An Integrated Congestion Management Architecture for Internet Hosts

Hari Balakrishnan, Hariharan S. Rahul, Srinivasan Seshan*
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
{hari,rahul}@lcs.mit.edu, srini@watson.ibm.com

Abstract

This paper presents a novel framework for managing network congestion from an end-to-end perspective. Our work is motivated by several trends in traffic patterns that threaten the long-term stability of the Internet. These trends include the use of multiple independent concurrent flows by Web applications and the increasing use of transport protocols and applications that do not adapt to congestion. We present an end-system architecture centered around a Congestion Manager (CM) that ensures proper congestion behavior and allows applications to easily adapt to network congestion. Our framework integrates congestion management across all applications and transport protocols. The CM maintains congestion parameters and exposes an API to enable applications to learn about network characteristics, pass information to the CM, and schedule data transmissions. Internally, it uses a stable rate-based control algorithm, a scheduler to regulate transmissions, and a lightweight loss-resilient protocol to elicit feedback from receivers. Its rate-based scheme uses additive increase/multiplicative decrease, combined with a novel exponential aging scheme when receiver feedback is infrequent, to obtain both stable network behavior and good application performance.

We describe how TCP and an adaptive real-time streaming audio application can be implemented using the CM. Our simulation results show that an ensemble of concurrent TCP connections can effectively share bandwidth and obtain consistent performance, without adversely affecting other network flows. Our results also show that the CM enables audio applications to adapt to congestion conditions without having to perform congestion control or bandwidth probing on their own. We conclude that the CM provides a useful and pragmatic framework for building adaptive Internet applications.

Keywords: Congestion control, Internet, adaptive applications

*IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

1 Introduction

The success of the Internet to date has been in large part due to the sound principles of additive-increase/multiplicative-decrease [4] on which its dominant transport protocol, TCP [16, 32], is based. Because most traffic in the Internet has been dominated by long-running TCP flows, the network has shown relatively stable behavior and has not undergone large-scale collapse in the past decade.

However, Internet traffic patterns have been changing rapidly and are certain to be very different in the future. First, Web workloads stress network congestion control heavily, and in unforeseen ways. Typical Web transfers are characterized by multiple concurrent, short TCP connections. These short Web transfers do not give TCP enough time or information to adapt to the state of the network, while concurrent connections between the same pair of hosts compete rather than cooperate with each other for scarce resources. Second, there are commercial products being developed today that "accelerate" Web downloads, usually by turning off or changing TCP's congestion control in unknown and potentially dangerous ways. Third, and perhaps most importantly, several increasingly popular real-time streaming applications run over UDP using their own user-level transport protocols for good application performance, but in most cases today do not adapt or react properly to network congestion.

All these trends, coupled with the unknown nature of future applications, threaten the long-term stability of the Internet. They make it likely that large portions of the network might suffer congestion-triggered collapse due to unresponsiveness in the face of congestion or aggressive mechanisms to probe for spare bandwidth. To some, this might sound overly pessimistic, but even the optimists amongst us will grant that applications should be able to track and adapt to congestion, available bandwidth,and varying network conditions to obtain the best possible performance. Unfortunately, protocol stacks today do not provide the right support for this; the desire to be a good network citizen forces applications to use a single TCP connection, even if this transport model is ill-suited to the application at hand. Or, more likely, because a single TCP connection is mismatched to the requirements of the application, the result is a proliferation of flows that are not well-behaved and are deleterious to the rest of the network.

Our work attempts to overcome these problems by developing a novel framework for managing network congestion from an end-to-end perspective. Unlike most past work on bandwidth management that focuses on mechanisms in the network to provide QoS to flows or reduce adverse interactions between competing flows [7, 25, 8, 5, 37, 2], we focus on developing an architecture at the end-hosts to:

- Ensure proper and stable congestion behavior by building on the well-proven principles of additive-increase/multiplicative (AIMD).
- Enable all applications and transport protocols to adapt easily to network congestion and varying bandwidth by providing adaptation APIs.

The resulting framework is independent of specific applications and transport protocol instances, but provides the ability for different flows to perform *shared state learning*. Here, flows learn from each other and share information about the state of congestion along common network paths.

Increasingly, the trend on the Internet is for unicast data servers to transmit a wide variety of data, ranging from best-effort (unreliable) real-time streaming content to reliable Web pages and applets. As a result, many logically different streams using different transport protocols will share the path between server and client. These streams have to incorporate control protocols that dynamically probe for spare bandwidth and react to congestion for the Internet to be stable. Furthermore, they will often have different reliability requirements, which implies that a general congestion management architecture should separate the functions of loss recovery and congestion control that are coupled in protocols like TCP.

At the core of our architecture is the *Congestion Manager (CM),* which maintains network statistics and orchestrates data transmissions governed by robust control principles. Rather than have each stream act in isolation and thereby adversely interact with the others, the CM coordinates host- and domain-specific path information. Path properties are shared between different streams because applications and transport instances perform transmissions only with the CM's consent.

Internally, the CM ensures stable network behavior by the sender because it reacts to congestion, carefully (and passively) probes for spare bandwidth, implements a robust and lightweight protocol to elicit feedback from receivers about losses and status, and schedules data transmissions by apportioning available capacity between different active flows. The CM's internal algorithms and protocols are described in Section 2, where we motivate them using ns-2 [21] simulation experiments and analysis.

The CM API is designed to enable easy application adaptation to congestion and variable bandwidth, accommodating heterogeneous flows. The API includes functions to query path status, schedule data transmissions, notify the CM upon data transmission, and update variables upon congestion or successful transmission. It also includes callbacks to applications upon rate change. Motivated by the end-to-end argument [29] and the principle of Application-Level Framing (ALF) [6], the CM API permits the application to have the final say in deciding what to transmit, especially when available bandwidth is smaller than what the application desires. We discuss our design decisions and the details of the API in Section 3. In the same section, we also discuss how two applications—a Web server and an audio server can be implemented using the CM API and adapt efficiently to congestion. Section 4 discusses the actual performance results for different applications.

The resulting end-to-end network architecture from the viewpoint of a data sender is shown in Figure 1. The CM frees transport protocols and applications from having to (re-)implement congestion control and management from scratch, and it discourages applications from using an inappropriate transport protocol (e.g., TCP for high-quality audio) simply because the transport implements a congestion control scheme. Above all, the CM provides the required support and a simple API over which adaptive Internet applications can be developed.

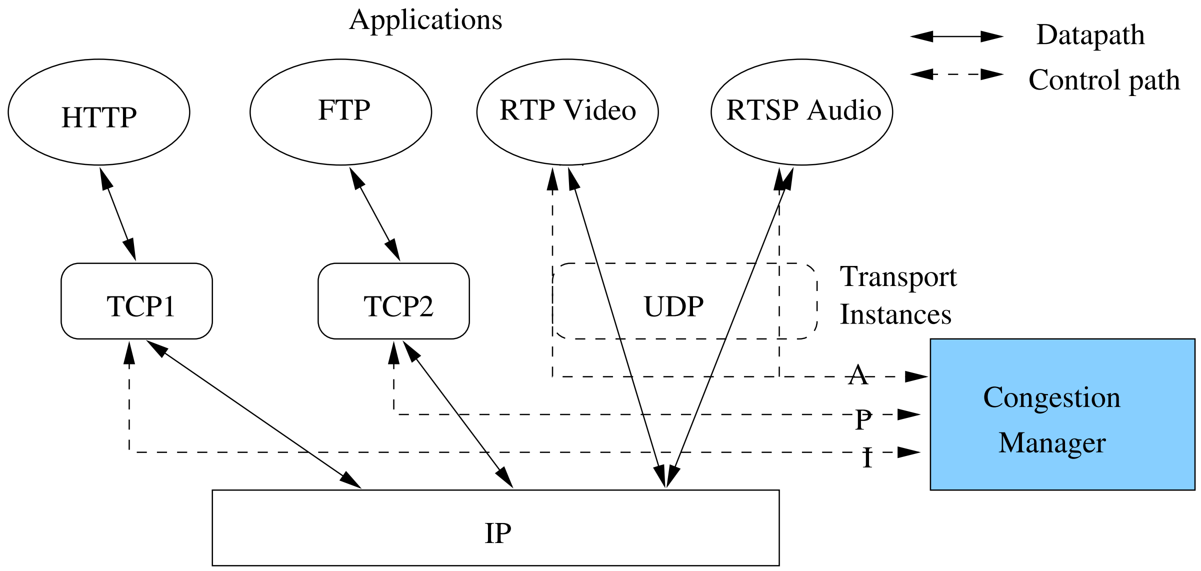

Figure 1: New sender architecture with centered around the Congestion Manager.

While we believe that there are several aspects of the CM that are novel, this is not the first paper to suggest aggregating congestion control information across flows. In RFC 2140 [34], Touch proposes a scheme called "TCP control block interdependence," where the goal is to share part of the TCP control block between connections to improve transient TCP performance, while maintaining backward-compatibility with existing implementations. In [1, 24], the authors present an integrated approach to TCP where TCP control block state is shared for better congestion control and loss recovery for concurrent connections. However, both these proposals restrict themselves to simultaneous TCP connections and do not consider other types of applications. Nor do they provide any APIs for application adaptation, primarily because they maintain TCP's API. As we will see in the rest of this paper, the mechanisms to probe for bandwidth, react to congestion and accommodate heterogeneous flows are significantly harder than multiplexing TCP flows alone. Section 5 discusses and compares our work to other approaches.

The following are our main contributions:

- **Congestion Manager (CM).** The design of a Congestion Manager to perform integrated congestion management across an ensemble of unicast flows in an application- and transport-independent manner. To ensure stable network behavior and shared state learning, the CM incorporates (i) a rate-based AIMD scheme, (ii) a loss-resilient protocol to periodically elicit feedback from receivers, (iii) an exponential aging mechanism to regulate transmissions in a stable manner when feedback is infrequent, (iv) loss-based segregation mechanisms for inferring the existence of routers implementing differential services, and (v) a scheduler to apportion bandwidth to flows.

- **CM adaptation API.** An API for applications and transport protocols to adapt well to network congestion and varying bandwidth. We also describe how TCP and an adaptive layered audio application can be implemented using the API.

- **CM applications and performance.** We present simulations of application performance that demonstrate that the CM ensures stable network behavior. It also greatly improves performance predictability and consistency, and enables applications such as audio servers to effectively transmit the best among several available source encodings.

2 CM Algorithms and Protocols

In this section, we present the CM's internal algorithms and protocols. We first present the CM's rate control algorithm based on AIMD and discuss experimental data showing its stable behavior and TCP-friendliness [10]. Then, we address the issue of receiver feedback, motivating why it is needed, how it is obtained, and what the CM does to ensure stability even when it is infrequent. We then discuss extensions to the CM to perform well over differentiated services network, by segregating flows based on observed loss rates and throughputs. We conclude this section with a description of the CM scheduler, which schedules all transmissions and ensures proper rate allocations to the different flows.

2.1 Stable Congestion Control

One of the key features of the CM is that it ensures proper congestion behavior. This implies that its mechanisms for reacting to network congestion and probing for spare capacity be sound and robust. In our current implementation, the CM achieves this by a *rate-based* AIMD control scheme. This rate changes as the CM learns from active flows about the state of the network and as it carefully increases the rates allocated to them to probe for spare capacity. The additive increase component is no more aggressive than a comparable TCP flow, in that both the number of bytes successfully transferred and the round-trip time estimate are taken into account in determining the rate increase. This does lead to a bias against long round-trip flows in a congested network [13, 36], but we felt that an accurate emulation of TCP's increase algorithm is currently the safest deployment alternative. Upon a loss, the rate reduces by a factor of two, and when persistent congestion occurs (e.g., a TCP timeout), the rate drops to a small value forcing slow start [16] to occur.

We chose to implement a rate-based instead of a TCP-like window-based scheme for two main reasons. First, carefully designed rate-based schemes avoid bursts of transmissions that window-based schemes (e.g., TCP) are prone to, which make them less likely to overwhelm bottleneck router buffers on the path to the receiver. Second, several applications, unlike TCP, provide relatively scarce and infrequent receiver feedback about received data, and our experiments showed that a stable rate-based scheme provides more consistent performance in these situations.

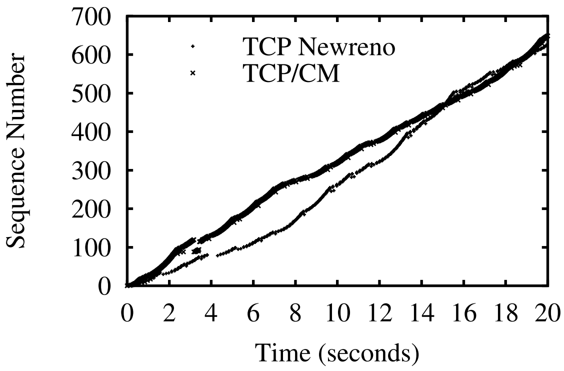

Figure 2: Sequence traces for TCP Newreno and TCP/CM, showing TCP/CM's true emulation of TCP

We conducted several experiments to validate the soundness of the CM's rate-based algorithm and tune it to perform well and in a TCP-friendly manner. Results from one set of experiments, for two connections—TCP Newreno [14] and TCP/CM—running over a network with random Web-like background traffic are shown in Figure 2. This figure shows sequence traces of the two TCPs over a large range of bottleneck capacities. It is clear from these results that TCP/CM closely emulates a vanilla TCP.

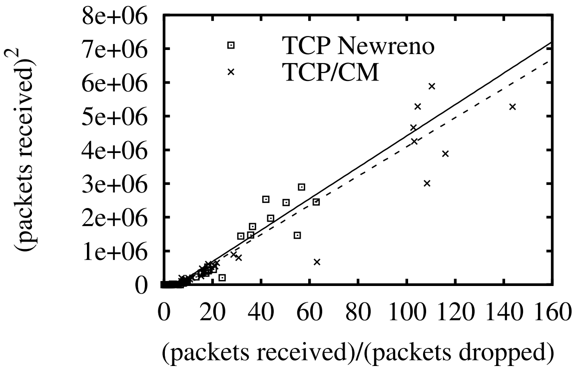

Figure 3: CM's rate control is TCP-friendly.

We now argue that our experimental data is consistent with a TCP-friendly congestion scheme. We performed a sequence of independent experiments with different bottleneck bandwidths for both TCP Newreno and TCP/CM. In any experiment, let n_r be the number of successfully received packets and n_d be the number of dropped packets, We show that the (n_r, n_d) data in our experiments is consistent with the $\lambda = K/\sqrt{p}$ TCP-friendliness relationship, where λ is the throughput, p the packet loss rate observed, and K is a constant that depends on the packet size and the round-trip time [18, 22, 23]

Let $n = n_r + n_d$ be the total number of transmitted packets. Clearly, $\lambda \propto n_r$ and $p \propto n_d/n$. Then, for the experimental data for TCP/CM to be consistent with the TCP-friendly relationship, $n_r = K/\sqrt{(n_d/n)}$, or $n_r^2 = K^2(n_r/n_d) + K^2$ must hold. That is, n_r^2 must be linear in n_r/n_d. Our measurements are consistent with this, as shown in Figure 3 which plots n_r^2 as a function of n_r/n_d for TCP/CM and TCP Newreno. The best-fit lines through these points have similar slopes for both protocols, since they have the same packet size and RTT.

2.2 Receiver Feedback

One of the fundamental requirements for stable end-to-end congestion control is receiver feedback. Without it, the sender would not be able to know if its current transmission rate is higher or lower than available capacity. Furthermore, this feedback about successfully received data and observed congestion needs to be communicated to the sender in some way. The sender's CM uses standard congestion indicators – packet losses and Explicit Congestion Notification (ECN) [9, 26] bits set by routers and echoed by the receiver.

We now address three issues: feedback frequency, feedback mechanism, and exponential aging to perform well when feedback frequency is lower than optimal.

2.2.1 Feedback frequency

TCP's feedback mechanism using ACKs provides the sender with feedback several times every round-trip, since the receiver generates an ACK for at least every other packet. In contrast, several streaming protocols are not reliable, and hence do not inform the sender of transmission status as frequently. Because the CM must function well across all applications, we first need to determine an appropriate feedback frequency.

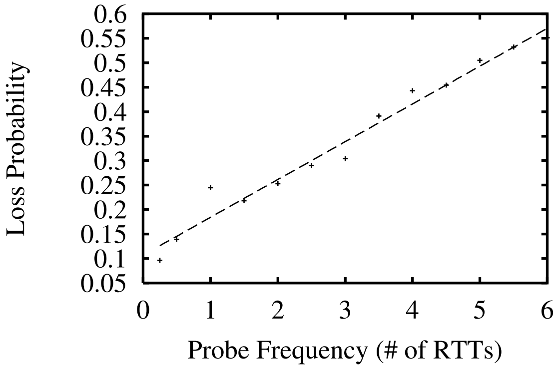

Figure 4: Variation of packet loss probability with feedback frequency

Figure 4 shows the performance of CM's congestion control at different feedback frequencies, by plotting packet loss probabilities. These results show that for good adaptation, feedback should be obtained at least twice every round-trip period.

2.2.2 Feedback mechanism

The CM uses two forms of feedback to adjust its rate and react to congestion: *implicit feedback* and *explicit feedback*. When the receiver application (or transport protocol) provides feedback to the sender application, implicit feedback is possible and no extra traffic need be generated. The sender application can now notify the CM about the number of transmitted and received bytes, if any losses occurred, and if any ECN information was received. For example, TCP over CM uses this method and the CM design for such situations does not require any changes at the receiver.

Unfortunately, not all applications are as considerate as TCP in providing frequent feedback. This moves us to incorporate an explicit feedback protocol in the CM architecture, with modifications to the receiver to respond to periodic probe messages from the CM sender and report loss or ECN information to the sender. This protocol should not generate too much traffic on its own and also be resilient to losses.

We now describe our lightweight probing protocol. The sender CM periodically sends probes to the receiver CM to elicit responses. The current frequency of these probes is twice every round-trip. Each probe includes an incrementing, unique sequence number. The receiver CM, on receiving this probe, responds with the numbers of the last probe it received (i.e., the current one), the last probe it responded to, and the number of packets received for each flow in between these two probes. Upon receipt of the response, the sender can estimate per-flow loss rates because it keeps track of the number of packets sent per flow, the total loss rate in the network, and update its round-trip time estimate. Because the sender maintains information about all probes since the last one for which a response was received, the protocol is robust to losses of probes or responses.

Figure 5 shows pseudocode for the probing protocol at the sender and receiver. For simplicity of exposition, we assume that the sender and receiver maintain information aggregated across all flows. The sender maintains an array `probe` indexed by the probe number. Each entry of the array is a structure with two elements: `timesent`, the time at which the probe was sent, and `nsent`, the number of bytes sent since the previous probe. It also has a variable `probeseqnum` which is the sequence number of the next probe to be sent.

2.2.3 Exponential aging

Figure 4 shows the problems with infrequent feedback, which does not allow the CM to adapt well to changing network conditions. The probing protocol described above addresses this by periodically eliciting receiver feedback. However, during times of congestion, probe messages or responses are lost, because of which the sender will not have an accurate estimate of the network state.

The first possible way to handle this is to clamp sender transmissions if more than one round-trip time elapses since the receipt of the last response. This is a conservative response and is the least likely to lead to instability. However, it comes at significant cost, because all flows stall until we hear a response once

Sending a probe to the receiver

```
message = <probe,probeseqnum>;
send(message);
probe(probeseqnum) = {probeseqnum, now, nsent};
nsent = 0;
probeseqnum = probeseqnum+1;
```

Responding to probe number `thisprobe`

```
message=<response,thisprobe,lastprobe,nrecd>;
send(message);
lastprobe = thisprobe;
nrecd = 0;
```

Sender action on receiving a response
`<response,thisprobe,lastprobe,nrecd>`

```
nsent = 0;
for(i=lastprobe+1; i<=thisprobe; i++) do
  nsent += probe(i).nsent;
end;
lossprob = nrecd/nsent;
Delete all entries in probe less than
thisprobe;
```

Figure 5: Sender and receiver side pseudocode for handling probes/responses

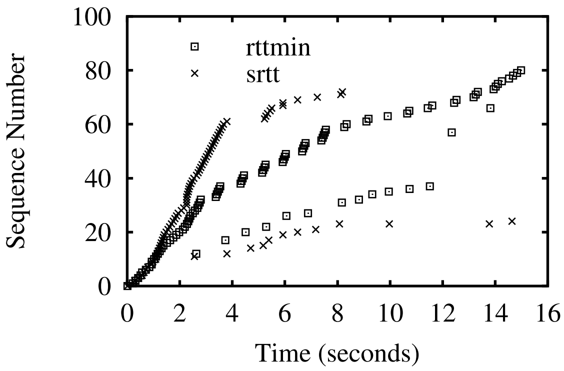

Figure 6: Sequence traces showing that exponential aging based on mean round-trip time is significantly inferior to using the minimum.

again, which could take quite a while longer because of the low probe frequency.

The second possible way to handle this is exactly the opposite: continue to transmit at the same rate until a response arrives, which may indicate that all packets were successfully received or that losses happened. The CM can now either increase or decrease its rate at this time. However, this is overly aggressive behavior because the sender transmits data in open-loop fashion for multiple round-trips without attention to the true state of the network. We are therefore forced to search for a compromise that avoids complete stalls, but yet transmits at prudent rates while in open-loop mode.

Our solution is a technique we call *exponential aging,* which is triggered when the CM does not receive a response to a probe message within a round-trip time. In each subsequent round-trip period starting from this point, the open-loop transmission rate is halved to its current value. This leads to an exponential fall-off in the rate as a function of time while in open-loop mode. It is not hard to see that this algorithm is stable because, in the worst case, each subsequent round-trip will also be congested. Such rate reduction would be the appropriate action if this were to happen, and it is easy to verify that the throughput-loss relationship has the same behavior as for TCP. Thus, exponential aging permits flows to continue transmitting data without stalls, albeit at lower rates.

An important parameter in exponential aging is the time intervals at which rate reduction is done, or the "half-life" of the algorithm. Our first choice was to use the sender's smoothed round-trip estimate for this. However, Figure 6 shows that this choice of half-life is too aggressive. This is because upon the onset of congestion, the sender's smoothed round-trip estimate often increases because of increased queueing delays, and rather than decay at an exponent governed by the true mean round-trip time, the decay occurs at a much slower rate. This leads to unstable behavior and induces a large number of losses.

Fortunately, there is an easy solution to this problem that significantly improves things by ensuring more conservative behavior. Because the problem is caused by the sender transmitting too rapidly and for too long in open-loop mode, we decrease the time-constant of exponential decay. The CM keeps track of the *minimum* of all its round-trip samples obtained over the duration of activity and decays the open-loop rate based on this. The improvements over using the mean round-trip estimate are apparent from Figure 6 which shows the sequence traces of transfers in each mode.

2.3 Better than Best-Effort Networks

Thus far, our design of the CM architecture assumes that the underlying network provides a best-effort service model. It is likely that the future Internet infrastructure will incorporate mechanisms such as differentiated services, integrated services, prioritization based on flow identifiers or port numbers, etc., and that a non-trivial fraction of Internet traffic will use these enhancements. In such situations, the previously described approach of aggregating congestion information based on the peer host address will in general be incorrect because different flows might experience different bandwidths and loss rates, depending on how routers treat them.

Fortunately, there is a solution to this problem based on *flow segregation*, where the API between a flow and the CM is keyed not by host address but by some combination of address, port numbers and flow identifiers. If an application knows beforehand that some of its flows will be treated differently from best-effort traffic, it can inform the CM of this. To function well in the absence of such explicit information, the CM incorporates a segregation algorithm to classify flows into aggregates based on loss rates and perceived receiver throughputs. Using a combination of the probing protocol and application hints, the CM obtains per-flow loss-rates and bandwidths, to segregate (and therefore also cluster) flows if their properties are very different. At this point, we have not implemented or experimented with this, but plan to do so soon.

2.4 Flow Scheduling

The CM internally schedules all requests using a Hierarchical Round Robin (HRR) Scheduler [17]. The scheduler apportions bandwidth among flows in proportion to pre-configured weights, as well as receiver hints. The scheduler is invoked whenever any application makes a call to the CM. If the scheduler can satisfy the next pending request based on the current bandwidth estimate of the CM at the present time, it performs an application callback informing it about the appropriate number of bytes allowed. Otherwise, it notifies the application at a future point in time based on the minimum number of bytes requested by the application, and the sending rate. The `app_notify()` call is described in greater detail in Section 3.

The scheduler as currently implemented performs only bandwidth allocation, and does not use delay bounds in its scheduling. This is adequate for flows which use TCP. We are however investigating other mechanisms [33] to provide combined bandwidth and delay guarantees.

Figure 7 shows flows starting at different times eventually achieving the same

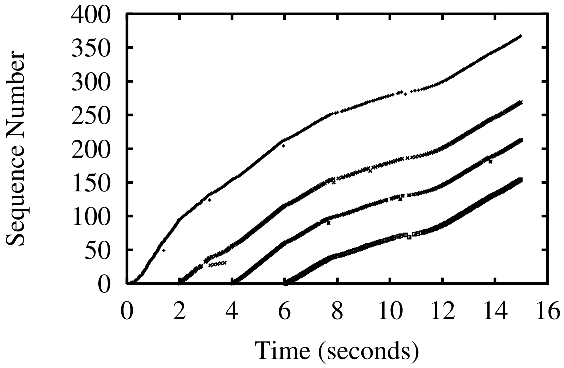

Figure 7: The CM scheduler apportions bandwidth well between simultaneous flows.

rate allocation from the scheduler.

3 The CM API

Network congestion is a property of the path traversed by flows between a sender and receiver. The CM frees applications from having to maintain information about the state of congestion and available bandwidth along any path. Using its API, flows can determine their share of the available bandwidth, request and have their data transmissions scheduled, inform the CM about successful transmissions, and be informed when the CM's estimate of path bandwidth changes.

3.1 Design Rationale

Rather than simply present the CM API without justification, we motivate our design choices and discuss the API in terms of four guiding principles.

1. *Put the application in control:* While the CM decides the rate at which each application flow can transmit data, it follows the end-to-end argument [29] and puts the application in firm control of two critical decisions: (i) deciding *what* to transmit at each point in time, and (ii) deciding the relative fraction of available bandwidth to allocate to each flow. To achieve this, the CM does not buffer any application data; instead, it allows applications the opportunity to adapt to unexpected network changes at the last possible instant. This design decision to not buffer any data is a direct consequence of the Application Level Framing (ALF) [6] approach to protocol design, and leads to the API described below.

If the CM were to queue data and eventually transmit it at some rate, the sending API would consist simply of a `cm_send()` call, much like the BSD Sockets API [31]. However, this would preclude applications from "pulling out" and repacketizing data upon learning about any rate change. Thus, we decide to design a non-blocking request/callback/notify API. Here, an application wishing to send `nsend` bytes calls `cm_request(nsend)`. After some time, depending on the rate, the CM invokes an application callback using `app_notify(nsend)`, which is a grant for the application to send up to `nsend` bytes. The application is expected to transmit up to `nsend` bytes soon after this, and it does not matter if those bytes are different from the ones for which the original request was made. In addition, the application notifies the CM using `cm_notify(nsent)` telling it that `nsent` bytes were transmitted so it can update its internal state.

To learn about per-flow available bandwidth and the round-trip time, applications use the CM's `cm_query(&rate, &srtt)` call, which fills in the desired quantities.

2. *Accommodate traffic heterogeneity:* The CM should benefit a variety of traffic types, including TCP bulk transfers and short transactions, real-time flows that can transmit at a continuum of rates, and layered streams that can transmit only at discrete rate intervals.

3. *Accommodate application heterogeneity:* The design of the CM API should not force a particular application style; rather, the API should be flexible enough to accommodate different styles. In particular, the API should accommodate two common styles of transmitters: the *asynchronous* style and the *synchronous* style.

Asynchronous transmitters do not transmit based on a periodic clock, but do so triggered by asynchronous events like file reads or captured frames. For these transmitters that typically have bytes ready to be transmitted, the request/callback/notify API described above is appropriate because their transmissions are scheduled by the CM. On the other hand, synchronous transmitters implement timer-driven, and would use the CM to adapt the frequency of their internal timers. Such applications will benefit from a CM callback informing them of changes in rates, for which we provide the `change_rate(newrate)` function. Thus, there are two callback functions implemented by the CM: `app_notify(nsend)` in response to a previous request call, and `change_rate(newrate)` whenever a flow's share of the available rate changes. This second method is provided for both types of transmitters, because the knowledge of sustainable rate is useful for asynchronous applications as well; e.g., an asynchronous Web server disseminating images using TCP could use `app_notify()` to schedule its transmissions and `change_rate()` to decide whether to send a low-resolution or high-resolution image.

4. *Learn from the application:* The API includes functions that applications

can use to provide feedback to the CM. In addition to `cm_notify()` to inform the CM on each transmission, they can use `cm_update(nrecd, duration, loss_occurred, rtt)` call to inform the CM that `nrecd` bytes were received over `duration` seconds, that the observed RTT was `rtt`, and whether any losses occurred. The feedback could be through ACKs as in TCP, through RTCP in the case of real-time applications, or through any other protocol. The CM uses this as a hint to internally update its sustainable sending rate and round-trip time estimates.

An application calls `cm_close()` when a flow is terminated allowing the CM to destroy the internal state associated with that flow and repartition available bandwidth.

The CM API is summarized in Figure 8.

3.2 Using the API

In this section, we describe how applications and transport protocols use the CM API. We focus on two applications—a Web server disseminating objects using TCP and an adaptive audio server that disseminates objects using a user-level transport protocol over UDP.

3.2.1 Web server over TCP

Using HTTP[1], clients request index files and sets of objects from the server. The CM enables the sender to decide what fraction of the bandwidth to use for what flow, based on hints from the receiver. It also helps the sender to choose between multiple representations that are available for some objects, e.g., low-, medium- and high-resolution images, for the best application performance.

Using the receiver CM API, the client expresses its relative interest in the n objects with a vector of tuples of the form $[o_1 : r_1, o_2 : r_2, \ldots, o_n : r_n]$, where o_i is the ith object and r_i the relative fraction of the available bandwidth to allocate to that stream. The sender takes this into account to apportion bandwidth while transmitting these objects. This is similar to the WebTP [12] protocol.

Multiple representations of different sizes exist for several of these objects. The sender uses the `cm_query()` call and the `change_rate()` handler to adapt to changing available bandwidths (tracked by the CM) and pick the representation that maximizes receiver quality without incurring high latency. We are currently extending the HTTP content negotiation protocol [15] to incorporate these ideas.

[1]It really does not matter what version of HTTP, but as we will see in Section 5, the use of persistent connections in P-HTTP has some drawbacks.

Data Structures:

```
struct cm_entry {
        addr dst;
        double rate;
        double mean_rtt;
        double rttvar;
};
typedef int cm_id;
```

Query:

```
void cm_query(cm_entry *entry, addr dst);
```

Control:

```
cm_id cm_open(addr src, addr dst);
void cm_request(cm_id id, int nbytes,
                int minbytes,
                double latency);
void cm_notify(cm_id id, int nsent);
void cm_update(cm_id id, int nrecd,
               bool loss_occured,
               double rtt);
void cm_close(cm_id id);
```

Application callback:

```
void app_notify(int nallowed);
void change_rate(double rate);
```

Figure 8: Data structures and functions for the sender-side CM API

The Web server uses TCP to disseminate data, which in turn uses the CM to perform congestion management; thus, the TCP/CM[2] now only performs loss recovery and connection management. We now outline how TCP congestion control can be written as a CM application.

Normally, TCP's congestion management keeps track of a congestion window on a per-connection basis. When ACKs arrive, TCP updates the congestion window and transmits data if its congestion window allows it, and when it detects losses, the window is reduced by at least a factor of two. To use the CM,

[2]This is supposed to be read as: "TCP over CM"

we modify TCP to call cm_open() when it establishes a connection. When nsend bytes of data arrive from the application (e.g., Web server), TCP/CM calls cm_request(nsend) to schedule the transmission of nsend bytes of data. When an ACK arrives from the network acknowledging nrecd bytes of data, TCP/CM calls cm_update(nrecd) as a useful hint to update the congestion state in the CM. It then calls cm_request() if the receiver-advertised flow control window has opened up and there is more data queued for transmission.

When the CM decides to service TCP/CM's request, it performs a callback using app_notify() to the TCP/CM send routine that accepts a parameter indicating the maximum amount of data it is allowed to transmit. The TCP send routine then transmits the minimum f the flow control window and the amount allowed by the CM in the callback. Immediately after transmitting this data, TCP/CM uses the cm_notify() call to update CM with the actual amount of data sent. This could be smaller than the amount permitted, and may even be zero at some points in time, e.g., when the TCP/CM sender performs silly window syndrome avoidance [35].

Notice that we have eliminated the need for tracking and reacting to congestion in TCP/CM, because proper congestion behavior is ensured by the CM and its callback-based transmission API. Notice also that duplicate ACK, timeout based loss recovery remain unchanged and end-to-end flow control based on advertised windows remain unchanged. In our implementation and experiments, we use the Newreno flavor of TCP/CM [14] because it performs better than TCP Reno under most conditions. The result is that the CM permits an ensemble of TCP connections to behave in a manner less deleterious to the health of the network.

3.2.2 Audio server for layered audio streams

Many Internet audio servers support a variety of audio sampling rates and audio encodings. Fundamentally, the purpose of supporting this selection is to provide the client with a tradeoff of quality for network bandwidth. Typically, the end user is forced to manually select the most appropriate encoding for the current network conditions. The use of the CM enables the audio server to correctly adapt its choice of audio encoding to the congestion state of the network.

When requested to transmit audio to a client, the server first performs a cm_open() call. It then uses the cm_query() call to determine how quickly it may transmit data. It then begins transmitting audio at the highest quality encoding that does not exceed the rate returned by cm_query(). Immediately after transmitting data, the server uses cm_notify() to inform CM of the amount sent. Although some real-time servers solicit feedback about network conditions from their clients, many do not. We have chosen to model a server which does not monitor network connectivity. As a result, the congestion feedback is provided by the CM's probing protocol. If the CM identifies a change in the available bandwidth upon the arrival of a probe response, it notifies the audio server of this change using the `change_rate()` callback. The audio server's implementation of `change_rate()` can then adjust its data encoding using the new rate information. Via these simple interactions with the CM, the audio server becomes capable of automatically adjusting audio quality to reflect the quality of client-server communications.

4 Application Performance

We have implemented the CM in the VINT ns-2 framework. We have also implemented a TCP agent and an audio server application to use the CM.

4.1 Web Performance

This section presents the results of experiments with a simple Web-like workload consisting of four concurrent connections with significant TCP and constant bit-rate cross-traffic in a network with a 1 Mbps bottleneck link and round-trip propagation delay of 120ms. Our results, shown in Figure 9, demonstrate that the CM ensures proper behavior in the face of congestion and improves the consistency of application performance.

Using TCP Newreno, the performance of the four connections varies between 120 Kbps and 213 Kbps, nearly a 100% difference in transfer time between the fastest and slowest connections! This is because of the lack of shared state learning and the competitive, rather than cooperative congestion control for the ensemble of connections. In contrast, the four connections using TCP/CM progress at very similar, consistent rates sharing bandwidth equitably. All four connections achieve throughputs of 120 Kbps, without incurring too many losses along the way. Thus, the CM enables the ensemble of connections to effectively share bandwidth and learn from each other about the network.

We note that the aggregate throughput obtained by TCP/CM ($\approx$ 500 Kbps) is lower than the aggregate throughput obtained by independent TCP Newreno connections ($\approx$ 650 Kbps). This is hardly surprising because the CM forces the concurrent connections to behave as one from the point of congestion control, whereas the effective decrease and increase coefficients for the independent connections are significantly larger than for a single TCP. The CM does indeed ensure that a group of connections between the same hosts behaves in a socially proper way. The observed throughput degradation, while unfortunate, is a consequence of cor-

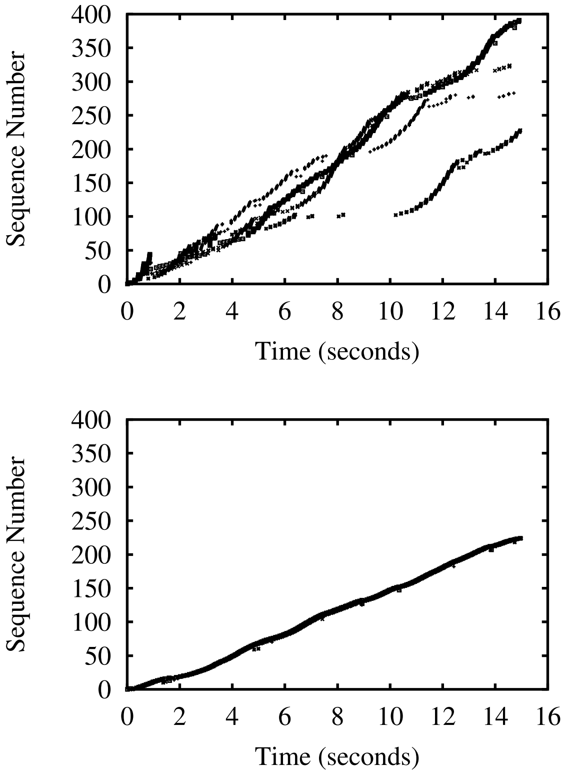

Figure 9: The top chart shows sequence traces for a Web-like workload using 4 concurrent TCP Newreno connections. The performance of these transfers is highly variable and inconsistent. The bottom graph shows the same workload over TCP/CM, demonstrating the consistent and predictable performance of a Web workload using the CM - the four connections are indistinguishable!.

rect congestion control. But TCP applications do directly benefit in significant ways: they obtain improved performance consistency and predictability, which is a definite incentive for adoption.

4.2 Layered Audio Performance

This section discusses the results of experiments testing the interactions of adaptive audio applications using CM with TCP traffic. Our experiment consisted of performing test transfers against competing TCP and constant bit rate cross traffic across a bottleneck link of 0.5 Mbps and a round-trip delay of 120ms. The test traffic consisted of a single audio transfer using CM, a single TCP/CM transfer (on the same end-host) and a TCP Newreno transfer. The expected and desired result is that the combined bandwidth of the TCP/CM and audio transfer would equal the bandwidth of the TCP Newreno transfer. In addition, the audio transfer should choose an encoding that most closely matched it to the bandwidth of the TCP/CM transfer. In our experiment, the audio application choose amongst encodings of 10, 20, 40, 80, 160 and 320 Kbps. It always performed transmissions of 1KB packets.

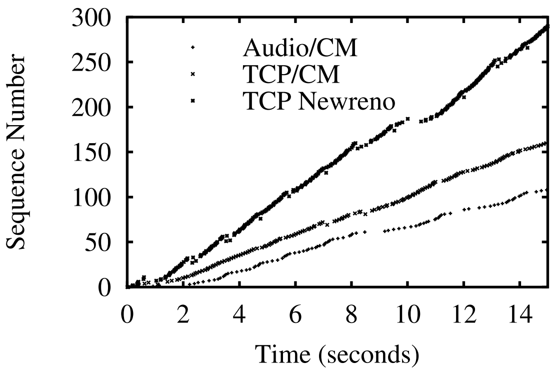

Figure 10: Performance of an adaptive audio application

The results of the experiment, shown in Figure 10, confirm that the CM, TCP/CM and adaptive audio perform as expected. The TCP Newreno transfer obtained approximately 150Kbps. The combination of the audio, at 50Kbps, and the TCP/CM, at 85Kbps, was quite close to the traditional TCP performance. The audio primarily used the 80Kbps encoding, occasionally switching to the 40 and 160Kbps encodings. Given the fact that the audio application had to deal with such coarse grained adaptation, its performance was sufficiently similar to the associated TCP/CM connection. From these results, it is clear that the CM allows streaming applications to perform the correct adaptation to congestion in the network.

5 Related Work

Most Web sessions today use multiple concurrent TCP connections. Each connection wastefully performs slow start irrespective of whether other connections are currently active to the same client. Furthermore, upon experiencing congestion along the path to a client, only a subset of the connections (the ones that experience losses) reduce their window. The resulting multiplicative decrease factor for the ensemble of connections is often larger than 0.5 [1], the value used by individual TCPs[3]. This is unfair relative to other clients that use fewer connections,

[3]If there are n concurrent connections with equal windows and m of them experience a loss, the decrease factor is $(1 - m/2n)$.

and worse, will lead to instability in a network where most clients operate in this fashion.

There has also been some recent work in developing application-specific congestion control schemes for real-time multimedia streams. We discuss two classes of solutions to the unicast congestion control problem—*application-level solutions* and *transport-level solutions*.

5.1 Application-level Solutions

Application-level solutions for Web transport multiplex several logically distinct streams onto a single transport (TCP) connection to overcome the adverse effects of independent competing TCP transfers. Examples of this include Persistent-connection HTTP (P-HTTP, part of HTTP/1.1), which is application-specific, and the Session Control Protocol (SCP) [30] and the MUX protocol [11], which are not tied to HTTP.

There are several drawbacks with this class of solutions.

- *Architectural problems:* These solutions are application-specific and attempt to *avoid* the poor congestion management support provided by protocol stacks today. However, congestion is a property of the network path and the right point in the system to manage it is inside the protocol stack, not at the application. If the right support is provided by the system, the need for such solutions can be eliminated.

- *Application-specificity:* These solutions require each class of applications (Web, real-time streams, file transfers, etc.) to reimplement much of the same machinery, or else force them to use protocols like TCP that are not well-suited to the task at hand.

- *Undesirable coupling:* These solutions typically multiplex logically distinct streams onto a single byte-stream abstraction. If packets belonging to one of the streams is lost, another stream could stall even if none of its packets are lost. This is because of the in-order delivery provided by TCP, which forces a linear order over all the transferred bytes when only a partial order is desired. This is a violation of the ALF principle [6], which states that independent Application Data Units (ADUs) should be independently processible by receivers independent of the order in which they were received.

The WebTP proposal [12] aims to develop a receiver-oriented approach to handling concurrent Web transactions. This includes maintaining congestion parameters at the receivers, which makes it easy to incorporate our equivalent receiver hints for bandwidth partitioning between flows. On the other hand, because the eventual transmissions are performed by the sender, we believe that the CM design is sound and also achieves some of WebTP's benefits. Like the CM, WebTP has also been heavily motivated by ALF as a protocol design principle.

There has been some recent work in developing congestion control protocols for real-time multimedia and streaming applications. Much of this work has been in the context of multicast video (e.g., IVS [3], RLM [20], etc.). There have also been numerous recent congestion control proposals for various reliable multicast applications (for a survey, see [28]). In contrast to these efforts which are application-specific, our aim is to develop a substrate that manages congestion and allows applications to implement their own adaptation policies. Perhaps, closer in spirit to our goal is the RAP protocol [27], which is a rate-based congestion control scheme intended for streaming applications. While the internal algorithms of the CM are in fact rate-based, its architecture is radically different from RAP. In particular, it is independent of the transport protocol and permits information to be shared between transports in a coherent manner (e.g., it integrates congestion management across concurrent rate-based audio flows and window-based TCP flows).

5.2 Transport-level solutions

Motivated in part by the drawbacks of the above solutions and by the desire to improve Web transfer performance, various researchers have proposed modifications to TCP itself [1, 24, 34]. Although these approaches do solve most of the problems associated with the Web scenario, they are transport-specific. They only handle TCP transfers, and applications that use other protocols cannot take advantage of them. Prominent examples of such applications include various real-time streaming media services.

Recently, a transport protocol for heterogeneous packet flows (HPF) has been described in [19]. A key difference between the CM and HPF is that the CM integrates congestion management across an ensemble of flows and provides a different adaptation API, while HPF does not consider the interactions between concurrent active flows.

6 Concluding Remarks

In this paper, we presented an end-system architecture centered around a Congestion Manager (CM) that ensures proper congestion behavior and allows applications to easily adapt to network congestion.

The CM incorporates a rate-based control protocol, a lightweight loss-resilient protocol for receiver feedback, and an exponential aging scheme to regulate transmissions when feedback is infrequent. It provides a simple API for applications to adapt conveniently to network congestion and varying bandwidth availability. It enables multiple concurrent flows to cooperate rather than compete for network resources, performing the function of a trusted intermediary for these resources.

We have simulated TCP and an adaptive audio application on top of the CM. Our results show that while an ensemble of vanilla TCP Newreno connections has almost a 100% variation between the slowest and fastest connections, an ensemble of TCP/CM connections with the same bottleneck bandwidth and cross-traffic shares bandwidth equally and consistently with little variation between the rates of different connections. Furthermore, the ensemble of CM-modulated flows displays social and stable network behavior while achieving this. The adaptive audio application is able to use the CM API to transmit an encoding that closely matches the varying available bandwidth, without having to constantly probe the network for excess capacity on its own. These results demonstrate that the CM ensures stable network behavior, while improving application performance in several ways.

References

[1] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., STEMM, M., AND KATZ, R. TCP Behavior of a Busy Web Server: Analysis and Improvements. In *Proc. IEEE INFOCOM* (Mar. 1998).

[2] BENNETT, J., AND ZHANG, H. Hierarchical Packet Fair Queueing Algorithms. In *Proc. ACM SIGCOMM* (Aug. 1996).

[3] BOLOT, J., TURLETTI, T., AND WAKEMAN, I. Scalable Feedback for Multicast Video Distribution in the Internet. In *Proc. ACM SIGCOMM* (London, England, Aug 1994).

[4] CHIU, D.-M., AND JAIN, R. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems 17* (1989), 1–14.

[5] CLARK, D., SHENKER, S., AND ZHANG, L. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms. In *Proc. ACM SIGCOMM* (August 1992).

[6] CLARK, D., AND TENNENHOUSE, D. Architectural Consideration for a New Generation of Protocols. In *Proc. ACM SIGCOMM* (September 1990).

[7] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulations of a Fair-Queueing Algorithm. *Internetworking: Research and Experience V*, 17 (1990), 3–26.

[8] FERRARI, D., AND VERMA, D. A scheme for real-time communication services in wide-area networks. *IEEE Journal on Selected Areas in Communications 8*, 3 (Apr. 1990), 368–379.

[9] FLOYD, S. TCP and Explicit Congestion Notification. *Computer Communications Review 24*, 5 (Oct. 1994).

[10] FLOYD, S., AND FALL, K. Router Mechanisms to Support End-to-End Congestion Control. Tech. rep., LBNL, 1997.

[11] GETTYS, J. Mux protocol specification, wd-mux-961023. http:// www.w3.org/ pub/ WWW/ Protocols/ MUX/ WD-mux-961023.html, 1996.

[12] GUPTA, R. *WebTP: A User-Centric Receiver-Driven Web Transport Protocol*. University of California, Berkeley, Berkeley, CA, 1998.

[13] HASHEM, E. Analysis of Random Drop for Gateway Congestion Control. Tech. Rep. LCS TR-465, Laboratory for Computer Science, MIT, 1989.

[14] HOE, J. C. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proc. ACM SIGCOMM '96* (Aug. 1996).

[15] HOLTMAN, K. *Transparent Content Negotiation in HTTP*. RFC, March 1998. RFC-2295.

[16] JACOBSON, V. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88* (August 1988).

[17] KALMANEK, C. R., KANAKIA, H., AND KESHAV, S. Rate Controlled Servers for Very High-Speed Networks. In *Proceedings of the IEEE Conference on Global Communications* (Dec 1990).

[18] LAKSHMAN, T. V., MADHOW, U., AND SUTER, B. Window-based Error Recovery and Flow Control with a Slow Acknowledgement Channel: A study of TCP/IP Performance. In *Proc. Infocom 97* (April 1997).

[19] LI, J., DWYER, D., AND BHARGHAVAN, V. A Transport Protocol for Heterogeneous Packet Flows. In *Proc. IEEE INFOCOM* (Mar. 1999).

[20] MCCANNE, S., JACOBSON, V., AND VETTERLI, M. Receiver-driven Layered Multicast. In *Proc ACM SIGCOMM* (Aug. 1996).

[21] ns-2 Network Simulator. http://www-mash.cs.berkeley.edu/ns/, 1998.

[22] OTT, T., KEMPERMAN, J., AND MATHIS, M. The Stationary Distribution of Ideal TCP Congestion Avoidance, 1996.

[23] PADHYE, J., FIROIU, V., TOWSLEY, D., AND KUROSE, J. Modeling TCP throughput: A Simple Model and its Empirical Validation. In *Proc. ACM SIGCOMM* (Sept. 1998).

[24] PADMANABHAN, V. N. *Addressing the Challenges of Web Data Transport*. PhD thesis, Univ. of California, Berkeley, 1998. In preparation.

[25] PAREKH, A. K., AND GALLAGER, R. G. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking 1*, 3 (June 1993), 344–357.

[26] RAMAKRISHNAN, K., AND FLOYD, S. A Proposal to Add Explicit Congestion Notification (ECN) to IPv6 and to TCP. Internet Draft draft-kksjf-ecn-00.txt, Nov. 1997. Work in progress.

[27] REJAIE, R., HANDLEY, M., AND ESTRIN, D. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. To appear in Proc. Infocom 99.

[28] Reliable Multicast Research Group. http://www.east.isi.edu/RMRG/, 1997.

[29] SALTZER, J., REED, D., AND CLARK, D. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems 2* (Nov 1984), 277–288.

[30] SPERO, S. Session control protocol (scp). http:// www.w3.org/ pub/ WWW/ Protocols/ HTTP-NG/ http-ng-scp.html, 1996.

[31] STEVENS, W. R. *UNIX Network Programming*. Addison-Wesley, Reading, MA, 1992.

[32] STEVENS, W. R. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*, Jan 1997. RFC-2001.

[33] STOICA, I., AND ZHANG, H. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Services. In *Proc. ACM SIGCOMM '97* (1997).

[34] TOUCH, J. *TCP Control Block Interdependence*. RFC, April 1997. RFC-2140.

[35] WRIGHT, G., AND STEVENS, W. R. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, Reading, MA, Jan 1995.

[36] ZHANG, L. A New Architecture for Packet Switching Network Protocols. Tech. Rep. LCS TR-455, Laboratory for Computer Science, MIT, Aug. 1989.

[37] ZHANG, L., DEERING, S., ESTRIN, D., SHENKER, S., AND ZAPPALA, D. RSVP: A new resource ReSerVation Protocol. *IEEE Network Magazine* (Sept. 1993), 8–18.