles Krasic
Oregon Graduate Institute of Science & Technology

Kang Li
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Citation Details

Goel, Ashvin, Charles Krasic, Kang Li, and Jonathan Walpole. "Supporting low latency TCP-based media streams." In Quality of Service, 2002. Tenth IEEE International Workshop on, pp. 193-203. IEEE, 2002.

Supporting Low Latency TCP-Based Media Streams

Ashvin Goel Charles Krasic Kang Li Jonathan Walpole
Oregon Graduate Institute, Portland
{ashvin,krasic,kangli,walpole}@cse.ogi.edu

***Abstract*— 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 these approaches raises the question whether TCP can be used for low-latency streaming. Low latency streaming allows responsive control operations for media streaming and can make interactive applications feasible. We examined adapting the TCP send buffer size based on TCP's congestion window to reduce application perceived network latency. Our results show that this simple idea significantly improves the number of packets that can be delivered within 200 ms and 500 ms thresholds.**

I. Introduction

Traditionally, the multimedia community has considered TCP unsuitable for streaming audio and video data. The main issues raised against TCP-based streaming have been related to *congestion control* and *packet retransmissions*. TCP congestion control is designed to probe available bandwidth through deliberate manipulation of the transmission rate. This rate variation can impede effective streaming because the streaming requirements are not necessarily matched with the transmission rate, causing either data dropping or accumulation of buffered data and thus delay. In addition, congestion control can lead to sustained or long-term reduction in rate.

TCP uses packet retransmissions to provide in-order, lossless packet delivery. Packet retransmissions can potentially introduce unacceptable end-to-end latency and thus re-sending media data may not be appropriate because it would arrive too late for display at the receiver.

Recently, several approaches have been proposed to overcome these problems [4], [26], [14], [25], [18]. These TCP-based stored media streaming approaches use a combination of *client-side buffering* and efficient *QoS adaptation* of the streamed data. Client-side buffering essentially borrows some current bandwidth to prefetch data to protect against future rate reduction. Thus, with sufficient client-side buffering, short-term rate variations introduced by TCP as well as the delay introduced by packet retransmissions can both be handled. QoS adaptation allows fine-grained adjustment of the rate-distortion tradeoff, i.e., rate versus quality adjustment, during the transmission process and thus allows handling long-term rate changes by adjusting quality dynamically.

TCP-based streaming is desirable because TCP offers several well known advantages. TCP provides congestion controlled delivery which is largely responsible for the remarkable stability of the Internet despite an explosive growth in traffic, topology and applications [13]. TCP handles flow control and packet losses, so applications do not have to worry about recovery from packet losses. This issue is especially important because the effects of packet loss are non-uniform and can quickly become severe. For instance, loss of the header bits of a picture typically renders the whole picture and possibly a large segment of surrounding video data unviewable while loss of certain pixel blocks may be virtually imperceptible. Thus media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [27] that potentially have high bandwidth and processing overhead. Finally, given the large TCP user base, there is great interest in improving its performance. Such improvements can also help media streaming.

In this paper, we study the feasibility of using TCP for low-latency media streaming. We are concerned with *protocol latency*, which we define as the time difference from a write on the sender side to a read on the receiver side, i.e., socket to socket latency. Low latency streaming is desirable for several applications. For streaming media, control operations such as the sequence of start play, fast forward and restart play become more responsive because the network and the end-points have low delay in the data path. For video on demand servers, low latency streaming offers faster channel surfing (starting and stopping of different channels). Similarly, multimedia document browsing becomes more responsive. Finally, with sufficiently low latency streaming, interactive streaming applications become feasible.

Although there have been several studies that describe the packet delays experienced by TCP flows [1], [23], [9] there has been much less work describing the protocol latency observed by applications streaming over TCP. This lack of study of protocol latency is partly because TCP has often been considered impractical for streaming applications and thus few TCP-based streaming applications have been developed. In addition, non-QoS adaptive streaming applications require large buffering at the ends to handle bandwidth variations, so protocol latency can be a second order effect. Fortunately, with quality adaptive streaming applications, the buffering needed at the end-points can be tuned and made small and thus protocol latency becomes more significant.

This paper examines TCP protocol latency by showing the latency observed at the sender side, receiver side and the network under various network conditions. Our results show that, surprisingly, a significant portion of the protocol latency occurs due to TCP's *send buffer* and this latency can be eliminated by making some simple send-buffer modifications to the sender side TCP stack without changing the TCP protocol in any way. These modifications dynamically adapt (reduce) the send buffer size and have similarity to the send-buffer tuning work by Semke [29]. However, unlike their work which focuses on improving TCP throughput, this work focuses on reducing

This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

socket to socket latency.

Our experiments show that these modifications reduce the average protocol latency to well within the interactive latency limits of approximately 200 ms [12] when the underlying network round-trip time is less than 100 ms (coast-to-coast round-trip time in the US [9]).[1] This reduction in latency comes at a small expense in throughput.

At this point, it may seem that our send-buffer reduction approach would reduce latency from the TCP layer but would re-introduce it at the application layer, and thus the net effect on application-level end-to-end latency is unclear. Fortunately, this issue is not a real problem because we assume that latency-sensitive applications are 1) quality-adaptive and 2) they use `poll` and non-blocking `write` calls on the sending side. The benefit of low latency streaming is that the sending side can wait longer before making its quality adaptation decisions, i.e., it has more control and flexibility over what data should be sent and when it should be sent. For instance, if the low protocol-latency network doesn't allow the application to send data for a long time, the sending side can drop low-priority data and then send 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 lose control over quality adaptation when that data is delayed in the send buffer). The non-blocking write calls ensure that the sending side is not blocked from doing other work (such as media encoding) while the network is busy. In addition, the application does not spend CPU cycles polling for the socket-write ready condition since the kernel informs the application when the socket is ready for writing.

The sender-side modifications reduce average protocol latency significantly but are not sufficient for interactive streaming applications since many packets can still observe latencies much higher than 200 ms. These latency spikes occur due to packet dropping and retransmissions and thus motivate the need for mechanisms that reduce packet dropping in the network. One such mechanism is explicit congestion notification (ECN) [24]. With ECN, routers use active queue management [5] and indirectly inform TCP of impending congestion by setting an ECN bit on packets that would otherwise have been dropped. TCP uses the ECN bit to pro-actively reduce its sending rate, thus reducing network load and packet dropping in the network. This paper explores how TCP enabled with ECN effects protocol latency.

The next section presents our modifications to the TCP sending side to reduce protocol latency. Section III describes our experimental methodology for evaluating the latency behavior of TCP. Section IV presents our results. Section V summarizes related work in multimedia and low latency streaming, and TCP congestion control. Section VI discusses future work in low-latency TCP streaming, and finally, Section VII presents our conclusions.

II. TCP Send Buffer

This section discusses our approach to reducing protocol latency by dynamically adjusting the TCP *send buffer* size. TCP is a window-based protocol, where its window size is the maximum number of distinct (and unacknowledged) packets in flight in the network at any time. TCP adapts the size of its window based on congestion feedback and stores this size value in the TCP variable CWND. TCP uses a *fixed size* send buffer to store application data before the data is transmitted. This buffer has two functions. First, it handles rate mismatches between the application sending rate and TCP's transmission rate. Second, it is used to keep copies of the packets in flight (its current window) so they can be retransmitted when needed. Since CWND stores the number of packets in flight, its value can never exceed the send buffer size.

From a latency perspective, the fixed size send buffer can introduce significant latency into the TCP stream. As a concrete example, the send buffer in most current Unix kernels is at least 64KB. For a 300 Kbs video stream, a full send buffer contributes 1700 ms of delay. By comparison, the round trip delay may lie between 50-100 ms for coast-to-coast transmission within the United States. In addition, the buffering delay increases for smaller bandwidth streams or with increasing competition since the stream bandwidth goes down.

We believe that for latency sensitive streams, sender-side buffering should be moved out of the TCP stack and applications should be allowed to handle buffering as much as possible. This approach is in keeping with the end-to-end principle followed by TCP where the protocol processing complexity is moved out of the network as much as possible to the stream end points. We do not modify TCP receive-side buffering because our applications aggressively remove data from the receive-side buffer. Thus, receive-side delay is only as issue when packets are retransmitted by TCP. This issue is discussed further in Section IV-C.

A. Adapting Send Buffer Size

One method for reducing the latency caused by the send buffer is to statically reduce the size of the send buffer. This approach has a negative effect on the throughput of the flow if the number of packets in flight (CWND) is limited by the send buffer (and not by the network congestion signal). In this case, the flow throughput is directly proportional to the send buffer size and decreases with a smaller send buffer. We reject this approach because although our main goal is to reduce protocol latency, we also aim to achieve throughput comparable to standard TCP.

Now suppose that the send buffer was sufficiently large that TCP could adjust the value of CWND based only on congestion (and receiver buffer) feedback. It should be clear that for this condition to hold, the size of the send buffer should be at least CWND packets. A smaller value would limit CWND to the send buffer size and reduce the throughput of the flow. A larger value should not affect throughput significantly since TCP would not send more than CWND packets anyway. However, a larger value increases protocol latency because only CWND packets can be in flight at any time, and thus the rest of the packets have to sit in the send buffer until acknowledgments have been received for the previous packets.

This discussion shows that adjusting the send buffer size to follow CWND can reduce protocol latency without signif-

[1] We are focusing on protocol latency (or socket to socket latency) and ignore the processing times at the application end points in this paper.

icantly affecting flow throughput. We have implemented this approach, as described in Section II-B. This approach impacts throughput when TCP could have sent a packet but there are no new packets in the send buffer. This condition can occur for several reasons. First, with each acknowledgment arrival, standard TCP has a packet in the send buffer that it can send immediately. If the send buffer size is limited to CWND, then TCP must inform the application and the application must write the next packet before TCP can send it. Thus, system timing and scheduling behavior can affect TCP throughput. Second, back-to-back acknowledgment arrivals exacerbate this problem. Finally, the same problem occurs when TCP increases CWND. These adverse affects on throughput can be reduced by adjusting the buffer size so that it is larger than CWND. To study the impact on throughput, we experimented with three different send buffer configurations as described in the next section.

B. Send Buffer Modifications

To reduce sender-side buffering, we have made a small send-buffer modification to the TCP stack on the sender side in the Linux 2.4 kernel. This modification can be enabled per socket by using a new SO_TCP_MIN_BUF option, which limits the send buffer size to $A*$CWND$+$MIN$(B,$ CWND$)$ packets 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. We assume, as explained in more detail later, that the size of each application packet is MSS (maximum segment size). With the send-buffer modification, an application is blocked from sending when there are $A *$ CWND $+$ MIN$(B,$ 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. From now on, we call a TCP stream that has the SO_TCP_MIN_BUF option turned on with parameters A and B, a MIN_BUF(A, B) stream.

With these modifications to TCP and assuming a MIN_BUF$(1, 0)$ stream, the send buffer will have at most CWND packets after an application writes a packet to the socket. TCP can immediately transmit this packet since this packet lies within TCP's window. After this transmission, TCP will again allow the application to write data. Thus as long as CWND is non-decreasing, TCP will not add any buffering delay to a stream. Delay is added only during congestion when TCP decreases the value of CWND. Our experiments in Section IV show that this delay is generally much smaller than the standard TCP send-buffer delay.

The SO_TCP_MIN_BUF option exposes the parameter A and B, because they represents a tradeoff between latency and throughput. Larger values of A or B add latency but can improve throughput as explained in the previous section. We experimented with three MIN_BUF streams: MIN_BUF$(1, 0)$, MIN_BUF$(1, 3)$ and MIN_BUF$(2, 0)$. These streams should have increasing latency and throughput. A MIN_BUF$(1, 0)$ stream is the default stream with the least protocol latency. We expect a MIN_BUF$(2, 0)$ stream to have the same throughput as TCP because there are CWND extra packets in the send buffer and even if acknowledgments for all packets in the previous window come simultaneously, the next window of packets can be sent without first getting packets from the application. Thus a MIN_BUF$(2, 0)$ stream should behave similarly (in terms of throughput) to a TCP stream [16]. Finally, we chose a MIN_BUF$(1, 3)$ stream to see how three extra packets affect latency and throughput. If no more than three acknowledgments arrive back to back, then this stream should behave similar to TCP in terms of bandwidth. Section IV presents latency and throughput results for the three streams. Briefly, our results show that 1) MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows has similar latencies and these latencies are much smaller than MIN_BUF$(2, 0)$ or TCP flows, and 2) while a MIN_BUF$(1, 0)$ flow suffers 30 percent bandwidth loss, the MIN_BUF$(1, 3)$ flow suffers less than 10 percent bandwidth loss. Thus, the MIN_BUF$(1, 3)$ flow represents a good latency-bandwidth compromise.

1) Sack Correction: The previous discussion about the send buffer limit applies for a non-SACK TCP implementation. For TCP SACK [15], we make a *sack correction* by adding an additional term *sacked_out* to $A *$ CWND $+$ MIN$(B,$ CWND$)$. The sacked_out term (or an equivalent term) 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.

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 $A *$ CWND $+$ MIN$(B,$ 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.

C. Application Model

In this paper, we are concerned with protocol latency. We ignore the processing time at the application end points since these times are application dependent. However, these times must also be included when studying the feasibility of a low latency application such as an interactive media streaming application.

We assume that latency-sensitive applications use non-blocking read and write socket calls. The protocol latency is measured from when the packet write is initiated on the sender side to when the same packet is completely read on the receiver side. The use of non-blocking calls generally means that the application is written using an event-driven architecture [22].

We also assume that applications explicitly align their data with packets transmitted on the wire (application level framing) [2]. This alignment has two benefits: 1) it minimizes any latency due to coalescing or fragmenting of packets below the application layer, 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 writes MSS (maximum segment size) sized 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 [17]. 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 would be to return a write error on an MSS change for a MIN_BUF socket.

III. Experiments

In this section, we describe the tests we performed to evaluate the latency and throughput behavior of standard TCP and MIN_BUF streams under various network conditions. All streams use TCP SACK and MIN_BUF streams use the sack correction described in Section II-B. We performed our experiments on a Linux 2.4 test-bed that simulates WAN conditions by introducing delay at an intermediate Linux router in the test-bed.

A. Experimental Scenarios

The first set of tests considers the latency response of TCP streams to a sudden increase in congestion. Increase in congestion is triggered with three types of flows: 1) competing long-lived TCP flows, 2) a flash crowd of many small TCP flows, and 3) a competing constant bit rate (CBR) flow, such as a UDP flow. The long-lived competing flows are designed to simulate other streaming traffic. The flash crowd of short TCP flows simulates web transfers. In our experiments, the small flows have fixed packet sizes and they are run back to back so that the number of active TCP connections is roughly constant [11]. The CBR flow simulates non-responsive UDP flows.

While these traffic scenarios do not necessarily accurately model reality, they are intended to explore and benchmark the latency behavior of TCP and MIN_BUF streams in a well characterized environment. These tests are designed to emulate a heavily loaded network environment.

The second set of tests measures the relative throughput share of TCP and MIN_BUF streams. Here we are mainly concerned with the bandwidth lost by MIN_BUF traffic. These experiments are performed with the same types of competing flows described above.

We are interested in several metrics of a latency-sensitive TCP flow. We explore three metrics in this paper: 1) protocol latency distribution, and specifically, the percentage of packets that arrive at the receiver within a *delay threshold*, 2) average packet latency, and 3) normalized throughput, the ratio of the throughput of a MIN_BUF flow to a TCP flow. We choose two delay thresholds, 200 ms, which is related to interactive streaming performance, and 500 ms, which is somewhat arbitrary, but chosen to represent the requirements of responsive media streaming control operations.

In addition to comparing the latency behavior of standard TCP and MIN_BUF streams, we are also interested in understanding the effects on protocol latency of ECN enabled TCP.

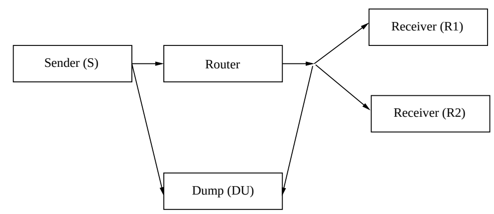


Fig. 1. Network Topology

Our results describe how this "streaming friendly" mechanism affects protocol latency.

B. Network Setup

All our experiments use a single-bottleneck "dumbbell" topology and FIFO scheduling at the bottleneck. The network topology is shown in Figure 1. 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 receiver *R1*. This stream is generated by an application that follows the application model described in Section II-C. The sender generates cross traffic for both receivers *R1* and *R2*. The router runs `nistnet` [20], a network emulation program that allows the introduction of additional delay and bandwidth constraints in the network path. The machine DU is used to dump TCP traffic for further analysis. The protocol latency 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 ms of each other using NTP.

We chose three round-trip times (RTT) for the experiments and conducted separate experiments for each RTT. The RTTs were 25 ms, 50 ms and 100 ms. These RTTs approximate some commonly observed RTTs on the Internet. The cable modem from our home to work has 25 ms delay. West-coast to west-coast sites or East-coast to East-coast sites in the US observe 50 ms median delay and west-coast to east-coast sites in the US observe 100 ms median delay [9].

We run our experiments over standard TCP and ECN enabled TCP. For each RTT, two router queue lengths are chosen so that bandwidth is limited to 12 Mbs and 30 Mbs. The TCP experiments use tail dropping. For ECN, we use DRED active queue management [7], which is supported in `Nistnet`. DRED is a RED variant that is implemented efficiently in software. The *drdmin*, *drdmax* and *drdcongest* parameters of DRED were chosen to be 1.0, 2.0 and 2.0 times the bandwidth-delay product, respectively. DRED sends ECN messages for 10 percent of packets when the queue length exceeds *drdmin*, progressively increasing the percentage until packets are dropped when the queue length exceeds *drdcongest*. Unlike RED, DRED does not average queue lengths.

IV. Results

In this section, we discuss the results of our experiments. We start by showing the effects of using TCP and MIN_BUF streams on protocol latency. Then we quantify the throughput loss of these streams. We investigate the latencies observed at

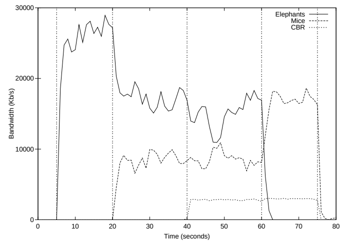


Fig. 2. The bandwidth profile of the cross traffic (15 elephants, 80 mice consuming about 30% bandwidth and 10% CBR traffic)

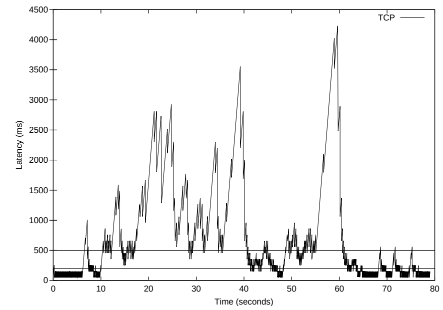
the sender, network and the receiver of TCP streams and the causes of each latency. Finally, we explore using ECN enabled TCP to improve protocol latencies.

A. Protocol Latency

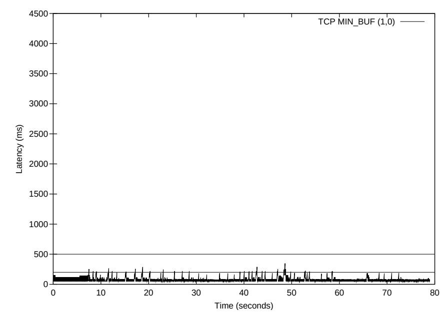
Our first experiment shows the protocol latency of TCP and MIN_BUF streams in response to dynamically changing network load. The experiment is run for about 80 seconds with load being introduced at various different time points in the experiment. The TCP or MIN_BUF long-lived stream being measured is started at t = 0s. We refer to this flow as the *latency* flow. Then at t = 5s, 15 other long-lived (*elephant*) flows are started, 7 going to receiver R1 and 8 going to receiver R2. At t = 20s, each receiver initiates 40 simultaneous short-lived (*mouse*) TCP flows. A mouse flow is a repeating short-lived flow that starts the connection, transfers 20KB of data, ends the connection and then repeats this process continuously [11]. The number of mouse flows was chosen so that the mouse flows would get approximately 30 percent of the total bandwidth. At t = 40s, CBR traffic that consumes 10 percent of the bandwidth is started. At t = 60s, the elephants are stopped and then the mice and the CBR traffic are stopped at t = 75s. Figure 2 shows the cross traffic (elephants, mice and CBR traffic) for a 30 Mbs bandwidth, 100 ms RTT experiment. Other experiments have a similar bandwidth profile.

Figure 3 shows the results of a run with standard TCP and MIN_BUF$(1, 0)$ streams when the bandwidth limit is 30Mbs and the round trip time is 100 ms. Both these streams originate at sender S and terminate at receiver R1. The figures shows the protocol latency of the latency flow as a function of packet receive time. The two horizontal lines on the *y* axis show the 200 ms and the 500 ms latency threshold.

Figure 4 shows the protocol latency of the three MIN_BUF configurations. Note that in this figure, the maximum value of the y axis is 500 ms. These figures show that the MIN_BUF streams have significantly lower protocol latency than a standard TCP stream. They show that, as expected, the MIN_BUF$(1, 0)$ flow has the lowest protocol latency while the MIN_BUF$(2, 0)$ has the highest protocol latency among the MIN_BUF flows. Looking at the throughput profile of the latencies flows (now shown here), we found that the protocol latency of TCP and MIN_BUF$(2, 0)$ is highest when the flow throughput is lowest. However, the protocol latency of MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows is not affected as much by their changing throughput. The reason is that the TCP send buffer drains slowly when the bandwidth available to the latency stream goes down. Since TCP and MIN_BUF$(2, 0)$ flows allow



(a) TCP



(b) MIN_BUF(1,0)

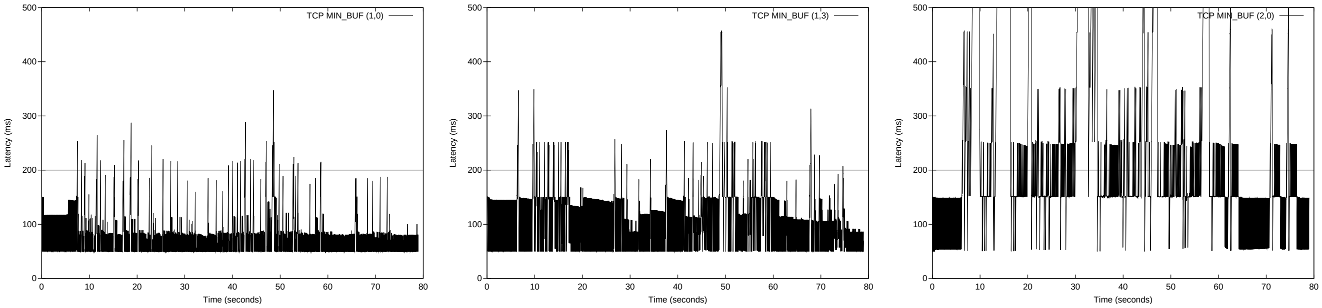
These figures show the protocol latency of packets plotted as a function of packet receive time. The bandwidth limit for this experiment is 30 Mbs and the round trip time is 100 ms. The horizontal lines on the figures show the 200 ms and 500 ms latency threshold.

Fig. 3. A comparison of the protocol latencies of TCP and MIN_BUF(1,0) streams

the send buffer to fill up more than the other two flows, these flows observe higher protocol latencies. The send buffer does not significantly affect the protocol latency in MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows. The latency spikes seen in these flows are chiefly a result of TCP congestion control and retransmission as discussed in Section IV-C.

The protocol latency distribution for this experiment is shown in Figure 5. The experiment was performed with 30Mbs and 12Mbs bandwidth limit and with 100 ms, 50ms and 25 ms RTT. Each experiment was performed 8 times and the results presented show the numbers accumulated over all the runs. The vertical lines show the 200 and 500 ms delay thresholds. The figures show that in all cases a much larger percent of TCP packets lie outside the delay thresholds as compared to MIN_BUF flows. Note that the x axis, which shows the protocol latency in milliseconds, is on a log scale. The figures show that, as expected, the percent of packets with large delays increases with increasing RTT and decreasing bandwidth. The percent of packets delivered within the 200 and 500 ms delay thresholds is summarized in Table I. This table also shows that the packets delivered within the delay thresholds is very similar for MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows.

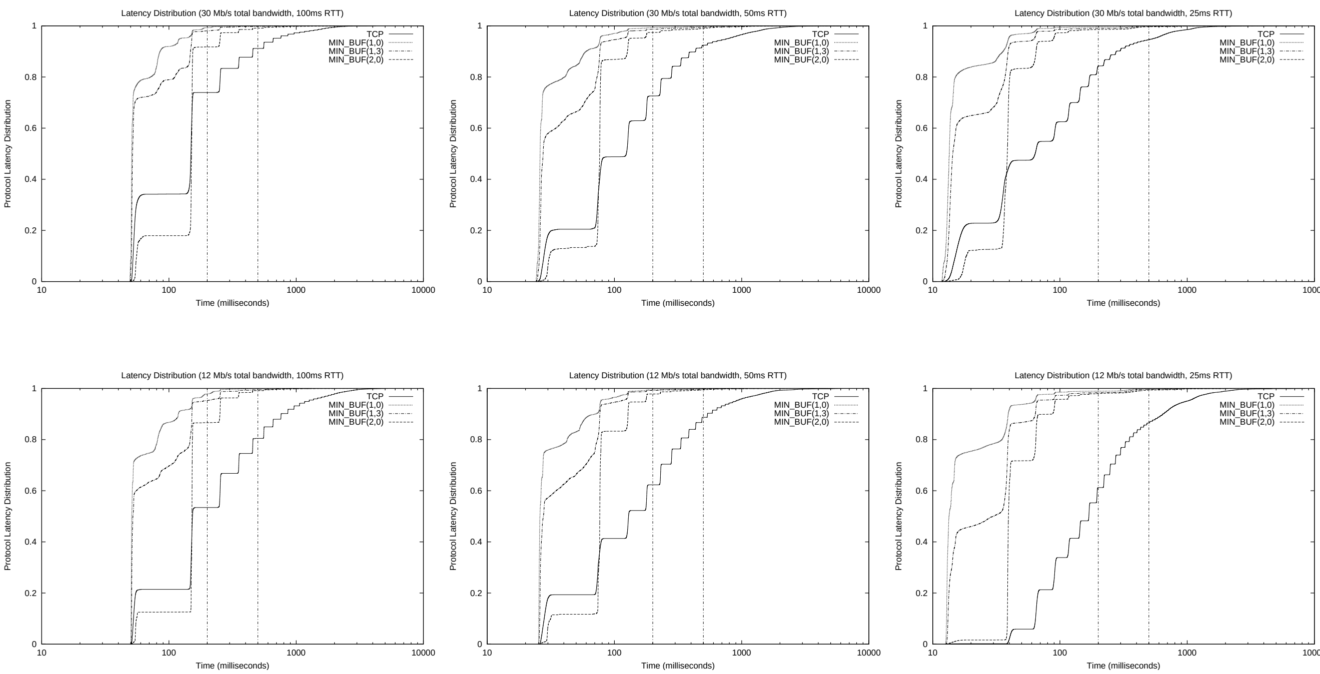
The average (one way) protocol latency for each configuration is shown in Table II. Each experiment was performed 8 times and these numbers are the mean of the 8 runs. The table



(a) MIN_BUF(1,0) (b) MIN_BUF(1,3) (c) MIN_BUF(2,0)

These experiments were performed under the same conditions as described in Figure 3. Note that the maximum value of the y axis is 500 ms, while it is 4500 ms in Figure 3.

Fig. 4. A comparison of the protocol latencies of 3 MIN_BUF configurations



The experiment was performed with a 30Mbs and 12Mbs bandwidth limit and with 100 ms, 50ms and 25 ms RTT. The vertical lines show the 200 and 500 ms delay thresholds. The x axis, which shows the protocol latency in milliseconds, is on a log scale.

Fig. 5. Protocol Latency Distribution of standard TCP, MIN_BUF$(1, 0)$, MIN_BUF$(1, 3)$ and MIN_BUF$(2, 0)$ flows

shows that MIN_BUF flows have much lower average latency and the deviation across runs is also much smaller.

B. Throughput Loss

We are interested in the throughput loss of MIN_BUF streams. We measured the throughput of each of the flows as a ratio of the total number of bytes received to the duration of the experiment. Table III shows the normalized throughput of each flow, which is the ratio of the throughput of the flow to the TCP flow. Again, these numbers are the mean (and 95% confidence interval) over 8 runs.

The table shows that the MIN_BUF$(2, 0)$ flows receive throughput close to standard TCP (within the confidence range). MIN_BUF$(2, 0)$ flows have CWND new packets that can be sent after a packet transmission. So even if all current CWND packets in flight are acknowledged almost simultaneously, TCP can send its entire next window of CWND packets immediately. Thus we expect that MIN_BUF(2,0) flows should behave similar to TCP flows.

The MIN_BUF$(1, 0)$ flows consistently receive the least throughput, about 70 percent of TCP. This result is not surprising because TCP has no new packets in the send buffer that can be sent after each packet is transmitted. TCP must ask the application to write the next packet to the send buffer before it can proceed with the next transmission. Thus, any scheduling or other system delays would make the MIN_BUF$(1, 0)$ flow an application-limited flow. TCP assumes that such flows need

		RTT = 100 ms		RTT = 50 ms		RTT = 25 ms	
Mbs	*Type*	*D200*	*D500*	*D200*	*D500*	*D200*	*D500*
30	std	0.73	0.91	0.72	0.92	0.84	0.94
30	m10	0.99	1.00	0.99	1.00	1.00	1.00
30	m13	0.98	1.00	0.99	0.99	0.99	1.00
30	m20	0.91	0.99	0.97	0.99	0.99	1.00
12	std	0.53	0.80	0.62	0.88	0.60	0.86
12	m10	0.98	1.00	0.99	1.00	0.99	1.00
12	m13	0.95	0.99	0.99	1.00	0.98	1.00
12	m20	0.86	0.99	0.97	0.99	0.98	0.99

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. The terms D200 and D500 refer to a delay threshold of 200 and 500 ms.

TABLE I

PERCENT OF PACKETS DELIVERED WITHIN 200 AND 500 MS THRESHOLDS FOR STANDARD TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ AND MIN_BUF$(2,0)$ FLOWS

Mbs	*Type*	*RTT = 100 ms*	*RTT = 50 ms*	*RTT = 25 ms*
30	std	226.31±0.87	218.84±40.34	138.61±21.0
30	m10	62.91±0.96	37.09±0.80	19.71±0.89
30	m13	76.19±2.71	51.54±3.73	28.29±1.70
30	m20	152.14±9.13	89.74±5.32	48.21±2.19
12	std	369.22±50.32	260.27±23.15	296.25±47.49
12	m10	69.73±2.15	38.50±1.09	25.94±1.80
12	m13	91.42±6.81	49.17±2.03	39.08±3.39
12	m20	162.26±6.06	87.90±1.46	61.31±5.59

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. All average latency numbers (together with 95% confidence intervals) are shown in milliseconds.

TABLE II

AVERAGE LATENCY OF STANDARD TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ AND MIN_BUF$(2,0)$ FLOWS

less bandwidth and reduces the window and thus the transmission rate of such flows.

Interestingly, the MIN_BUF$(1,3)$ flows receive throughput close to TCP, about 90 percent of TCP or more. Three additional packets in the send buffer (in addition to the CWND packets in flight) seem to reduce the throughput loss due to the artificial application-flow limitation introduced by MIN_BUF$(1,0)$ flows.

For a latency sensitive, quality-adaptive application, one metric for measuring the average flow quality could be the product of the percent of packets that arrive within a delay threshold and the normalized throughput of the flow. This relative metric is related to the number of packets that arrive within the delay threshold across different flows. Thus a larger value of this metric could imply better perceived quality. From the numbers presented above, MIN_BUF(1,3) flows have the highest value for this quality metric because both their delay threshold numbers (shown in Table I) and normalized throughput numbers (shown in Table III) are close to the best numbers of other flows.

Mbs	*Type*	*RTT = 100 ms*	*RTT = 50 ms*	*RTT = 25 ms*
30	std	1.00	1.00	1.00
30	m10	0.66±0.11	0.71±0.08	0.76±0.10
30	m13	0.96±0.12	0.87±0.08	0.92±0.12
30	m20	1.02±0.18	1.13± 0.36	0.91±0.10
12	std	1.00	1.00	1.00
12	m10	0.67±0.09	0.76±0.05	0.89±0.11
12	m13	0.92±0.15	1.06±0.09	1.08±0.22
12	m20	1.13±0.16	1.08±0.14	1.12±0.17

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. The normalized throughput (NT) is the ratio of throughput of each flow to the ratio of a standard TCP flow.

TABLE III

THE NORMALIZED THROUGHPUT OF A STANDARD TCP FLOW AND MIN_BUF FLOWS

C. *Understanding Worst Case Behavior*

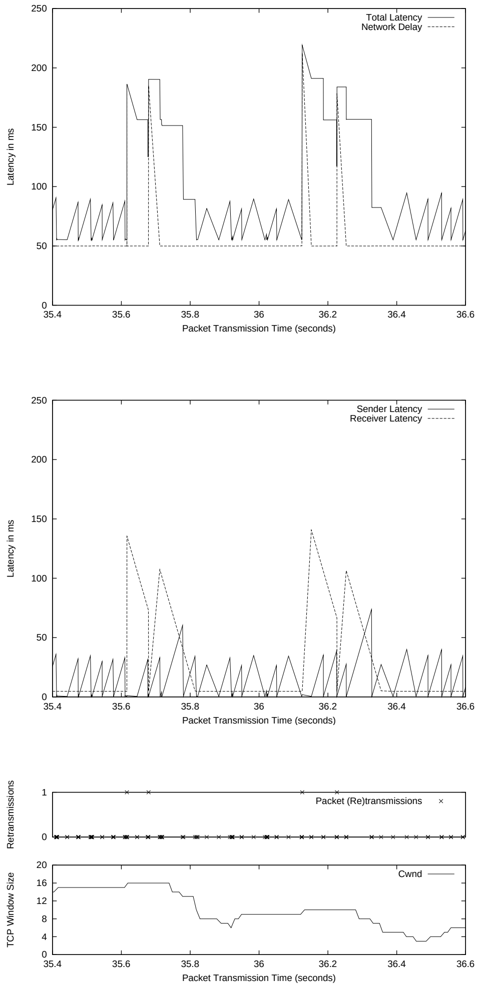
Figure 4 shows that MIN_BUF$(1,0)$ and MIN_BUF$(1,3)$ flows occasionally show protocol latency spikes even though they have small send buffers. To understand the cause of these spikes, we measured the delays experienced by each packet on the sender side, in the network and on the receiver side.

Figure 6 shows these delays for a small part of the experiment when packets were lost and retransmitted. The sender latency of each packet is the time from when an application writes to the socket to TCP's first transmission of the packet. The network delay is the time from the first transmission of each packet to the first arrival at the receiver. The receiver latency is the time from the first arrival of each packet to an application read. Figure 6 shows that the latency spikes are primarily caused by packet losses and retransmissions. In particular, the protocol (or total) latency does not depend significantly on the flow throughput (or the congestion window size). For instance, the congestion window size at t=35.5 ms and t=36.5 ms is 15 and 4, but the total latency at these times is roughly the same.

Packet retransmissions initially cause the network delay to increase, followed by an increase in the receiver latency. The receiver latency increases because TCP delivers packets in order and a lost packet temporarily blocks further packets from being released to the application. In addition, the sender latency increases slightly because TCP reduces its congestion window after a packet loss. Thus packets that were already accepted into the send buffer are delayed. Note that after a packet loss, increases in latency at the network, receiver and the sender are typically not additive (for any given packet) since they are shifted in time. However, this time shifting implies that the total latency stays high for several packets after a packet is dropped. These findings motivated the need to explore mechanisms that can reduce packet dropping. One such mechanism that has been studied by the networking community is explicit congestion notification (ECN) [24], [28].

D. *Protocol Latency with ECN*

ECN enabled routers inform TCP senders of impending congestion by setting an ECN bit on certain packets. When an ECN enabled TCP sender receives such a packet, it takes pro-active



This experiment was performed with a MIN_BUF$(1,0)$ flow. The bandwidth limit is 30 Mbs and the RTT is 100 ms. All figures are plotted as a function of the packet transmission time. These figures show that the sender side latency is small for MIN_BUF$(1,0)$ flows and that spikes in total latency occur primarily due to packet loss and retransmissions.

Fig. 6. The packet delay on the sender side, the network and the receiver side

measures to reduce its sending rate to avoid packet dropping in the router.

We ran the same set of experiments as described in Section IV-A to measure and compare the protocol latency of ECN flows and MIN_BUF (with ECN) flows. Figure 7 shows the bandwidth profile of the competing traffic. Figures 8 and 9 show the comparative protocol latencies. These figures are generated from experiments that are similar to those shown in Figure 3 except we enabled ECN at the end points and used DRED active queue management at the intermediate router.

These figures show that the protocol latency spikes are reduced in all cases when compared to Figure 4. A close look at the raw data showed that ECN reduced packet dropping and retransmissions and thus had fewer spikes. More experimental results for ECN can be found in an extended version of this paper [8].

ECN in these experiments showed several interesting band-

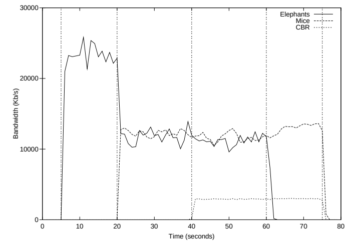
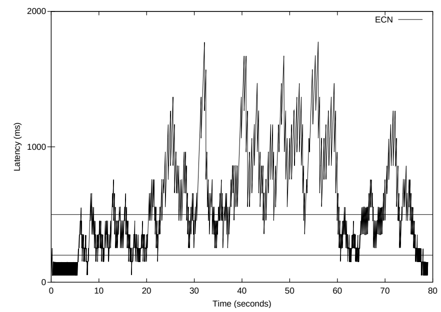
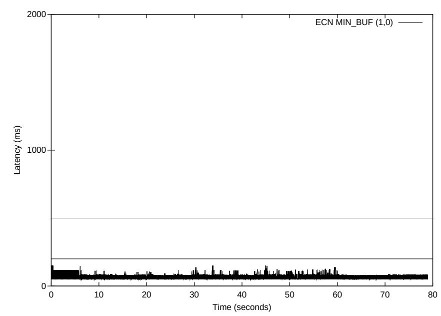


Fig. 7. The bandwidth profile of the cross traffic (15 elephants, 80 mice consuming about 50% bandwidth and 10% CBR traffic)



(a) ECN

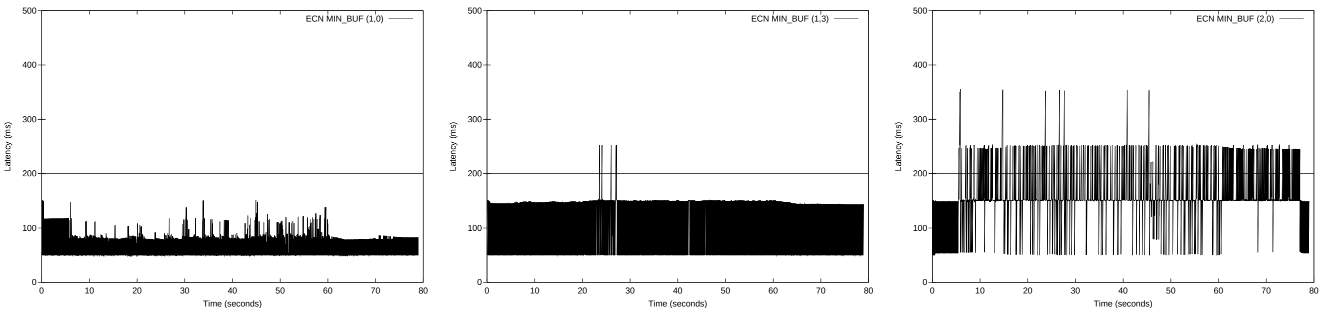


(b) MIN_BUF(1,0) with ECN

These figures show the protocol latency as a function of packet receive time. The bandwidth limit for this experiment is 30 Mbs and the round trip time is 100 ms. The horizontal lines on the figures show the 200 ms and 500 ms latency threshold.

Fig. 8. A comparison of the protocol latencies for ECN and MIN_BUF$(1,0)$ streams

width related properties. First, the mouse bandwidth was tuned to 50 percent of the bandwidth capacity as shown in Figure 7, instead of 30 percent as shown in Figure 2. The mice were able to achieve their bandwidth share quickly and more accurately. With TCP, in some configurations (lower bandwidth and smaller RTT), the mice were not able to achieve 50 percent bandwidth share even when the application starts very large numbers of mice. This is because the elephants are very aggressive and the mouse are unable to connect for long periods of time. In addition, the ratio of mice to elephants needed to achieve fair sharing between the mice and the elephants is much smaller for ECN than with regular TCP flows. Thus, elephants do not steal as much bandwidth from mice and also have a smoother throughput profile (not shown here). We believe that



(a) MIN_BUF(1,0) (b) MIN_BUF(1,3) (c) MIN_BUF(2,0)

These experiments were performed under the same conditions as described in Figure 8. Note that the maximum value of the y axis is 500 ms, while it is 2000 ms in Figure 8.

Fig. 9. A comparison of the protocol latencies of 3 MIN_BUF configurations

although ECN may loose throughput compared to TCP for long lived flows, its reduced aggressiveness leads to fewer retransmissions and thus it is desirable for low latency streaming.

V. Related Work

The feasibility of TCP-based stored media streaming has been studied by several researchers. Generally, the tradeoff in these QoS adaptive approaches is short-term improvement in video quality versus long term smoothing of quality. Rejaie [26] 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. Krasic [14] contends that new compression practices and reduced storage costs make TCP a viable and attractive basis for streaming stored content and uses standard TCP, instead of a TCP-friendly scheme, for media streaming. Feng [4] and Krasic use priority-based streaming, which allows a simpler and more flexible implementation of QoS adaptation. We believe that similar QoS adaptive approaches will be useful for low latency streaming also.

Researchers in the multimedia and networking community have proposed several alternatives to TCP for media streaming [30], [6]. 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 non-uniform 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 [27] 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 [21] explores introducing FEC as a transparent layer under an ARQ scheme to improve transmission efficiency.

Popular interactive streaming applications include Voice over IP (VoIP) products such as Microsoft NetMeeting [19]. NetMeeting provides reasonable voice quality over a best effort network but is implemented over UDP because the delays introduced by TCP are considered unacceptable. This paper shows that MIN_BUF TCP should yield acceptable delays, especially for QoS adaptive applications. For interactive applications, ITU G.114 [12] 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.

Our send-buffer adaptation approach is similar to the buffer tuning work by Semke [29]. Semke tunes the send buffer size to between 2∗CWND and 4∗CWND to improve the throughput of a high bandwidth-delay connection that is otherwise limited by the send buffer size. The 4∗CWND 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 CWND and also achieve significant reduction in protocol latency.

Many differentiated network services have been proposed for low latency streaming. These schemes are complementary to our work since, generally, a MIN_BUF TCP implementation can be used for the low delay flow. Hurley [10] 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 QoS adaptation decisions.

Active queue management and explicit congestion notification (ECN) [24] have been proposed for improving the packet loss rates of TCP flows. Salim [28] shows ECN has increasing throughput advantage with increasing congestion levels and ECN flows have hardly any retransmissions. Feng [3] shows that adaptive active queue management algorithms (Adaptive RED) and more conservative end-host mechanisms can significantly reduce loss rates across congested links.

Claffy [1] presents the results of a measurement study of the T1 NSFNET backbone and delay statistics. In 1992, the one way median delays between end points ranges from 20 to 80 ms with a peak at 45 ms. Newer data in 2001 [9] shows that the median RTT for East-coast to East-coast or West-coast to West-coast is 25-50 ms and East-coast to West-coast is about 100 ms. We use these median results in our experiments. US to Europe median RTT is currently 200 ms. While the 200 ms median RTT makes interactive applications challenging, responsive control operations for streaming media should be possible.

VI. Future Work

The results in this paper are based on experiments conducted over an experimental network test-bed. While simulating our experiments under more exhaustive conditions using a network simulator, such as `ns`, would be useful, the task is not trivial because `ns` does not simulate the send buffer. Thus a simulator for the send buffer would have to be implemented. In addition, we are interested in observing whether scheduling and other timing effects change the latency or throughput behavior of MIN_BUF streams. Simulating such effects is beyond the scope of `ns`.

We have explored adapting the send buffer using three different sizes for MIN_BUF(A, B) flows. These different configurations, with increasing buffer sizes, have increasing latency and throughput. Another approach for adapting the send buffer is to auto-tune the values of A and B so that the send buffer contributes a certain amount of delay while providing the best possible throughput.

We are currently implementing a streaming quality-adaptive media server that will allow channel surfing as well as basic control operations such as fast forward, stop, rewind, etc. We plan to compare the latency of these operations using standard TCP versus MIN_BUF flows. We are also integrating a real-time MPEG encoder into the media server, which will allow us to investigate some of the challenges raised by low latency streaming, including the handling of late packets.

VII. 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. These approaches use a combination of client-side buffering and QoS adaptation to overcome various problems that were considered inherent with TCP-based media streaming.

The success of TCP-based streaming led us to explore the limits to which TCP can be used for low-latency media streaming. Low latency streaming allows responsive streaming control operations and sufficiently low latency streaming would make interactive applications feasible. We examined adapting the TCP send buffer size based on TCP's congestion window to reduce protocol latency or application perceived network latency. Our results show that this simple idea reduces protocol latency and significantly improves the number of packets that can be delivered within 200 ms and 500 ms thresholds.

References

[1] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Traffic Characteristics of the T1 NSFNET Backbone. In *INFOCOM*, pages 885–892, 1993.

[2] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, 1990.

[3] Wu-chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang S. Shin. Techniques for Eliminating Packet Loss in Congested TCP/IP Networks. Technical Report CSE-TR-349-97, U. Michigan, Nov 1997.

[4] Wu-Chi Feng, Ming Liu, Brijesh Krishnaswami, and Arvind Prabhudev. A Priority-Based Technique for the Best-Effort Delivery of Stored Video. In *Proc. of SPIE Multimedia Computing and Networking Conference (MMCN)*, January 1999.

[5] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[6] Sally Floyd, Mark Handley, and Eddie Kohler. Problem Statement for DCP. Work in progress, IETF Internet Draft draft-floyd-dcp-problem-00.txt, expires Aug 2002, Feb 2002.

[7] M. Gaynor. Proactive Packet Dropping Methods for TCP Gateways. `http://www.eecs.harvard.edu/~gaynor/final.ps`, October 1996.

[8] Ashvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting Low Latency TCP-Based Media Streams. Technical Report CSE-02-002, Oregon Graduate Institute, March 2002. `ftp://cse.ogi.edu/pub/tech-reports/2002/02-002.ps.gz`.

[9] Bradley Huffaker, Marina Fomenkov, David Moore, and kc claffy. Macroscopic Analyses of the Infrastructure: Measurement and Visualization of Internet Connectivity and Performance. In *A workshop on Passive and Active Measurements*, Amsterdam, April 2001.

[10] P. Hurley and J. Y. Le Boudec. A Proposal for an Asymmetric Best-Effort Service. In *Proceedings of IEEE/IFIP IWQoS 1999*, pages 132–134, May 1999.

[11] Gianluca Iannaccone, Martin May, and Christophe Diot. Aggregate Traffic Performance with Active Queue Management and Drop from Tail. *ACM Computer Communication Review*, 31(3), July 2001.

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

[13] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, pages 314–329, Stanford, CA, August 1988.

[14] Charles Krasic, Kang Li, and Jonathan Walpole. The Case for Streaming Multimedia with TCP. In *8th International Workshop on Interactive Distributed Multimedia Systems (iDMS 2001)*, pages 213–218, Sep 2001. Lancaster, UK.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. Internet RFC 2018, October 1996.

[16] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM*, 1996.

[17] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6. Internet RFC 1981, August 1996.

[18] Microsoft Inc. Windows Media Player. `http://www.microsoft.com/windows/windowsmedia`.

[19] Microsoft Inc. Windows NetMeeting. `http://www.microsoft.com/netmeeting`.

[20] NIST. The NIST Network Emulation Tool. `http://www.antd.nist.gov/itg/nistnet`.

[21] Jörg Nonnenmacher, Ernst W. Biersack, and Don Towsley. Parity-Based Loss Recovery for Reliable Multicast Transmission. *ACM/IEEE Transactions on Networking*, 6(4):349–361, 1998.

[22] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 USENIX Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[23] V. Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, pages 139–152, September 1997.

[24] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. Internet RFC 3168, September 2001.

[25] Real Networks. RealPlayer Streaming Media Player. `http://www.real.com`.

[26] Reza Rejaie, Mark Handley, and Deborah Estrin. Quality Adaptation for Congestion Controlled Video Playback over the Internet. In *ACM SIGCOMM*, pages 189–200, 1999.

[27] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM Computer Communication Review*, 27, 1997.

[28] J. Hadi Salim and U. Almed. Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks. Internet RFC 2884, July 2000.

[29] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP Buffer Tuning. In *ACM SIGCOMM*, pages 315–323, 1998.

[30] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. Internet RFC 2960, Oct 2000.