January 1997

# Flow and congestion control for internet media streaming applications

Shanwei Cen

Calton Pu

Jonathan Walpole

# Flow and Congestion Control for Internet Media Streaming Applications*

Shanwei Cen, Calton Pu and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
*scen, calton, walpole@cse.ogi.edu*

## Abstract

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 our long-time experience in building and using an adaptive real-time streaming video player. The proposed scheme is designed to allow SCP-based streaming traffic to live in harmony with each other, and with TCP-based traffic. At the same time, it improves smoothness in streaming, and ensures low, predictable latency and effective use of network bandwidth. SCP also incorporates mobility awareness and supports limited data rates and possible pauses in streaming. In this paper, we present a description and analysis of SCP, and an evaluation of it through Internet-based experiments.

## 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 the Berkeley MPEG player [15], the OGI (Oregon Graduate Institute) distributed video player [4], the Vosaic player [5], and the Mbone tools [11]. Over the past year, many industrial streaming media players have also been released, such as the Netscape streaming video plug-ins [12], RealAudio [14] and Vxtreme [17]. It is expected that real-time media streaming traffic will increase rapidly, and will soon make up a significant portion of the total Internet bandwidth.

The key characteristics of such real-time streaming applications are the potential for high data rates, and the need for low and predictable latency and latency variance. Unfortunately, the Internet is characterized by a great diversity in host processing speed and network bandwidth, wide-spread resource sharing among applications for bulk data transfer, interaction, and media streaming, and highly dynamic workload. The Internet is also currently a best-effort network, without any facility for resource reservation or QoS guarantees. Consequently, Internet-based applications experience big variations in available bandwidth, latency and latency variance. For a streaming application to survive in this highly dynamic Internet environment, feedback-based adaptation and robustness in the presence of data loss are necessary. For example, streaming media players can preserve the real-time play-out of their data by adaptively sacrificing other presentation quality dimensions such as the total reliability, video frame rate, spatial resolution and signal-to-noise ratio. Through feedback-based adaptation, streaming applications can dynamically discover the currently available bandwidth, and scale the media in one or more of these quality dimensions to fully utilize that bandwidth. To mask short-term variations in the available bandwidth or end-to-end latency, players typically buffer data at the sender or receiver or both. Reliability through indefinite data retransmission is not desirable since streaming applications can often tolerate some degree of data loss, but can not usually tolerate the delay introduced by the retransmission of lost data.

Successful adaptation relies on accurate and reliable discovery of the currently available network bandwidth. This is achieved through flow and congestion control mechanisms. It is important for such mechanisms to be "good network citizens". That is, they should not allow an undue amount of traffic to be generated, congesting the network, and causing other network traffic to back-off. Similarly, they should be sensitive to increases in network congestion, and should respond to them by backing-off. Without this behavior, their potential to generate very high data rates could cause serious congestion in the Internet, and perhaps another Internet congestion collapse [9, 6]. Consequently, such mechanisms must operate in harmony with TCP [9], which is the base for the currently dominant FTP [13] and Web/HTTP [1] traffic. They should ensure that multiple streaming sessions share the network among themselves and with other non-streaming traffic in a fair manner. Finally, they should attempt to minimize latency and maximize the smoothness of the streaming data.

This paper describes SCP (Streaming Control Protocol), a unicast [1] streaming flow and congestion control

*This project is supported in part by grants from DARPA under grant N00014-94-1-0845 and contract MDA904-95-C-5547, and by Tektronix, Inc. and Intel Corporation.

[1]There are two types of streaming: unicast and multicast. A unicast stream is sent from a single sender to a single receiver, while a multicast stream can be simultaneously received by multiple receivers. Because of this fundamental difference, unicast and multicast streaming need to be treated differently in flow and con-

scheme that has the properties described above. SCP has been incorporated into our adaptive streaming video player [3]. Similar to the congestion control in TCP, SCP employs sender-initiated congestion detection through positive acknowledgement, and uses a congestion-window based policy to back-off exponentially. During the start-up phase, SCP uses TCP-style slow-start policy to quickly discover the available network bandwidth. The similarities of the SCP congestion control to that of TCP make SCP as robust and as good a network citizen as TCP, and enables the two of them to share the Internet fairly. But unlike TCP, when the network is not congested, SCP invokes a combined rate- and window-based flow control policy, that maintains smooth streaming with maximum throughput and low latency. While TCP repeatedly increases its congestion window size, causes packet loss, and backs off, SCP tries to maintain an appropriate amount of buffering in the network for sufficient utilization of available bandwidth. SCP also ensures fair and stable partitioning of network bandwidth between multiple streams. SCP does not retransmit data lost in the network, thus it avoids the associated unpredictability in latency and waste of bandwidth. Streaming sessions have unique properties not identified in non-real-time sessions, such as limited source data rate (e.g. a typical MPEG1 video stream has a data rate of around $1.5Mbps$), pause in the middle (e.g. when the user hits a 'pause' button) etc. SCP is designed to handle these characteristics properly. SCP also has mobile awareness [7]. Its internal states and parameter estimators can be reset upon mobility events such as switching between different network interfaces (e.g Ethernet and wireless PPP), the slow-start policy is then invoked to quickly discover the capacity of the new network connection.

This paper is organized as follows. Section 2 discusses other approaches to streaming control. Section 3 presents the design of SCP. This is followed by an analysis of steady-state bandwidth sharing in Section 4. Next, Section 5 briefly describes the implementation of SCP. Then Section 6 presents the Internet experiments and their results. Finally, we summarize the work in Section 7.

## 2 Related Work

Rate-based feedback is a widely used approach to streaming flow control. In a rate-based feedback scheme, the receiver continuously monitors the quality of streaming, such as receiving data rate or loss rate/ratio, and sends feedback messages to the sender to adjust the data rate accordingly. Rate-based feedback can be found in the Berkeley MPEG player [15], the Vosaic player [5], etc. Rate-based flow control based on RTP [16] as proposed in [2] works with multicast as well as unicast streaming applications. While rate-based feedback works well most of the time, it has the inherent danger of overflowing the network buffers, since the directly controlled metric is the data rate instead of the amount of buffering inside the network. 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 time-/state-based and involves sluggish low-pass filtering, and feedback messages (to reduce the data rate) may not be able to get through to the source in time. These drawbacks make the rate-based schemes more aggressive than TCP, thus forcing the latter to back off when competing insufficient network bandwidth.

In a rate-based feedback scheme, selecting thresholds for data rate increase/decrease is inherently problematic, no matter whether the thresholds are fixed or dynamic. The base packet drop ratio of the Internet is highly variable, closely depending on the specific connection and time in question. We have observed virtually no packet loss for connections between OGI and Georgia Institute of Technology in quiet evenings and weekends, but more than 10% during rush hours. For example, [2] proposed fix packet drop rate thresholds, such as 2% for data rate increase, and 4% for decrease. These thresholds would make a streaming session more aggressive than TCP in quiet hours, and prevents the streaming sessions from sending any data during rush hours. On the other hand, if the thresholds are adjusted dynamically depending on the network condition observed, bandwidth sharing between multiple sessions becomes a problem. If two sessions competing the same set of bottleneck links, it is inevitable that one session come up with higher thresholds than the other, and eventually force the latter to bail out completely.

In [10], a TCP-friendly unicast rate-based flow control scheme is proposed. By having the the sender (instead of the receiver) to monitor the overall packet loss ratio based on the acknowledgements from the receiver. This scheme solves the problem of feedback messages not getting though in time in the traditional rate-based feedback approach. Nevertheless, the other problems such as sluggishness in loss rate estimation, and the danger of network buffer overflow still exist. As to the packet loss thresholds, though the authors did observe that the they should be chosen so a streaming session using the proposed congestion control would send at a rate similar to that of an equivalent TCP session, no method of selecting these thresholds is given. We would expect the same threshold selection problem mentioned above to show up in this scheme.

Another choice for streaming flow and congestion control is HTTP [1]/TCP [9]. The Internet has proved its robustness, and its ability to share the bandwidth between multiple sessions. TCP has been used by many media streaming players (mostly as an option instead of as default), including RealAudio [14], Vxtreme [17], several Netscape plug-ins [12], etc. However, TCP is designed for reliable distribution of non-real-time data. When used for media streaming, it has the drawbacks of unpredictable latency, wasted network bandwidth, and bursty data flow. TCP implements reliability through infinite retransmission, causing unpredictable and unbounded latency, es-

gestion control as well as in many other aspects.

pecially in congested networks. It wastes network bandwidth by retransmitting late data. TCP is also inherently bursty even in steady-state, due to its repeated process of increasing its congestion window size until packets are lost, then backing off exponentially.

As discussed in [8], it is also possible to use TCP minus retransmission for streaming application. Removing the retransmission part from TCP eliminates the resultant problems of latency unpredictability and wasted network bandwidth. But due to TCP's congestion control algorithm, the streaming remains inherently bursty.

## 3 SCP Architecture

This section presents the design of SCP following a hierarchical method for adaptive system construction [3]. SCP is composed of a set of policies for congestion window size adjustment, each of which has a domain (network and session condition) in which it is applicable. When triggered by events indicating changes in network and session condition, a suitable policy is activated.

### 3.1 Media Streaming with SCP

A unicast streaming scenario with SCP consists of a media sender and a media receiver, linked by a network connection, with SCP residing at the sender side. Media packets are streamed in real-time from the sender to the receiver. Each packet carries among other fields an incremental sequence number. For each packet, SCP records the time when it is sent, and initiates a separate timer. The receiver acknowledges each packet received with an ACK carrying the sequence number of the packet. Based on the reception of ACKs and timer expiration events, the sender adjusts the size of its congestion window to control the flow and avoids network congestion. SCP maintains following internal state variables and parameter estimators.

- *state* — The current state: *paused*, *slowStart*, *steady*, or *congested*. Each state corresponds to a specific network and session condition and a flow and congestion control policy.
- *next* — The sequence number of the next packet to send. It is incremented by 1 after sending of each packet.
- *acked* — The sequence number of the latest packet whose ACK was received or whose timer has expired.
- $W_l$ — The size of the congestion window (number of packets).
- $W = next - (acked + 1)$ — 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.
- $W_{ss}$ — The threshold of $W_l$ for switching from slow-start policy to steady-state policy.
- $\hat{T}_{brtt}$ — Estimator of the base RTT (round trip time) – the transmission RTT of a packet sent when the network is otherwise quiet.
- $\hat{T}_{rtt}$ — Estimator of the recent average RTT.
- $\hat{D}_{rtt}$ — Estimator of the standard deviation of the recent RTT.
- $\hat{rto}$ — Estimator of the timer duration.
- $\hat{r}$ — Estimator of the packet rate – the rate at which ACKs are received.

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 bursting them out in chunks, so as to improve smoothness in streaming. Whenever an ACK is received, or a timer expires, the congestion window size $W_l$ is adjusted using a policy as determined by the current state.

- ACK$_i$ is received  If $i > acked$ ($i > acked + 1$ indicates a gap [2] in ACK), then $acked \leftarrow i$, estimators $\hat{T}_{brtt}$, $\hat{T}_{rtt}$, $\hat{D}_{rtt}$, $\hat{rto}$ are updated, all un-expired timers for packets up to and including $i$ are reset, and $W_l$ is adjusted. If $i <= acked$, ACK$_i$ is a duplicate and is ignored.
- The timer for packet $i$ ($i > acked$) expires  The un-expired timers for all packets up to $i$ are reset, and $W_l$ and $\hat{rto}$ are adjusted.

### 3.2 Overall Architecture

The observation on which SCP is based is that excessive packets in the round-trip network connection result in buffer buildup in the network routers and switches, and increase in RTT. When too many packets are held inside the network, buffers may overflow, and packets are dropped. The amount of buffering is affected by all the real-time streaming and non-real-time data transfer sessions sharing the network.

To adapt a streaming session to the changing network conditions, SCP tries to quickly find out how much buffering is appropriate for maximum throughput while avoiding over-buffering, or buffer overflow and resultant packet loss. As SCP runs, it keeps pushing an appropriate number of additional packets into the network connection to maintain the optimal condition, traces the changes in available bandwidth closely. When network congestion is detected, SCP reacts immediately with exponential backoff. Depending on the condition of the network and the streaming session under control, SCP is in one of the four states: *slowStart*, *steady*, *congested* and *paused*. Each state is associated with a specific condition and a congestion window size adjustment policy as listed in Table 1.

The domain of each state is guarded against events indicating if the domain is entered or left. Upon these events, actions are taken by SCP to update its internal variables, to transfer to a new state and switch to a new policy. These actions change the internal structure of SCP, thus can be referred to as meta-adaptation actions. There are events indicating whether an SCP session becomes paused or active, whether the available network bandwidth has been fully utilized, whether the network is congested, etc.

[2]Packets are assumed to be transmitted in-order. Gaps in ACK sequence numbers indicate packets loss. This assumption is relaxed in the actual 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 maximum 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 slow-start policy. |

Table 1: SCP states, their associated network and session conditions (domains), and congestion window size adjustment policies

| Category | Events | Guarded states | SCP Meta-adaptation actions |
|---|---|---|---|
| Mobility indication | Network interface switch | all states | Resets and enters the *paused* state. |
| Network becomes congested | timeout; gap in ACKs | *slowStart*, *steady*, *congested* | Backs off and and enters the *congested* state. |
| Bandwidth becomes fully utilized | RTT significantly long ($\hat{T}_{rtt} > K\hat{T}_{brtt}$ where $K > 1$); $W_l \geq W_{ss}$ | *slowStart* | Enters the *steady* state. |
| Session becomes paused | No more outstanding packet ($W = 0$) | *slowStart*, *steady*, *congested* | Enters the *paused* state. |
| Session becomes active | A new packet is sent | *paused* | Shrinks the window size and enters the *slowStart* state. |

Table 2: Events and SCP meta-adaptation actions

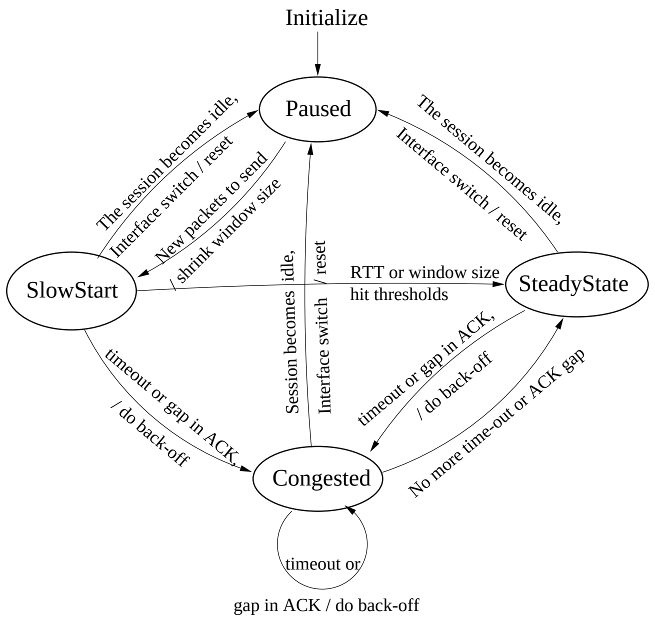


Figure 1: SCP state transition diagram

There are also explicit events such as network interface switch. Table 2 lists the events, their categories, the states (domains) in which they should be guarded, and the meta-adaptation actions. Figure 1 then shows how SCP transfers between its states upon the events. If multiple events happen simultaneously, the event listed first in Table 2 takes precedence.

## 3.3 Congestion Window Adjustment Policies and State Transition

**Initialization** Upon initialization, SCP sets $acked = 0$, $next = 1$, $W = 0$, $W_l = 1$ and $W_{ss} = LW_{ss}$, where $LW_{ss}$ is a limit on the congestion window size. $\hat{T}_{brtt}$, $\hat{T}_{rtt}$ and $\hat{D}_{rtt}$ are all set to $\infty$. $\hat{rto}$ is set to an initial default value. After initialization, SCP enters the *paused* state. Sending 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 open up the congestion window and detect the available network bandwidth. $W_l$ is incremented by 1 upon each ACK received in-order, thus is doubled after each RTT amount of time. SCP leaves the *slowStart* state upon events indicating network congestion, full utilization of available bandwidth, pause in streaming, or network interface switch (Fig. 1).

**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 changes in available bandwidth closely. In the *steady* state, estimation of ACK rate $\hat{r}$ is enabled. Whenever a new $\hat{r}$ value is available, $W_l$ and $W_{ss}$ are adjusted according to the following Equation 1, where $W_\Delta$ is a constant referred to as window size incremental coefficient.

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

The idea behind the above flow control Equation 1 is that SCP assumes that $\hat{r}$ is an approximation to the network bandwidth actually available to the session, and calculates the bandwidth-delay product of the network connection when buffering is kept minimum: $\hat{r} \times \hat{T}_{brtt}$. This product is the amount of data SCP should keep outstanding inside the network in order to maintain maximum throughput with minimum buffering. Since the network is shared and highly dynamic, the bandwidth available to the session in question changes. If SCP just keeps $\hat{r} \times \hat{T}_{brtt}$ amount of data outstanding, when the available bandwidth decreases, $\hat{r}$ would decrease, thus SCP would trace the change by reducing the amount of outstanding data. However, if the available bandwidth increases, there is no way for SCP to detect that. To solve this problem, SCP pushes $W_\Delta$ amount of extra outstanding data. When the network available bandwidth increases, releasing the extra data buffered by the network results in increase in packet rate $\hat{r}$, which in turn results in larger amount of outstanding data as defined by the increased $W_l$. It is easy see that the increase in $W_l$ is no more than additive, which is what TCP does in its steady state. Though at the beginning $\hat{r}$ may not be a reliable estimation of the actual available bandwidth, if the network is stable enough, the same policy in Equation 1 would eventually bring $\hat{r}$ to the actual bandwidth through iterations.

Upon network congestion events, SCP does back-off and enters the *congested* state. SCP enters the *paused* state if the session becomes paused.

**Exponential Back-off Upon Network Congestion** Whenever the network is congested, as indicated by events such as gap in ACK or timer expiration, SCP backs off multiplicatively by reducing its congestion window size in half:

$$W_{ss} = W_l = \frac{W_l}{2}$$

The data rate estimator $\hat{r}$ is no longer accurate and thus reset and disabled. If the back-off is triggered by timeout, $\hat{rto}$ may have been too short and thus is doubled. With the multiplicative decrease in congestion window size and multiplicative increase in timer duration, in the case when the network is so congested that no packet can get through, SCP would back 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 no reflect correctly the network condition after the back-off, and thus should be ignored. The *congested* state is designed for this purpose. 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 No Packet to Send** Every real-time streaming session has a finite data rate, and there may not be data to send when the congestion window is open. Also a user may want to pause a streaming session temporarily 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 uses will gradually be discovered and taken by other sessions. So when the session resumes at a later time, it should start with the *slowStart* state at a reduced congestion window size. Currently, a policy is adopted to half 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, the route from the sender to the receiver is changed, and the new connection may go through links with totally different capacities (e.g Ethernet and wireless PPP). SCP handle this mobility issue by making it reset-able. Upon each network interface switch event, SCP discards the no-longer valid parameter estimations, ignores all outstanding ACKs, and restart afresh with a slow-start policy to discover the capacity of the new connection quickly.

## 4 Bandwidth Sharing Analysis

With the steady-state policy stated in Equation 1, it is possible for multiple SCP sessions to share network links in a 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, whenever the congestion window is open.

Suppose in a steady state, session A has estimations $\hat{r}_1$, $\hat{T}_{brtt1}$, $\hat{T}_{rtt1}$, and $W_{l1} = \hat{r}_1\hat{T}_{brtt1} + W_\Delta$. Session B has $\hat{r}_2$, $\hat{T}_{brtt2}$, $\hat{T}_{rtt2}$, and $W_{l2} = \hat{r}_2\hat{T}_{brtt2} + W_\Delta$. Since A and B share the same network link, we can make the following observations:

(1) The aggregate packet rate $\hat{r}_l$ of the network link is the sum of the two sessions: $\hat{r}_l = \hat{r}_1 + \hat{r}_2$.

(2) Base RTT estimator is the actual link base RTT $T_{brttl}$ plus some estimation error: $\hat{T}_{brtt1} = T_{brttl} + E_1$ and $\hat{T}_{brtt2} = T_{brttl} + E_2$. 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 — packets of other sessions preventing the network buffers from becoming empty. Other less important components include sampling noise, etc.

(3) Sessions A and B have the same RTT estimation (when the sampling noise can be ignored): $\hat{T}_{rtt1} =$

$\hat{T}_{rtt2} = \hat{T}_{rttl}$. 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:

$$\frac{\hat{r}_1}{\hat{r}_2} = \frac{W_{l1}}{W_{l2}} = \frac{\hat{r}_1(T_{brttl} + E_1) + W_\Delta}{\hat{r}_2(T_{brttl} + E_2) + W_\Delta} \qquad \Longrightarrow$$

$$\hat{r}_1 = \frac{\hat{r}_2 W_\Delta}{\hat{r}_2(E_2 - E_1) + W_\Delta} \tag{2}$$

Combining observations (1) and (3), it is clear that there is a single solution for $\hat{r}_1$ and $\hat{r}_2$. The session with a bigger residual buffering tends to have a bigger error $E$, and thus gets a bigger portion of the bandwidth. In a special case where the two sessions have the same base RTT estimation: $E_1 = E_2$, we have $\hat{r}_1 = \hat{r}_2 = \frac{\hat{r}_l}{2}$, indicating that the two sessions split the network bandwidth evenly.

# 5 Implementation of SCP

An SCP package has been implemented as a layer on top of UDP, based on a toolkit for developing adaptive systems [3], which is also being developed at OGI. This SCP package has also been incorporated into the OGI adaptive streaming video player [4, 3]. In this section, we briefly discuss the the parameter estimators and various implementation issues. More details can be found in [3].

## 5.1 Estimation of the Parameters

Base RTT $\hat{T}_{brtt}$, average RTT $\hat{T}_{rtt}$, RTT standard deviation $\hat{D}_{rtt}$, and timer duration $\hat{rto}$ are estimated based on the history of the raw RTT measurements of the current session. $\hat{T}_{brtt}$ is estimated as the minimum of all the past RTT measurements. The $\hat{T}_{rtt}$ and $\hat{D}_{rtt}$ estimations are done in a way similar to that in TCP [9]. The $\hat{T}_{rtt}$ is the result of applying a lowpass filter [3] to the sequence of raw RTT measurements. $\hat{D}_{rtt}$ is a lowpass filtering of the difference between $\hat{T}_{rtt}$ and the raw RTT measurements. $\hat{rto}$ is also estimated in a way similar to that in TCP [9], which proposed $\hat{rto} = \hat{T}_{rtt} + 4\hat{D}_{rtt}$. Unfortunately, in our implementation with $10ms$ clock resolution, the above $\hat{rto}$ estimator results in timers being too sensitive to jitter in RTT. Many timers expire though their ACKs are not actually lost and will show up soon. This is our experience as well as an observation in LINUX's implementation of TCP [4]. To avoid this problem, we adopt LINUX's modified version of the $\hat{rto}$ estimator:

$$\hat{rto} = \frac{5}{4}(\hat{T}_{rtt} + 4\hat{D}_{rtt})$$

The average ACK reception rate $\hat{r}$ is estimated as the inverse of the recent average ACK interval measurements, which is in term obtained by applying a lowpass filter on the raw interval measurements.

[3]A simple lowpass filter is defined by $v = (1-\alpha)v + \alpha u$, where $u$ is the raw data, $v$ is the smoothed output, and $0 < \alpha \leq 1$ is a parameter.

[4]See comments around line number 860 in file `net/ipv4/tcp_input.c` of the LINUX version 2.0.18 source code for more details

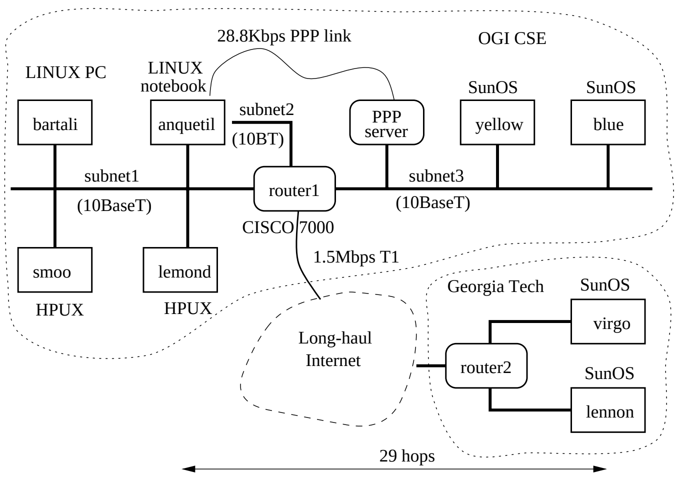


Figure 2: Network configuration for the SCP experiments

## 5.2 Implementation Issues

**Packet Reordering** The IP protocol used by the Internet is connection-less and best-effort. IP packets may be dropped or duplicated. They may also be reordered. In the presence of packet reordering, a gap in ACK does not necessarily mean that packets have been dropped. Whenever a gap in ACK is detected, SCP temporarily assumes that the missing ACKs are lost and backs off, but with the hope that they are just late, also saves the current state (*state*, $W_l$, etc.), and continues packet rate estimation. Later if the missing ACKs show up when SCP is still in the *congested* state, the saved state is restored. As a result, SCP responds to potential packet drop immediately, but the throughput will not be affected much if the out-of-order ACKs are received soon.

**Low data rate session** Every streaming session has a limited data rate. It is possible that the rate of a session is lower than the available bandwidth (so that no packet is dropped) but is big enough to keep the session active. In this case, the session would stay in the slow-start state, and the congestion window size would be driven quickly to the limit $LW_{ss}$ which is usually big. On the other hand, only a small number of packets are outstanding. If later there is a big burst of packets, they will all be sent out virtually back to back, potentially causing heavy packet drop. To prevent this from happening, in *slowStart* state, SCP increases $W_l$ only when the congestion window is sufficiently full, and shrinks it if it is almost empty.

# 6 Experiments and Results

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

The network configuration for the experiments is shown in Fig. 2. We have two LANs, one at OGI CSE department, and the other at Georgia Institute of Technology (Georgia Tech) connected through the long-haul Internet with almost 30 hops. Among the hosts, *anquetil* is a LINUX PC notebook, homed at subnet 1 in the CSE OGI LAN. But it can optionally be moved to subnet 2, and can have a $28.8Kbps$ PPP connection. This configuration covers typical types of Internet connections such as PPP, LAN (single subnet, and across a single router), and WAN. Network interface switch in mobile environment is simulated by having the notebook *anquetil* to switch between its Ethernet and PPP interfaces manually.

For all the experiments, unless stated explicitly, following parameter values are used: steady-state congestion window size incremental coefficient $W_\Delta = 3$, RTT threshold coefficient $K = 2$, rate estimator lowpass filter parameter $\alpha = 0.1$, data packet size $1472B$ [5], and a limit on congestion window size $LW_{ss} = 43$. [6] The size of TCP and UDP(SCP) socket sending and receiving buffers are set up to 64KB for better throughput.

## 6.1 Experiments Across PPP Link

The first set of experiments are between *anquetil* as the receiver and *lemond* as the sender across the $28.8Kbps$ PPP link with an MTU of $1500B$.

First, single TCP and SCP sessions are played. For the SCP sessions, the receiving data rate is $2.24pkt/s \approx 26.9Kbps$, which is close to the $28.8Kbps$ PPP link speed. Figure 3(a) shows the packet rate histograms of an SCP session and a TCP one. 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 [7] histograms 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, excepts for the periodical downward spikes, which are caused by packet loss (by the PPP server) and retransmission. However, SCP and TCP sessions maintain different levels of buffering inside the network. The SCP session has a steady-state buffering delay about 1.2 second, while the TCP session pushes the buffering delay up to 16 seconds (which corresponds to a full and periodically overflowing buffer in the PPP server) 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 HPUX TCP, but a much worse start-up phase. LINUX TCP seems introduced some techniques for the PPP case to manage the buffering. It pushes the delay to 7 seconds, and reduces to about 3 second in its steady state.

Figure 4 shows the smoothed[8] packet rate histograms of three SCP sessions, when two of them are played simultaneously from *bartali* to *anquetil* across 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 the residual buffering error in the based RTT estimation on bandwidth sharing. The base RTT estimation of the long session has a residual buffering caused by the first short session, thus it gets a bigger share of the bandwidth. The second short session then 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.

Experiments have also been carried out to play one SCP session and one TCP session simultaneously across the PPP link. Since SCP controls 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. HPUX 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.

## 6.2 Experiments On Single Subnet

All the single SCP sessions played from *bartali* to *anquetil* across subnet 1 yield smooth and stable throughput of about $700pkt/s = 8.4Mbps$. The buffering delay of these sessions, which are mainly caused by MAC-level back-off and host processing latency, is kept within $4ms$. For comparison, individual TCP sessions yield less smooth $670 \sim 680pkt/s$ 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*$\rightarrow$*smoo* and *anquetil*$\rightarrow$*lemond* across subnet 1, the two SCP/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 is the result of MAC-level exponential back-off and retransmission failure upon collision. Not much can be done by SCP or TCP to eliminate the problem except for explicitly limiting the data rate at the sender side.

## 6.3 Experiments Across Single Router

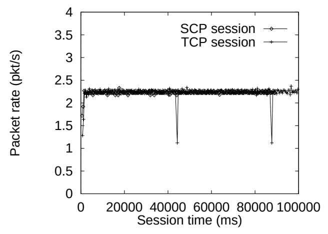
To evaluate the performance of SCP streaming across a single router, *anquetil* is moved to subnet 2, and experiments are performed. Table 3 shows the configurations and the overall performance of the experiments. Figure 5

[5]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 randomized data.

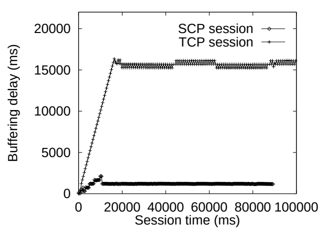
[6] $\frac{64KB}{1.5KB} \approx 43$, where $1.5KB$ is the Ethernet MTU.

[7]The time $T$ from when the sender SCP or TCP accepts a packet until when the receiver gets it, minus the transmission latency estimated as the minimum of all the $T$'s in the whole session.

[8]By applying a lowpass filter with $\alpha = 0.1$ to the raw data. $\alpha = 0.01$ is used for all non-PPP packet rate histograms.



(a) Packet rate



(b) Buffering delay

Figure 3: Performance histograms of single SCP and TCP sessions over PPP

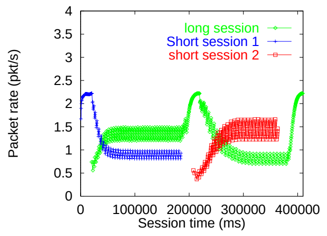


Figure 4: Smoothed pkt rate of two SCP sessions over PPP

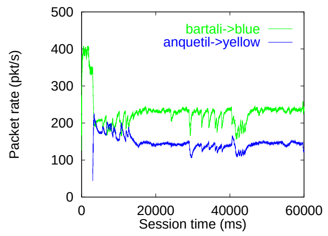


Figure 5: Smoothed pkt rate of two SCP sessions across single router

| 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 3: Results of single TCP and SCP sessions across a single router

shows the smoothed packet rate of two simultaneous SCP sessions. These experiments were done in a late evening when the network is lightly-loaded.

A single SCP session yields a lower throughput than a single TCP session, but has a significantly lower buffering delay. One possibly reason for an SCP to have a lower bandwidth is that SCP maintains lower level of buffering, thus has more chances of getting empty buffers and idle links.

Figure 5 shows that when two SCP sessions are competing 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 statistic data shows 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 difference in host CPU speeds and other factors, and results in an uneven bandwidth partitioning. This doesn't mean that the SCP is unfair.

When an SCP session is competing against a TCP session, they are still 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 of the bandwidth.

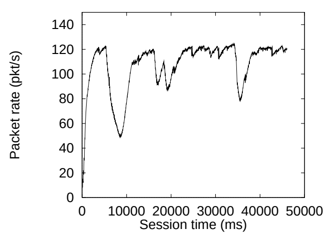
## 6.4 Experiments Across the Internet

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 Internet. Nevertheless, there are still times, such as midnights and weekends, when the network is relatively lightly loaded, and times such as weekdays when it is heavily loaded.

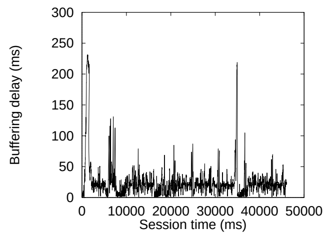
As shown in Fig. 6, when the network is lightly loaded, SCP is able to stream smoothly, and to explore the network bandwidth sufficiently. From time to time, there are still network congestion and packet loss (Fig. 6(d), which also shows that there is an out-of-order packet. In this case, $W_l$ is halved but restored before any packet is sent, thus the back-off is not shown in Fig. 6(c)), causing SCP to back off. Otherwise, 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 $115pkt/s (\approx 1.4Mbps$, close to the $1.5Mbps$ T1 line speed), and low buffering delay. On the other hand, as shown in Fig. 7, single TCP sessions have much jerkier throughput, and long and unpredictable buffering delay (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 histograms of two simultaneous SCP sessions. Random packet drop in routers upon congestion causes competing sessions to back-off randomly, but on average the bandwidth sharing is fair, and the throughput is not too jerky.

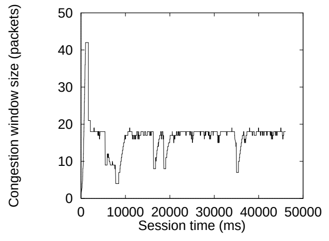
SCP and TCP sessions across the long-haul Internet share the network bandwidth more evenly than when across a single router. Figure 9 shows the packet rate histograms of two simultaneous SCP and TCP sessions. The two competing sessions, SCP: *lennon→anquetil* and TCP: *virgo→bartali*, produce a throughput ratio around $SCP 50pkt/s : TCP 40pkt/s$. While TCP sessions have long latency, the buffering delay of the competing SCP sessions are still below $100ms$ for most packets.



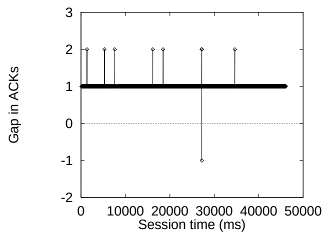
(a) Smoothed packet rate



(b) Buffering delay

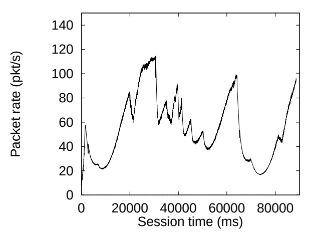


(c) Cong. window size

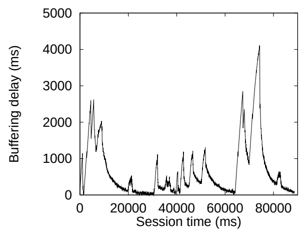


(d) Gap in ACKs

Figure 6: Histograms of a single SCP session across the lightly loaded Internet



(a) Smoothed packet rate



(b) Buffering delay

Figure 7: Histograms of a TCP session across the lightly loaded Internet

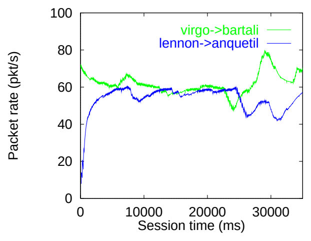


Figure 8: Smoothed pkt rate of two SCP sessions across the Internet

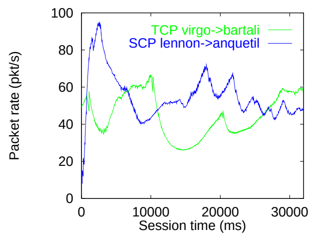


Figure 9: Smoothed pkt rate of SCP and TCP sessions across the Internet

During busy weekdays, it is more obvious that SCP yields smoother throughput and lower and more consistent delay than TCP, while still operate in harmony with the latter. The performance results of two SCP and TCP sessions from *virgo* and *bartali* are shown in Fig. 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 delay [9] below $100ms$. SCP sessions observed up to 10% loss in ACKs. Due to the heavy packet loss and frequency retransmission, TCP sessions get only 2 to $6pkt/s$, with delay up to 70 seconds, and the throughput is very jerky. 70 second delay is simply not acceptable to any streaming applications!

## 6.5 Network Interface Switch

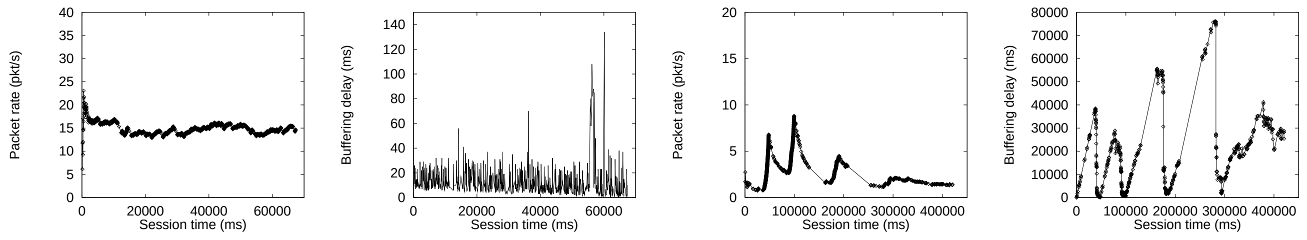
To evaluate SCP's ability to adapt to different network capacities upon network interface switch, both the PPP and Ethernet interfaces of the notebook *anquetil* are activated, and SCP sessions are played from *bartali* to *anquetil* through either the PPP or the Ethernet link. Switch between PPP and 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.

[9] In a busy network when the buffers in routers are never empty, buffering delay histograms actually reflects the variations in buffering delay.

In our experiment, upon network switch, 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 a WAN connection such as the one between OGI and Georgia Tech. with an optimal congestion window size of 18 packets, we expect that reset on SCP would make a difference. The slow-start policy following the reset helps figure out the new window size quickly. However, our experiments show that reset on SCP helps keep the parameter estimations accurate. The base RTT for a $1500B$ packet over Ethernet is less than $1ms$, but it is about half a second across a PPP link. If SCP is not reset, the base RTT estimation would stay less than 1 millisecond after switching from Ethernet to PPP. This severe under-estimation of base RTT would result in the calculated congestion window size to be much lower than it should be. Fortunately the minimum steady-state congestion window size of $W_\Delta = 3$ is sufficient for exploring the network bandwidth of both PPP and Ethernet LAN, so the under-estimation does not hurt the throughput much. We expect the under-estimation to hurt throughput of a WAN session seriously.

# 7 Discussion

We have presented SCP, an effective flow and congestion control scheme for real-time media streaming appli-



(a) SCP packet rate (b) SCP buffering delay (c) TCP packet rate (d) TCP buffering delay

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

cations. SCP eliminates the unpredictability in latency caused by retransmission of lost packets. It uses positive acknowledge to detect network congestion, and rate- and window-based policies to control the streaming flow effective. During start-up phase, SCP employs a slow-start policy to open the congestion window quickly. When the network is not congested, SCP's rate- and window-based policy ensures smooth streaming, sufficient use of the network bandwidth and low latency. It also enables a stable sharing of network bandwidth between multiple streams. Upon detection of network congestion, SCP backs off exponentially. SCP has features to handle the limited data rate and possible pause in streaming. With the ability to reset its internal state and parameter estimators, SCP also has mobile awareness.

Experiments have been conducted to stream packets over typical network configurations, including PPP, single subnet and across single router (LAN), and across the long-haul Internet (WAN). The experiments demonstrated that SCP is able to stream data packets smoothly in all the configurations. The buffering delay of the stream is low and predictable. Multiple SCP streams are able to share the network bandwidth in a stable and in many case fair manner. SCP streams are also able to share the network bandwidth with TCP streams across the same network links. These properties of SCP makes it a promising protocol for real-time multimedia streaming across the Internet.

SCP has been incorporated into the OGI adaptive streaming video player [3]. 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. The experimental results in [4] show the robustness and adaptability of the streaming player.

## Acknowledgments

Discussions in the Quasar Group at CSE, OGI greatly motivated this work. Special thanks to Jon Inouye for his insights to TCP and other network protocols, and discussions on the ideas in this work. We are grateful to Prof. Karsten Schwan of Georgia Institute of Technology, for providing the Georgia Tech account we used in the Internet experiments.

## References

[1] Berners-Lee, T., Fielding, R., and Nielsen, H. Hypertext transfer protocol – HTTP/1.0. Internet RFC 1945, May 1996.

[2] Busse, I., Deffner, B., and Schulzrinne, H. Dynamic QoS control of multimedia applications based on RTP. *Computer Communications 19* (January 1996), 49–58.

[3] Cen, S. *A Software Feedback Toolkit and Its Applications in Adaptive Multimedia Systems.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1997. In preparation.

[4] Cen, S., Pu, C., Staehli, R., Cowan, C., and Walpole, J. A distributed real-time MPEG video audio player. *NOSSDAV'95, Lecture Notes in Computer Science 1018* (1995), 151–162.

[5] Chen, Z., Tan, S.-M., Campbell, R. H., and Li, Y. Real time video and audio in the World Wide Web. In *Fourth International World Wide Web Conference* (Boston, Massachussetts, December 1995).

[6] Floyd, S., and Fall, K. Router mechanisms to support end-to-end congestion control. Tech. Rep. 97, Network Research Group, Lawrence Berkeley National Laboratory, February 1997. ftp://ftp.ee.lbl.gov/papers/collapse.ps.

[7] Inouye, J., Cen, S., Pu, C., and Walpole, J. System support for mobile multimedia applications. In *NOSSDAV'97* (May 19-21 1997), pp. 143–154. To appear.

[8] Jacobs, S., and Eleftheriadis, A. Real-time dynamic rate shaping and control for Internet video application. In *Workshop on Multimedia Signal Processing* (Princeton, New Jersey, June 1997).

[9] Jacobson, V. Congestion avoidance and control. In *SIGCOMM'88* (August 1988), pp. 79–88.

[10] Mahdavi, J., and Floyd, S. TCP-friendly unicast rate-based flow control. end2end mailing list, `ftp://ftp.isi.edu/end2end`, January 1997.

[11] McCanne, S., and Jacobson, V. *vic:* A Flexible Framework for Packet Video. In *Proceedings of the Third ACM Conference and Exhibition (Multimedia '95)* (San Francisco, California, November 1995), pp. 511–522.

[12] Netscape Communications Corporation. Netscape plug-ins: Audio/video. `http:// home. netscape. com /comprod /products /navigator /version_2.0 /plugins /audio-video.html`, 1997.

[13] Postel, J., and Reynolds, J. File transfer protocol. Internet RFC 0959, October 1985.

[14] Progressive Networks. HTTP versus RealAudio client-server streaming. `http:// www. realaudio. com /help /content /http_vs_ra.html`, 1996.

[15] Rowe, L. A., et al. MPEG video in software: Representation, transmission and playback. In *Symposium on Elec. Imaging Sci. and Tech.* (San Jose, California, USA, February 1994).

[16] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A transport protocol for real-time applications. Internet Draft, March 1995.

[17] Vxtreme, Inc. Vxtreme streaming video player. `http:// www. vxtreme. com`, 1997.