12-1997

Flow and Congestion Control for Internet Streaming Applications

Shanwei Cen

Calton Pu

Jonathan Walpole
Portland State University

Citation Details

Shanwei Cen ; Jonathan Walpole and Calton Pu, "Flow and congestion control for Internet media streaming applications", Proc. SPIE 3310, Multimedia Computing and Networking 1998, 250 (December 29, 1997); doi:10.1117/12.298426; http://dx.doi.org/10.1117/12.298426.

Flow and Congestion Control for Internet Media Streaming Applications

Shanwei Cen[a], Jonathan Walpole[b] and Calton Pu[b]

[a]Tektronix, Inc., Video and Networking Division
P.O.Box 500, M/S 50-490 Beaverton, Oregon 97077 USA

[b]Department of Computer Science, Oregon Graduate Institute
P.O.Box 91000, Portland, Oregon 97291 USA

ABSTRACT

The emergence of *streaming* multimedia players provides users with low latency audio and video content over the Internet. Providing high-quality, best-effort, real-time multimedia content requires adaptive delivery schemes that fairly share the available network bandwidth with reliable data protocols such as TCP. This paper proposes a new flow and congestion control scheme, SCP (Streaming Control Protocol), for real-time streaming of continuous multimedia data across the Internet. The design of SCP arose from several years of experience in building and using adaptive real-time streaming video players. SCP addresses two issues associated with real-time streaming. First, it uses a congestion control policy that allows it to share network bandwidth fairly with both TCP and other SCP streams. Second, it improves smoothness in streaming and ensures low, predictable latency. This distinguishes it from TCP's jittery congestion avoidance policy that is based on linear growth and one-half reduction of its congestion window. In this paper, we present a description of SCP, and an evaluation of it using Internet-based experiments.

Keywords: Internet, flow control, congestion control, adaptive systems, streaming multimedia systems

1. INTRODUCTION

The real-time distribution of continuous audio and video data via streaming multimedia applications accounts for a significant, and expanding, portion of the Internet traffic. Many research prototype media players have been produced, including Rowe's MPEG player,[1] the authors' distributed video player,[2] and McCanne's vic,[3] etc. Over the past two years, many commercial streaming media players have also been released, such as RealAudio and RealVideo players, Microsoft Netshow, and Vxtreme.

Common characteristics of media streaming applications include their need for high bandwidth, smooth data flow, and low and predictable end-to-end latency and latency variance. In contrast, the Internet is a best-effort network that offers no quality of service guarantees, and is characterized by a great diversity in network bandwidth and host processing speed, wide-spread resource sharing, and a highly dynamic workload. Hence, in practice, Internet-based applications experience large variations in available bandwidth, latency, and latency variance.

To address these problems the new generation of Internet-based multimedia applications use techniques such as buffering and feedback-based adaptation of presentation quality. Buffering is used at the sender, receiver, or both, to mask short-term variations in the available bandwidth or latency. Through adaptation, applications scale media quality in one or more quality dimensions to better utilize the currently available bandwidth. For example, real-time play-out can be preserved in the presence of degraded bandwidth by adaptively sacrificing other presentation quality dimensions such as video frame rate or spatial resolution.

Dynamic adaptation is a powerful approach, but it requires new flow and congestion control mechanisms for accurate discovery and appropriate utilization of the available network bandwidth. Congestion control mechanisms must determine, dynamically, the share of the network bandwidth that can be fairly used by the adaptive application in the presence of competing traffic. If the mechanisms are not sensitive enough

Please send correspondence to Shanwei Cen at shanwei.cen@tek.com

SPIE Vol. 3310 • 0277-786X/97/$10.00

to competing traffic the potentially high multimedia data rates could cause serious network congestion. On the other hand, if they are too sensitive they will under-utilize the bandwidth and presentation quality will be forced to degrade unnecessarily. In practice, such congestion control mechanisms must share bandwidth fairly among competing multimedia streams and must be compatible with TCP congestion control,[4] since TCP is the base protocol for currently-dominant HTTP and FTP traffic.

Flow control mechanisms should attempt to minimize end-to-end latency due to buffering, and maximize the smoothness of the data stream. Reliability through indefinite data retransmission is usually not desirable, since streaming applications can often tolerate some degree of quality degradation due to data loss, but are usually less tolerant of the delay introduced by the retransmission of lost data.

This paper presents a flow and congestion control scheme, called SCP (Streaming Control Protocol), for unicast media streaming applications. Like TCP, SCP employs sender-initiated congestion-detection through positive acknowledgment, and uses a similar window-based policy for congestion avoidance. The similarities in congestion avoidance between SCP and TCP make SCP robust and a good network citizen, and enable SCP and TCP to share the Internet fairly.

Unlike TCP, when the network is not congested, SCP invokes a hybrid rate- and window-based flow control policy that maintains smooth streaming with maximum throughput and low latency. Conversely, TCP repeatedly increases its congestion window size until packets are lost, then halves its window size (a behavior of TCP-Reno). One consequence of this behavior is that TCP sessions exhibit burstiness and develop long end-to-end latency due to the build up of packets in network router buffers. This behavior is particularly problematic over PPP links since some PPP servers have buffers that hold 15 seconds of data, or more. Also unlike TCP, SCP does not retransmit data lost in the network. Thus it avoids the associated unpredictability in latency and wasted bandwidth.

Further salient features of SCP include its support for rapid adaptation in the presence of drastic bandwidth fluctuations, such as those that occur when mobile computers migrate among different network types,[5] and its support for streaming-specific operations such as pausing.

This paper is organized as follows. Section 2 presents the design of SCP. Section 3 follows with an analysis of SCP's steady-state bandwidth sharing. Section 4 describes the implementation of SCP. Section 5 explains the experimental results. Section 6 then briefly discusses other congestion control schemes, and finally, Section 7 concludes the paper and discusses future work.

2. THE DESIGN OF SCP

A unicast streaming scenario with SCP involves a sender, which streams media packets over a network connection in real-time to a receiver. SCP policies are implemented at the sender side. Each packet contains a sequence number, and for each packet, SCP records the time at which it is sent, and initiates a separate timer*. The receiver acknowledges each packet it receives by returning an ACK containing the sequence number of the packet to the sender. Based on the reception of ACKs and expiration of timers, SCP monitors the available bandwidth, detects packet loss, and adjusts the size of its congestion window to control the flow and avoid network congestion. To achieve this task SCP maintains the following internal state variables and parameter estimators.

- *state* — The current state. SCP has several states, each corresponding to a specific network and session condition and associated flow and congestion control policy.
- W_l — The size of the congestion window (in number of packets).
- W — The number of outstanding packets sent but not acknowledged. When $W < W_l$, the congestion window is open, and more packets can be sent, otherwise it is closed and no packets can be sent.
- W_{ss} — The threshold of W_l for switching from the slow-start policy to the steady-state policy.
- $\hat{T}_{brtt}$ — An estimator of the base RTT (round trip time) – the RTT of a packet sent when the network is otherwise quiet.
- $\hat{T}_{rtt}$ — An estimator of the recent average RTT.
- $\hat{D}_{rtt}$ — An estimator of the standard deviation of the recent RTT.

*For clarity, we explain the design of SCP based on a timer per packet, but this aspect is optimized in the implementation.

State	Network and session condition	Congestion window adjustment policy
slowStart	Available bandwidth not discovered yet	SCP opens the congestion window exponentially by increasing the window size by one upon the receipt of each ACK.
steady	Available bandwidth being fully utilized	SCP maintains appropriate amount of buffering inside the network to gain sufficient throughput, avoid excessive buffering or buffer overflow, and trace the changes in available bandwidth.
congested	The network is congested.	SCP backs off multiplicatively by halving the window size. Persistent congestion results in exponential back-off.
paused	No outstanding packet in the network	When a new packet is sent, SCP shrinks the window size and invokes the slow-start policy.

Table 1. SCP states, network and session conditions, and flow and congestion control policies

- $\widehat{rto}$ — An estimator of the timer duration.
- $\hat{r}$ — An estimator of the receiving packet rate.

As long as the congestion window is open, the sender streams packets at a rate no more than $\frac{W_l}{\hat{T}_{brtt}}$, instead of sending them out in bursts, so as to improve the smoothness of the stream. When an ACK is received, or a timer expires, the congestion window size W_l is adjusted using a policy associated with the current state.

SCP adopts window-based policies similar to those of TCP for slow-start, and exponential back-off upon network congestion. It also estimates RTT in a way similar to TCP. Therefore, many parameters and state variables have counterparts in TCP. However, SCP differs from TCP in that it has a base RTT estimator $\hat{T}_{brtt}$ and a packet rate estimator $\hat{r}$. The use of these estimators to ensure smooth streaming and low latency is discussed in the following sections.

2.1. Overall architecture

SCP is based on the observation that excessive packets in the round-trip network connection accumulate in buffers at network routers and switches and lead to an increased RTT. As the accumulation of packets increases, these buffers overflow and packets are dropped. The goal of SCP is to ensure smooth streaming at a suitably high throughput, but without causing excessive buffering or congestion in the network. To achieve this goal, SPC monitors fluctuations in the available bandwidth and pushes an appropriate number of additional packets into the network connection. If network congestion is detected, SCP reacts immediately with exponential back-off. Depending on the condition of the network and the streaming session under control, SCP is in one of the following four states: *slowStart*, *steady*, *congested* or *paused*. Each state is associated with a specific condition and congestion window size adjustment policy as listed in Table 1.

SCP handles events that indicate changes in network and session conditions. Such events include indications that: the SCP session becomes paused or active; the available network bandwidth is fully utilized; the network is congested; the network interface has been switched. Upon these events, SCP updates its internal state and possibly switches to a new policy. Figure 1 shows the events that SCP handles and its associated state transitions.

2.2. Congestion window adjustment policies and state transitions

Initialization: Upon initialization, SCP sets $W = 0$, $W_l = 1$ and $W_{ss} = L_w$, where L_w is a limit on the congestion window size. $\hat{T}_{brtt}$, $\hat{T}_{rtt}$ and $\hat{D}_{rtt}$ are set to infinity. $\widehat{rto}$ is set to an initial default value. After initialization, SCP enters the *paused* state. Transmission of the first packet brings SCP to the *slowStart* state.

Slow-start: The slow-start policy is invoked after initialization or when SCP resumes from a pause. Its goal is to quickly grow the congestion window and discover the available network bandwidth. W_l is incremented by 1 when an ACK is received in-order, thus doubling the congestion window after each RTT amount of time. SCP leaves the *slowStart* state upon detecting events indicating network congestion, full utilization of available bandwidth, pause in streaming, or network interface switch.

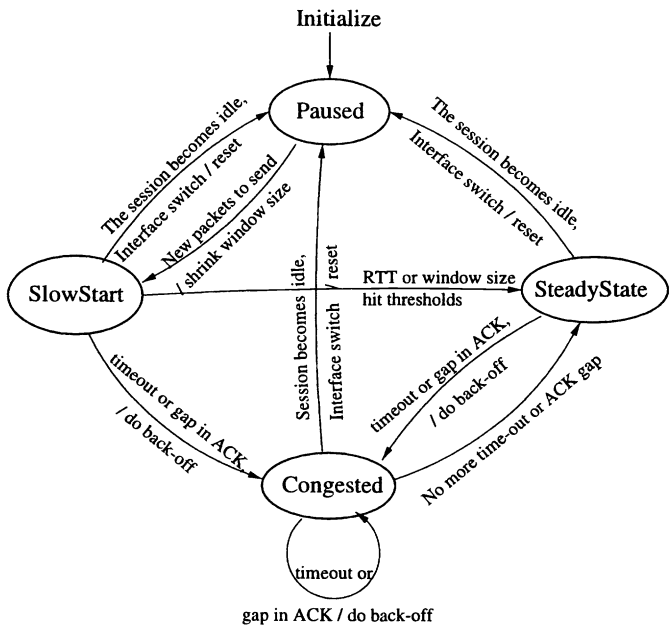


Figure 1. SCP state transition diagram

Steady-state smooth streaming: By pushing an appropriate number of extra packets, the steady-state policy enables sufficient utilization of the available bandwidth while avoiding over-buffering. It also traces the fluctuations in available bandwidth. In the *steady* state, estimation of packet rate $\hat{r}$ is enabled. Whenever a new $\hat{r}$ value is available, congestion window size W_l and threshold W_{ss} are adjusted according to Equation 1 below, where W_Δ is a constant referred to as the window-size incremental coefficient.

$$W_l = W_{ss} = \hat{r}\hat{T}_{brtt} + W_\Delta \tag{1}$$

The idea behind the flow control defined by Equation 1 is that SCP assumes that $\hat{r}$ is an approximation of the network bandwidth (in terms of packet rate) available to the session, calculates the bandwidth-delay product of the network connection with minimum buffering $\hat{r} \times \hat{T}_{brtt}$, and adjusts its congestion window size W_l accordingly. The product $\hat{r} \times \hat{T}_{brtt}$ is the amount of data SCP should keep outstanding inside the network in order to maintain sufficient throughput with minimum buffering. Since the network is shared and highly dynamic, the bandwidth available to the session under control fluctuates. If SCP just sets its congestion window size to $\hat{r} \times \hat{T}_{brtt}$, when the available bandwidth decreases, $\hat{r}$ would decrease, and SCP would reduce its congestion window size and thus the amount of outstanding data. However, if the available bandwidth increases, there is no way for SCP to detect that. This is because the receiving packet rate $\hat{r}$ could not be increased unless SCP increases the sending packet rate first, but SCP would not increase its sending packet rate by increasing the congestion window size unless it observes a increased $\hat{r}$. To solve this "chicken-and-egg" type of problem, SCP pushes W_Δ amount of extra outstanding data, which are held in router and switch buffers. When the network available bandwidth increases, releasing the extra data buffered in the network results in an increase in packet rate $\hat{r}$, which in turn results in larger amount of outstanding data as defined by the increased W_l. It can be seen that the increase in W_l is no more than additive, which is what TCP does in its steady state. Due to the smoothing effect of lowpass filtering in estimation of $\hat{r}$, W_l will be changed gradually, causing smooth changes in throughput. Also, with the flow control policy stated in Equation 1, though at the beginning $\hat{r}$ may not be a reliable estimation of the actual available bandwidth, if the network is stable enough, SCP will eventually bring $\hat{r}$ to the actual bandwidth through iterations. Overall, the steady-state policy is able to converge to a stable throughput, and to trace the fluctuation in available bandwidth smoothly.

Upon detecting network congestion, SCP backs off and enters the *congested* state. SCP enters the *paused* state when the session is paused, or when a switch in network interface is detected.

The steady-state policy defined by Equation 1 is what makes SCP different from TCP. If network conditions remain the same for a sufficiently long period of time, SCP converges its window size and data rate to stable values. SCP is also able to limit the buildup of end-to-end latency resulted from buffering in routers, regardless of the actual sizes of router buffers. These properties are desirable for media streaming applications. In contrast, TCP repeatedly increases its congestion window size linearly until packets are lost, and then backs off by halving the window size. The result is that TCP traffic is inherently bursty. TCP also tends to fill up router buffers no matter how big they are. A consequence is that across network connections with deep buffers, TCP yields long latencies. The deeper the buffers are, the longer the latencies are.

Exponential back-off upon network congestion: Upon network congestion, SCP invokes an exponential back-off policy similar to that of TCP. Whenever events such as gaps in ACK or timer expirations are detected, SCP backs off multiplicatively by reducing its congestion window size in half. The data rate estimator $\hat{r}$ is no longer accurate and thus is reset and disabled. If the back-off is triggered by timeout, $\widehat{rto}$ may have been too short and is doubled. When the network is so congested that no packet can get through, with the multiplicative decrease in congestion window size and multiplicative increase in timer duration, SCP backs off exponentially until it virtually stops sending any further packet. This gives a chance for the network to quickly recover from the congestion.

At the time of a back-off, there may already be some outstanding packets inside the network. Loss of these packets does not correctly reflect the network condition after the back-off, and thus should be ignored. The *congested* state is designed for this purpose. After each back-off, SCP enters the *congested* state. Further back-off is disabled until the first packet sent in the current *congested* state is acknowledged, found lost or its timer expired. If the first packet is acknowledged, SCP enters the *steady* state, otherwise another round of back-off is initiated.

Pause when there is no packet to send: In a real-time streaming session, a user may want to pause in the middle. If the sender has no data to send for a while, W eventually decreases to 0. At this moment, the streaming session becomes idle, and SCP enters the *paused* state.

When an SCP session is paused, the bandwidth it previously took will gradually be discovered and taken by other sessions. When the session resumes at a later time, SCP invokes the slow-start policy with a reduced initial congestion window size. Currently, we adopt a ad-hoc policy of halving the congestion window size in every $\hat{T}_{brtt}$ time elapsed in the *paused* state.

Reset upon network interface switch: When either end of an SCP streaming session has its network interface switched (e.g., from Ethernet to cellular modem, or vice versa), the route from the sender to the receiver may be changed, and the new connection may go through links with totally different capacities. SCP handles this dynamic reconfiguration issue by providing a *reset* interface. Upon each network interface switch event, SCP discards the out-dated estimators, ignores all outstanding ACKs, and resumes with the slow-start policy to quickly discover the capacity of the new connection.

3. ANALYSIS OF BANDWIDTH SHARING BETWEEN SCP SESSIONS

With the steady-state rate- and window-based policy defined by Equation 1, it is possible for multiple SCP sessions to share network links in a fair and stable manner. In this section, we analyze a simple case, in which two sessions with the same packet size share a single network link. Both sessions send packets at maximum rate while the congestion window is open.

Suppose that, in a steady state, session A has estimators $\hat{r}_a$, $\hat{T}_{brtta}$, $\hat{T}_{rtta}$, and $W_{la} = \hat{r}_a\hat{T}_{brtta} + W_\Delta$; and session B has $\hat{r}_b$, $\hat{T}_{brttb}$, $\hat{T}_{rttb}$, and $W_{lb} = \hat{r}_b\hat{T}_{brttb} + W_\Delta$. Since A and B share the same network link, we have following observations:

(1) The aggregate packet rate of the network link, $\hat{r}_l$, is the sum of the two sessions: $\hat{r}_l = \hat{r}_a + \hat{r}_b$.

(2) Base RTT estimate is the actual link base RTT T_{brttl} plus some estimation error: $\hat{T}_{brtta} = T_{brttl} + e_a$ and $\hat{T}_{brttb} = T_{brttl} + e_b$. When $\hat{T}_{brtt}$ is estimated as the minimum of the past RTT measurements of a session, the main component of the estimation error is the residual buffering, which is caused by packets of other sessions preventing the network buffers from becoming empty. Another less important component is sampling noise.

(3) Sessions A and B have the same RTT estimator (when the sampling noise can be ignored): $\hat{T}_{rtta} = \hat{T}_{rttb} = \hat{T}_{rttl}$. Furthermore, the number of packets sent by a session in one RTT equals its congestion window size. Thus the packet rate ratio of A and B equals the ratio of their window size:

$$\frac{\hat{r}_a}{\hat{r}_b} = \frac{W_{la}}{W_{lb}} = \frac{\hat{r}_a(T_{brttl} + e_a) + W_\Delta}{\hat{r}_b(T_{brttl} + e_b) + W_\Delta} \quad \Longrightarrow \quad \hat{r}_a = \frac{\hat{r}_b W_\Delta}{\hat{r}_b(e_b - e_a) + W_\Delta} \tag{2}$$

Combining observations (1) and (3), it is clear that there is a single solution for $\hat{r}_a$ and $\hat{r}_b$. The session with a larger residual buffering tends to have a larger estimation error, thus gets a larger portion of the link bandwidth. In a special case where the two sessions have the same RTT estimation error, $e_a = e_b$, we have $\hat{r}_a = \hat{r}_b = \frac{\hat{r}_l}{2}$, indicating that the two sessions split the link bandwidth evenly.

4. IMPLEMENTATION OF SCP

SCP has been implemented as a layer on top of UDP, based on a toolkit for developing adaptive systems.[6] In this section, we briefly discuss the parameter estimators and various implementation issues. More details may be found in Cen's thesis.[6]

Several parameters are estimated by SCP. Base RTT $\hat{T}_{brtt}$, average RTT $\hat{T}_{rtt}$, RTT standard deviation $\hat{D}_{rtt}$, and timer duration $\widehat{rto}$ are estimated based on the history of the raw RTT measurements. $\hat{T}_{brtt}$ is estimated as the minimum of all the past RTT measurements. The estimations $\hat{T}_{rtt}$, $\hat{D}_{rtt}$ and $\widehat{rto}$ are done in a way similar to those in TCP.[4] The receiving packet rate $\hat{r}$ is estimated as the rate at which ACKs are received. This is reasonable since the receiver acknowledges every packet it receives.

Several performance issues have been addressed in the implementation. The first is that, in the design of SCP discussed in Section 2, each outstanding packet has a separate timer. The overhead of maintaining these timers would make SCP non-scalable. In the implementation, only an ordered table is maintained to record the sending time of all outstanding packets. Upon receipt of an ACK, SCP removes the entry of the acknowledged packet from the table. Only when there is a new packet to send and the congestion window is closed, SCP checks the older packets in the table to see if their timers should have been expired, and performs back-off if necessary. Other issues, such as out-of-order packet delivering, congestion-window adjustment in low-data-rate sessions, etc., are also addressed.

5. EXPERIMENTS AND RESULTS

To evaluate the performance of SCP, a test program has been built to stream dummy packets between Internet hosts using SCP or TCP. The test program reads parameters and experimental scripts from files, and collects statistics for analysis.

The network configuration for the experiments is shown in Figure 2. We have two LANs, one at Oregon Graduate Institute (OGI) and the other at Georgia Institute of Technology (Georgia Tech) connected through the long-haul Internet with 29 hops. Among the hosts, *anquetil* is a Linux PC notebook on subnet 1 of the OGI LAN. This notebook can optionally be moved to subnet 2 or a $28.8kb/s$ PPP link. This configuration covers typical types of Internet connections such as phone lines, LAN, and WAN. Network interface switching in mobile environments is simulated by having *anquetil* to switch between its Ethernet and PPP interfaces dynamically.

Experiments have been performed to evaluate the performance of SCP in sessions over different network connections. For all the experiments, unless stated explicitly, the following parameter values are used: steady-state congestion window size incremental coefficient $W_\Delta = 3$, and data packet size $1472B$†. The size

†IP packet size $= 20 + 8 + 1472 = 1500B$ is the Ethernet MTU. $1472B$ is the UDP/TCP payload size, including the SCP header (in the case of SCP), the various fields carried in the packet, followed by a block of randomized data.

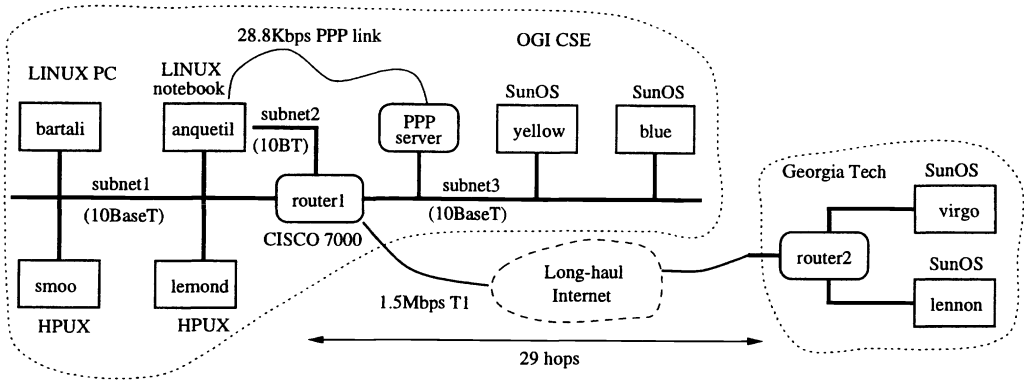
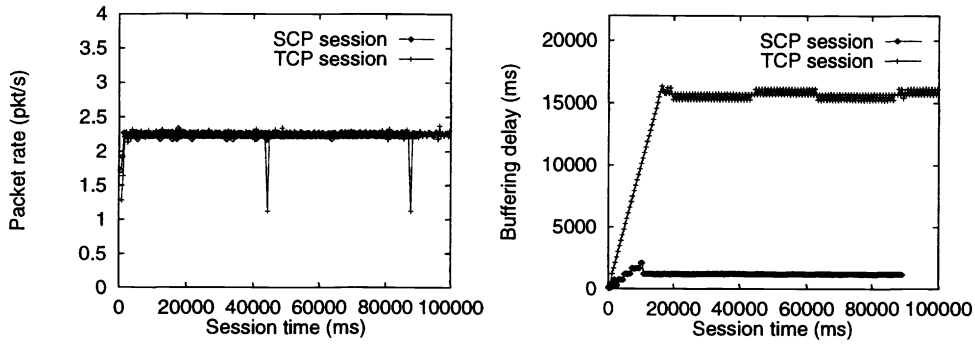


Figure 2. Network configuration for experiments



(a) Packet rate

(b) Buffering delay

Figure 3. Performance of single SCP and TCP sessions from *lemond* to *anquetil* over PPP

of TCP and SCP socket sending and receiving buffers is up to $64KB$, which is the maximum supported in many Berkeley socket implementations. The value of W_Δ is selected in an ad-hoc manner, but experiments have shown that it yield satisfactory performance.

In every SCP or TCP session, the sender uses non-blocking mode to send packets. It repeatedly waits until the socket is ready to send (in the case of TCP) or the congestion window is open (in the case of SCP), timestamps a dummy packet, and sends it out immediately. Each packet carries a sender timestamp. The receiver records the time each packet is received. With the receiving timestamps, intervals between packets and packet rate can be measured. Assuming that the clocks of the sender and the receiver have the same rate, with the sender and receiver timestamps, it is possible to measure the application-level end-to-end *buffering delay* — the delay from when a packet leaves the sender application until when it is received by the receiver application, minus the minimum of all the delay measurements, which is the transmission and processing delay. For each packet, the buffering delay is measured as the difference between the sender and receiver timestamps, minus the minimum of all the measurements in the history. This end-to-end buffering delay include the times spent in the buffers in the sender, the receiver, and routers and switches. For all TCP and SCP sessions, buffering delay and receiving *packet rate* are measured. For SCP sessions, the congestion window size, and the gaps in ACKs are also measured.

5.1. Experiments over PPP link

In the first set of experiments, packets are streamed from *lemond* to *anquetil* over the $28.8kb/s$ PPP link with a MTU of $1500B$.

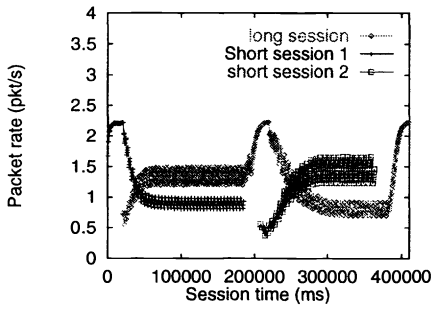


Figure 4. Smoothed packet rate of two SCP sessions from *bartali* to *anquetil* over PPP link

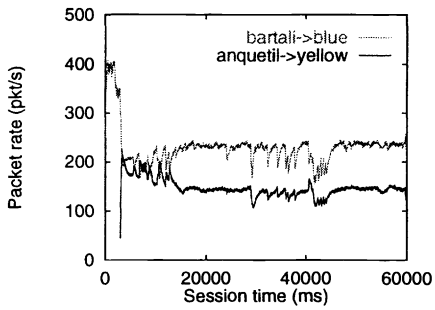


Figure 5. Smoothed packet rate of two SCP sessions through single router

First, single TCP and SCP sessions are played. For the SCP sessions, the receiving data rate is about $2.24pkt/s \approx 26.9kb/s$, which is close to the $28.8kb/s$ PPP link speed. Figure 3(a) shows the packet rate of a SCP session and a TCP session over time. It shows the raw packet rate (inverse of packet interval) versus the session time (the time relative to the beginning of the session). Figure 3(b) shows the buffering delay of these two sessions. After a startup phase, the SCP session yields a smooth packet rate, and utilizes the bandwidth sufficiently. The TCP session has similar data rate, except for the periodic downward spikes which are caused by packet loss (by the PPP server). However, SCP and TCP sessions maintain different levels of buffering inside the network. The SCP session has a steady-state buffering delay of about 1.2 seconds. In contrast, the TCP session pushes the buffering delay up to 16 seconds and stays there. As a matter of fact, different TCP implementations have very different behaviors over PPP. SunOS TCP has a steady-state behavior similar to that of HP-UX TCP, but a much worse start-up phase. Linux TCP pushes the delay to 7 seconds and stays there. Long network buffering delay for TCP over PPP is extremely detrimental to interactive streaming traffic. In an interactive streaming video presentation, each time the user changes playback speed, it is simply not tolerable to see no effect until after more than 16 seconds.

Careful readers may have noticed that 16 seconds is roughly the time for transferring $64KB$ (sender buffer size) of data over the $28.8kb/s$ PPP link. But this long delay is actually caused by a combination of PPP link's low bandwidth and the relatively large buffer in PPP server for individual PPP links. In our experiments, when UDP packets are sent from *lemond* or other hosts to *anquetil* over the PPP link, more than 15 seconds of data will be buffered in the PPP server before packets are dropped. A TCP connection over the PPP link will have its congestion window increased until either the sender or receiver socket buffer is full, or when packets are dropped when the buffer in PPP server overflows. To see that the long buffering delay is not caused by the big sender buffer itself, we also measured several TCP sessions in the reverse direction, from *anquetil* to *lemond*, and got buffering delays of about 2.5 seconds. It seems that LINUX has incorporate some techniques to reduce the latency for TCP over PPP to 7 seconds. Of cause, buffering delay can be reduced by reducing the TCP sender socket buffer size. But small sender socket buffer size is not always desirable. It will limit the size of congestion window, and will not be able to explore the bandwidth of long and fat network connections. There is not a single socket buffer size which fits all types of connections.

Figure 4 shows the smoothed[‡] packet rate of three SCP sessions, when two of them are played simultaneously from *bartali* to *anquetil* over the PPP link. It can be seen that the two competing sessions eventually reach a stable share of the PPP bandwidth. This experiment also shows the impact of residual buffering error in the based RTT estimation on bandwidth sharing. The base RTT estimation of the first half of the long session has a residual buffering caused by the first short session, thus it gets a larger share of the bandwidth. This result supports the observation in the bandwidth sharing analysis (Section 3). Later, after the first short session ends and the second short session starts, the latter gets a residual buffering error caused by the long session, and also gets a bigger portion of the PPP bandwidth. Further experiments show that if the start times of two competing SCP sessions are close enough, they will split the bandwidth evenly.

[‡]Smoothing is achieved by applying a lowpass filter to the raw packet rate measurements.

Experiment configuration	Single SCP *anquetil→yellow*	Single TCP *anquetil→yellow*	SCP: *anquetil→yellow* SCP: *bartali→blue*	SCP: *anquetil→yellow* TCP: *bartali→blue*
Packet rate (pkt/s)	320	380	150 : 230	SCP 100, TCP 300
Buffering delay (ms)	0∼30 around 5	5∼80 around 40	0∼40 around 14	SCP 0∼50, TCP 10∼80

Table 2. Results of single TCP and SCP sessions through a single router

Experiments have also been carried out to play one SCP session and one TCP session simultaneously over the PPP link. Since SCP limits the amount of network buffering, the bandwidth partition depends on how aggressive a specific TCP implementation is. For sessions from *bartali* to *anquetil* the packet rates are $0.6pkt/s$ for SCP and $1.7pkt$ for TCP. HP-UX and SunOS TCP implementations are more aggressive. When SCP and TCP sessions are played from *lemond* to *anquetil*, virtually all bandwidth is taken by the TCP session.

5.2. Experiments on single subnet

All the single SCP sessions played from *bartali* to *anquetil* on subnet 1 yield smooth and stable throughput of about $700pkt/s = 8.4Mb/s$. The buffering delays of these sessions, primarily caused by MAC-level collisions and back-offs, remain within $4ms$. For comparison, individual TCP sessions yield $670 \sim 680pkt/s$ and up to $40ms$ of buffering delay. This indicates that SCP is efficient, smooth, and has low latency.

When two SCP sessions, one SCP and one TCP, or two TCP sessions are played *bartali→smoo* and *anquetil→lemond* on subnet 1, the two SCP and TCP sessions are able to share the Ethernet, but they are all very jerky, with highly variable throughput ($100pkt/s$ to $700pkt/s$) and buffering delay (up to 0.8 second). SCP also drops packets. This jerky throughput and packet drop is the result of severe MAC-level collisions and back-off which can not be detected by the sender application. Not much can be done by SCP or TCP to eliminate this problem except for explicitly limiting the data rate at the sender side.

5.3. Experiments through single router

To evaluate the performance of SCP streaming through a single router, *anquetil* is moved to subnet 2. Table 2 shows the configurations and the overall performance of the experiments. Figure 5 shows the smoothed packet rates of two simultaneous SCP sessions. These experiments were run late in the evening when the network was lightly-loaded.

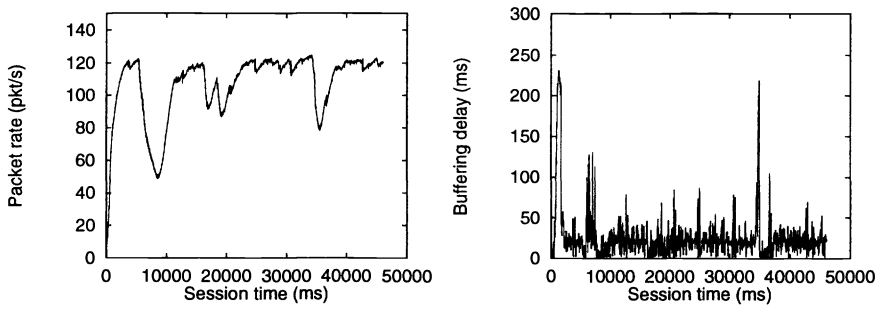
A single SCP session yields a lower throughput than a single TCP session, but has a significantly lower buffering delay. Possibly reasons for SCP to have a lower bandwidth are that SCP sends more ACKs (one per packet), and maintains lower level of buffering thus the router buffers may be empty for more time.

Figure 5 shows that when two SCP sessions are competing for the output port of the single router, they are able to share the bandwidth in a stable manner. To understand why there is a packet rate difference between the two sessions, a closer look at the statistical data reveals that the two sessions have the same steady-state congestion window size of 5, but they have a different RTT. When played separately, the RTT for sessions *bartali→blue* and *anquetil→yellow* are around $15ms$ and $22ms$ respectively. When played simultaneously, they are around $22ms$ and $35ms$ respectively. The difference in RTT may be caused by the differences in host CPU speeds and other factors, and results in an uneven bandwidth partitioning. This doesn't mean that SCP itself is unfair.

When an SCP session competes against a TCP session, they are able the share the bandwidth. But SCP's steady-state control of network buffering make it less aggressive than TCP and gets a significantly smaller amount (1/4) of the bandwidth.

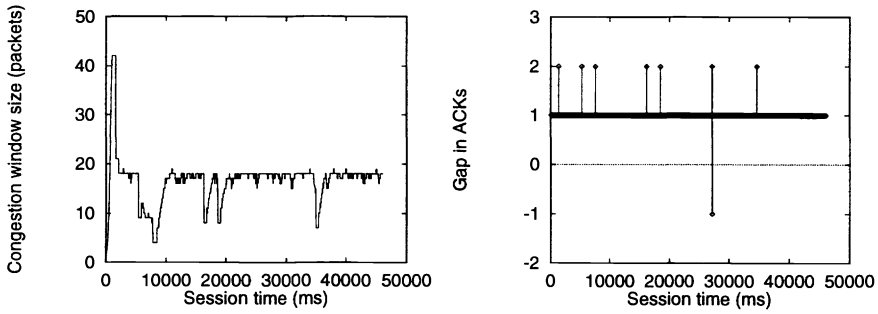
5.4. Experiments across the Internet

To evaluate the performance of SCP across the long-haul Internet, SCP and TCP sessions are played between hosts at OGI and Georgia Tech. There is no way to control the traffic in the Internet. Nevertheless, there are still times, such as midnights and weekends, when the Internet is relatively lightly loaded, and times such as weekdays when it is heavily loaded.



(a) Smoothed packet rate

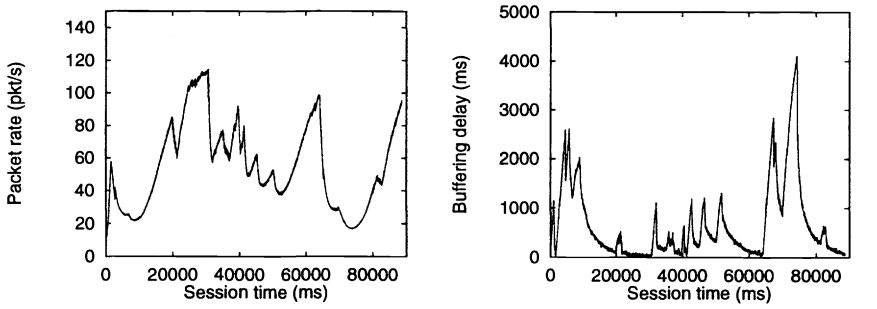
(b) Buffering delay



(c) Congestion window size

(d) Gap in ACKs

Figure 6. Performance of a single SCP session across the lightly loaded Internet



(a) Smoothed packet rate

(b) Buffering delay

Figure 7. Performance of a TCP session across the lightly loaded Internet

As shown in Figure 6, when the network is lightly loaded, SCP is able to stream smoothly with low buffering delay, and to explore the network bandwidth sufficiently. From time to time, there are still periods of network congestion and packet loss. Figure 6(c) and (d) identify times when packets are lost, causing SCP to back off. The negative ACK gap shown in Figure 6(d) indicates an ACK delivered out of order. In this case, W_l is halved and then restored before any further packet is sent, thus the back-off is not shown in Figure 6(c). When packets are not dropped, SCP's steady-state rate- and window-based flow control policy maintains a stable congestion window size of about 18 packets, a stable throughput of about $115pkt/s(\approx 1.4Mb/s$, close

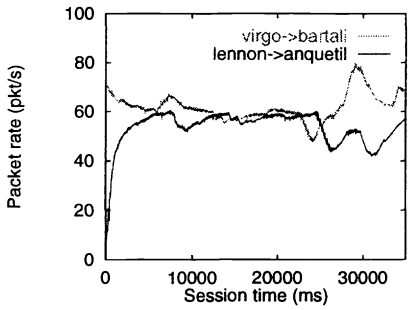


Figure 8. Smoothed packet rate of two SCP sessions across the Internet

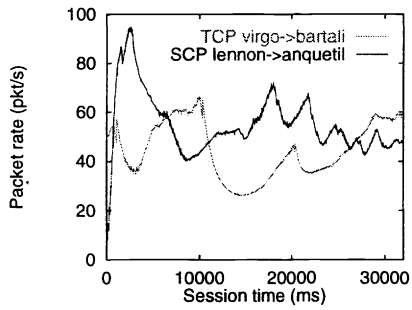


Figure 9. Smoothed packet rate of SCP and TCP sessions across the Internet

to the $1.5Mb/s$ T1 line speed), and low buffering delay (mostly lower than $50ms$). On the other hand, as shown in Figure 7, a single TCP session has more erratic throughput plus long and unpredictable buffering delays (up to 4 seconds)[§].

Two simultaneous SCP sessions are able to share the bandwidth in a fair manner. In our experiment, two competing SCP sessions: *lennon→anquetil* and *virgo→bartali* are played across the lightly loaded Internet. The ratio of the average bandwidth goes from $70pkt/s : 40pkt/s$ to $60pkt/s : 60pkt/s$. The buffering delay for the sessions are kept mostly below $100ms$. Figure 8 shows the packet rate of two simultaneous SCP sessions. Random packet drop causes competing sessions to back-off randomly, but on average the bandwidth sharing is fair, and the throughput is relatively smooth.

SCP and TCP sessions across the long-haul Internet share the network bandwidth more evenly than when through a single router. Figure 9 shows the packet rates of two simultaneous SCP and TCP sessions. The two competing sessions, SCP: *lennon→anquetil* and TCP: *virgo→bartali*, produce a throughput ratio around $SCP50pkt/s : TCP40pkt/s$. While TCP sessions have long latency, the buffering delay of the SCP sessions are still below $100ms$ for most packets.

During busy weekdays, it is more obvious that SCP yields smoother throughput and lower and more consistent delay than TCP, while still operating in harmony with the latter. The performance results of two SCP and TCP sessions from *virgo* and *bartali* are shown in Figure 10. The network is so congested that even "`ping virgo`" from *bartali* shows up to 10% packet loss. SCP sessions consistently get about $14pkt/s$ with buffering delays[¶] below $100ms$. SCP sessions observed up to 10% loss in ACKs. Due to the heavy packet loss, TCP sessions get only 2 to $6pkt/s$, with buffering delays up to 70 seconds[‖], and the throughput is very jerky. It is true that 10% packet loss will reduce the video presentation quality significantly, but a 70 second delay is simply intolerable to any streaming application!

5.5. Experiments with network interface switch

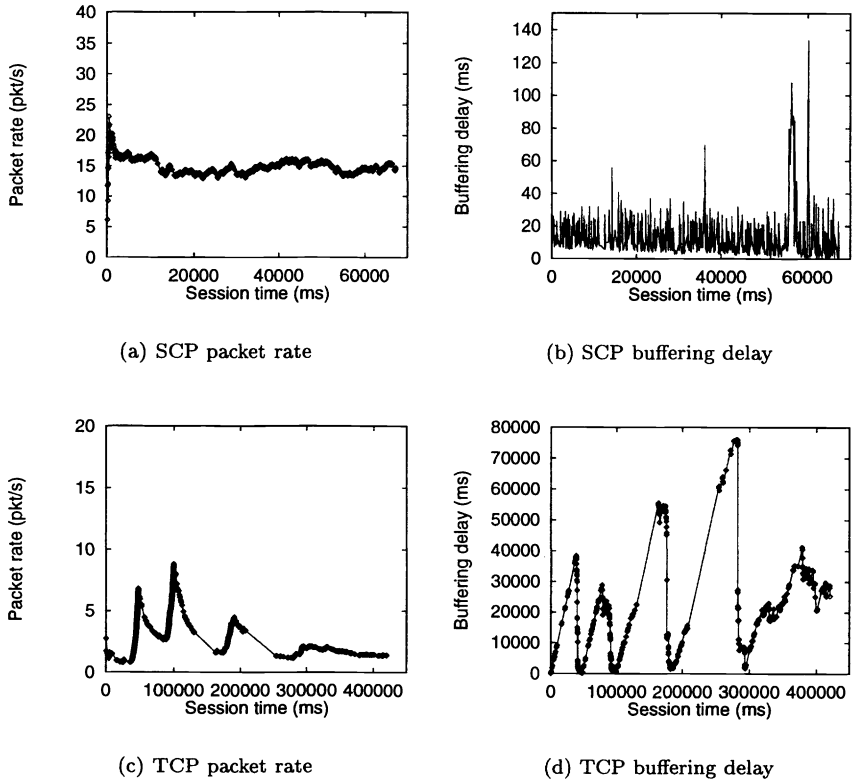
To evaluate SCP's ability to adapt to different network capacities resulted from network interface switches, both the PPP and Ethernet interfaces of the notebook *anquetil* are activated, and SCP sessions are played from *bartali* to *anquetil* through either the PPP or the Ethernet interface. Switching between PPP and Ethernet is simulated by reconfiguring the default IP route and sending a HUP signal to the receiver of the on-going SCP session. When the receiver gets a HUP signal, it re-establishes the control and data connections to the sender, optionally resets SCP, and continuous the session from where interrupted.

In our experiment, SCP is able to adapt to network change quickly and utilize the available bandwidth right away, with or without being reset. The reason is that the same congestion window size, around 5 packets, is optimal for both the PPP and the Ethernet link. If the network switch is between a PPP link and

[§]For TCP sessions, the end-end buffering delay measurement includes the time for back-off and retransmission of lost packets.

[¶]In a busy network when the buffers in routers are never empty, buffering delay measurements actually reflect the fluctuations in buffering delay.

[‖]In the presence of severe packet loss, most of the buffering delay measurement is the result of repeated back-off and retransmission of lost packets.



(a) SCP packet rate

(b) SCP buffering delay

(c) TCP packet rate

(d) TCP buffering delay

Figure 10. Performance of single SCP and TCP sessions across the busy Internet

a WAN connection with a large optimal congestion window size, we expect that reset on SCP would make a big difference. The slow-start policy following the reset helps figure out the new window size quickly.

6. RELATED WORK

Various schemes have been proposed for flow and congestion control in the context of streaming real-time multimedia data across the Internet. These schemes include HTTP/TCP, TCP without retransmission, rate-based flow and congestion control, and router-based congestion control. Most of these schemes address congestion control for unicast streaming. For multicast streaming there are also multilayered schemes.

HTTP/TCP based schemes: HTTP[7]/TCP[4] has been used in virtually all commercial streaming media players. The Internet has proven TCP's robustness and ability to share bandwidth among multiple sessions. Also important to commercial streaming players is HTTP's ability to penetrate firewalls. However, TCP is designed for reliable distribution of non-real-time data. When used for media streaming, it has the drawbacks of long and unpredictable latency, wasted network bandwidth, and bursty data flow. TCP implements reliability through infinite retransmission, causing unpredictable and unbounded latency and wasted bandwidth. TCP data flow is inherently bursty, due to its repeated process of increasing its congestion window size until packets are lost, then backing off exponentially. In network connections with deep buffers, such as PPP links, TCP's greediness in filling up the buffers incurs long latencies that are not tolerable in interactive streaming applications. SCP addresses the problems in TCP by not doing retransmission, taking increases in buffering latency as an early indication of network congestion, introducing a smooth steady-state policy, and doing packet pacing at the sender.

The problems of long latency caused by deep router buffers in (traditional) TCP has been addressed in TCP Vegas.[8] Vegas observes that in congested networks, router buffers will fill, causing increased latency, before they eventually overflow. A mechanism is used to measure the increase in latency and adjust the congestion window size accordingly. SCP is based on similar observations, but it uses a different flow control policy. The policy has been shown to ensure smooth streaming, low latency, and stable and fair sharing between multiple streams.

A congestion control protocol based on TCP, but without the retransmission is used in Jacobs' Internet video applications.[9] Removing reliability from TCP eliminates the resultant problems of unpredictability latency and wasted network bandwidth. However, other problems of TCP, including bursty data flow, and long latency in connections with deep buffers, remain the same. To reduce the impact of bursty data flow on video quality, Jacobs uses a buffer at the sender side, which introduces more latency. Instead, SCP tries to smooth the streaming itself. It directly paces out the sending of packets, and it employs a flow control policy with a steady state in which smooth streaming are maintained.

Rate-based schemes: Rate-based feedback is widely used for flow and congestion control in streaming applications. Examples are Smith's cyclic-UDP,[10] our first distributed MPEG player,[2] and the RTP[11] rate-based flow control proposed by Busse and Schulzrinne.[12] In a typical rate-based feedback scheme, the receiver continuously monitors the metrics of the incoming stream, such as the received data rate or data loss ratio, and sends feedback messages to the sender. The sender then adjusts the data rate accordingly. The goal is to maintain the stream quality metrics such as packet loss ratio at a controlled level. While rate-based feedback works well most of the time, it has the inherent danger of overflowing the network buffers, since the metric being directly controlled is the data rate, instead of the amount of buffering inside the network as in TCP. In the case of severe congestion, receiver-initiated negative feedback (the sender only reduces data rate when asked by the receiver) may not be able to react quickly enough, both because data rate or packet drop rate estimations are state-based and involve sluggish lowpass filtering, and feedback messages (to reduce the data rate) may not be able to get through to the sender in time.

Several sender-initiated rate-based schemes have also been proposed. In Keshav's control-theoretic packet-pair scheme,[13] packet-pairs are used by the sender to measure the available bandwidth and a control-theory-based policy is used for flow control. This scheme is based on the assumption that the routers/switches use round-robin-like queuing (fair queuing), but most, if not all, Internet routers/switches uses a totally different first-come-first-serve queuing.

A TCP-friendly unicast rate-based flow control scheme has been proposed by Mahdavi and Floyd.[14] By having the sender (instead of the receiver) monitor the overall packet loss ratio based on the acknowledgments from the receiver, this scheme solves the problem of feedback messages not getting through in time. However, the other problems such as sluggishness in estimation of packet loss rate, and the danger of network buffer overflow still exist. The scheme uses packet loss thresholds to detect congestion, and adopts a congestion avoidance policy similar to that of TCP. We believe that whether satisfactory thresholds exist, or if they can feasibly be computed, and how the scheme actually interacts with TCP, is yet to be seen. In contrast, SCP uses the same congestion avoidance policy as TCP. Only when there is no congestion, SCP's hybrid rate- and window-base policy will bring a stream to a steady state with a stable packet rate.

Router-base schemes: SCP as well as all the above-discussed schemes are end-host based. They are implemented exclusively on the sender or receiver sides. There is another category of schemes which are implemented on routers, such as the schemes proposed by Jeffay,[15] and by Floyd.[16] While router-based schemes may be more effective than host-based ones, they need to be implemented in routers. This requirement greatly slows down the speed at which the scheme can be deployed. Consequently, very few router-based flow or congestion control mechanisms can be found in the current Internet infrastructure.

Congestion control for multicast streaming: Our work focuses on unicast streaming, but some other researchers are examining rate adaptation policies for best-effort, real-time *multicast* streams. McCanne's receiver-driven layered multicast (RLM) combines layered content encoding with a layered transmission

system.[17] Layered content is multicasted using multiple multicast groups, and receivers are responsible for adapting to network bandwidth variations by dynamically joining and leaving multicast groups.

7. CONCLUSIONS AND FUTURE WORK

We have presented SCP, an effective flow and congestion control scheme for Internet media streaming applications. SCP uses the same congestion avoidance mechanism as TCP. However, when the network is not congested, SCP's hybrid rate- and window-based flow control policy ensures stable, smooth streaming, and low latency. To adapt quickly to drastic changes in network bandwidth following a pause in the stream, or a switch of network interfaces, the state of SCP can be reset to begin a slow-start policy that uses a reduced initial congestion window size. Finally, SCP does not retransmit lost packets and hence eliminates that source of unpredictability in latency.

The experiments described in this paper test SCP over typical network configurations, including PPP, single subnet, single router (LAN), and across the long-haul Internet (WAN). The experiments demonstrate that SCP is able to stream data packets smoothly in all the cases. The buffering delay of SCP streams, even over PPP links with deep buffers, remains suitably low. Multiple SCP streams are able to share the network bandwidth in a stable, fair manner. Although our experiments show that residual errors in base RTT measurement can have a significant impact on network bandwidth partitioning among SCP sessions. Ensuring fair bandwidth partitioning without direct cooperation from network routers and switches, while not impossible, is not easy.

Our experiments also showed that SCP streams are able to share network bandwidth with TCP streams. SCP seems somewhat less aggressive than TCP, especially in the case of the PPP experiment. This behavior is a result of SCP's steady-state policy that limits packet buildup in router buffers, unlike TCP which tends to fill those buffers. This observation raises the question of whether TCP is too aggressive or SCP is too conservative. Filling up deep buffers in PPP servers merely results in more latency, but does not help to increase throughput. In interactive streaming applications, multiple seconds of latency is intolerable, and hence TCP's behavior over PPP links is clearly detrimental.

The properties described above make SCP a promising protocol for use with best-effort multimedia streaming across the Internet. We have incorporated SCP into our adaptive streaming video player.[6] Based on the available network bandwidth discovered by SCP, the player adapts the video presentation quality in multiple dimensions, including spatial resolution, frame rate, end-end latency, etc.

In the future, we plan to fine tune SCP's parameters and policies, study its execution overhead, optimize the number of ACK packets it generates, and evaluate SCP in more complex network scenarios. We also plan to compare SCP with other congestion control schemes - especially those used in commercial streaming media players - and further study, through analysis and experiments, the interaction between multiple SCP sessions and other congestion control schemes.

ACKNOWLEDGEMENTS

Discussions with the Quasar group at OGI motivated this work. Jon Inouye, Dylan McNamee and Ashvin Goel provided thoughtful comments on early drafts of this paper. Special thanks to Jon Inouye for his insights into TCP and other network protocols, and for discussions on the ideas behind this work. We are grateful to Prof. Karsten Schwan (Georgia Tech) for providing the account used to run the Internet experiments. This project was supported in part by DARPA grants N00014-94-1-0845, N66001-97-C-8522 and N66001-97-C-8523, and by Tektronix, Inc. and Intel Corporation.

REFERENCES

1. L. A. Rowe, K. D. Patel, B. C. Smith, and K. Liu, "MPEG video in software: Representation, transmission and playback," in *Symposium on Electronic Imaging Science and Technology*, pp. 15–26, (San Jose, California), February 1994.
2. S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole, "A distributed real-time MPEG video audio player," *NOSSDAV'95, Lecture Notes in Computer Science* **1018**, pp. 151–162. Springer-Verlag, 1995.

3. S. McCanne and V. Jacobson, "*vic:* a Flexible Framework for Packet Video," in *Proceedings of the Third ACM Conference on Multimedia*, pp. 511–522, (San Francisco, California), November 1995.
4. V. Jacobson, "Congestion avoidance and control," in *SIGCOMM'88*, pp. 79–88, August 1988.
5. J. Inouye, S. Cen, C. Pu, and J. Walpole, "System support for mobile multimedia applications," in *NOSSDAV'97*, pp. 143–154, May 19-21 1997.
6. S. Cen, *A Software Feedback Toolkit and Its Applications in Adaptive Multimedia Systems.* PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, October 1997.
7. T. Berners-Lee, R. Fielding, and H. Nielsen, "Hypertext transfer protocol – HTTP/1.0." Internet RFC 1945, `http://ds.internic.net/ds/rfc-index.html`, May 1996, [September 11, 1997].
8. L. S. Brakmo *et al.*, "TCP Vegas: New techniques for congestion detection and avoidance," in *SIGCOMM'94*, pp. 24–35, August 1994.
9. S. Jacobs and A. Eleftheriadis, "Adaptive video applications for non-QoS networks," in *International Workshop on Quality of Service'97*, pp. 161–166, (Columbia University, New York), May 1997.
10. B. C. Smith, "Cyclic-UDP: a priority-driven best-effort protocol." Computer Science Department, Cornell University, `http://www2.cs.cornell.edu/Zeno/papers/cyclicudp.pdf`, May 1994, [October 20, 1997].
11. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications." Internet RFC 1889, `http://ds.internic.net/ds/rfc-index.html`, January 1996, [September 11, 1997].
12. I. Busse, B. Deffner, and H. Schulzrinne, "Dynamic QoS control of multimedia applications based on RTP," *Computer Communications,* **19**, pp. 49–58, January 1996.
13. S. Keshav, "A control-theoretic approach to flow control," in *Proceedings of SIGCOMM'91*, pp. 3–16, (Zurich, Switzerland), Sept. 1991.
14. J. Mahdavi and S. Floyd, "TCP-friendly unicast rate-based flow control." In end2end mailing list, `ftp://ftp.isi.edu/end2end`, January 1997, [September 11, 1997].
15. K. Jeffay, M. Parris, F. D. Smith, and T. M. Talley, "A router-based congestion control scheme for real-time continuous media," in *Proceedings of NOSSDAV'96*, (Zushi, Japan), April 1996.
16. S. Floyd and K. Fall, "Router mechanisms to support end-to-end congestion control." Network Research Group, Lawrence Berkeley National Laboratory. `ftp://ftp.ee.lbl.gov/papers/collapse.ps`, February 1997, [September 11, 1997].
17. S. McCanne and V. Jacobson, "Receiver-driven layered multicast," in *SIGCOMM'96*, (Stanford, California), August 1996.