An Integrated Congestion Management Architecture for Internet Hosts

Hari Balakrishnan, Hariharan S. Rahul
M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
{hari, rahul}@lcs.mit.edu

Srinivasan Seshan
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
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 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 window-based control algorithm, a scheduler to regulate transmissions, and a lightweight protocol to elicit feedback from receivers.

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.

1 Introduction

The success of the Internet to date has been in large part due to the sound principles of additive-increase/multiplicative-decrease (AIMD) congestion control [4] on which its dominant transport protocol, TCP [15, 30], 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, and concurrent connections between the same pair of hosts compete rather than cooperate with each other for scarce resources. Second, some commercial products "accelerate" Web downloads 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. Furthermore, there are applications such as teleconferencing where multiple concurrent streams co-exist (e.g., audio, video, whiteboards, text), that will benefit from efficient multiplexing and sharing of bandwidth.

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. While this might sound overly pessimistic, 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 (e.g., [7, 22, 8, 5, 36, 2]), we focus on developing an architecture at the end-hosts to:

- Enable efficient multiplexing of concurrent flows, en-

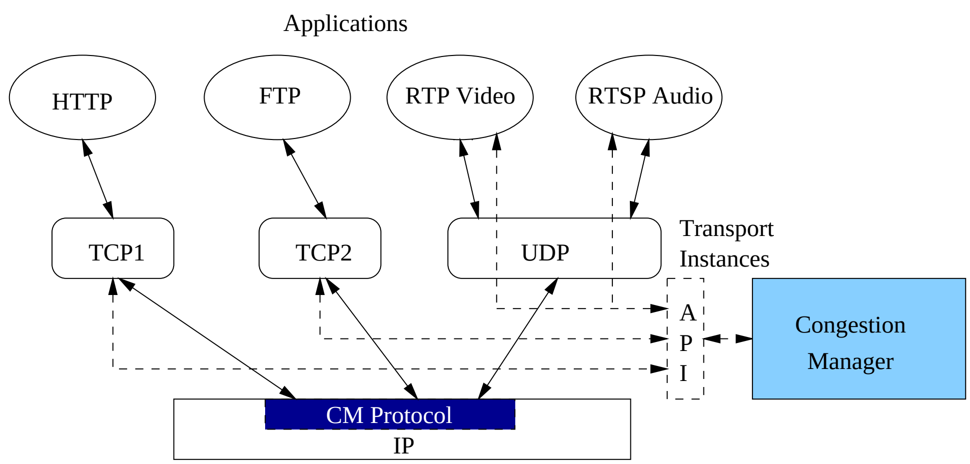

Figure 1: New sender protocol stack with the Congestion Manager.

suring proper and stable congestion behavior using AIMD principles.

- Enable applications and transport protocols to adapt to network congestion and varying bandwidth by providing an adaptation API.

The resulting framework is independent of specific applications and transport protocols, 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 appropriately 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 protocols perform transmissions only with the consent of the CM.

Internally, the CM ensures stable network behavior by the sender because it reacts to congestion, carefully probes for spare bandwidth by permitting applications to send at a higher rate, 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 4, where we justify them using ns-based [20] simulations 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 [26] 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 5 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.

The following are the main contributions of this paper:

- **CM algorithms and protocol.** 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 window-based AIMD scheme with traffic shaping, (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, and (iv) a scheduler to apportion bandwidth to flows.

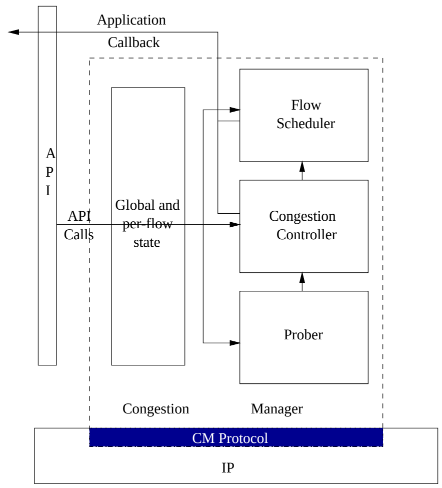

Figure 2: Internal organization of the Congestion Manager at the sender.

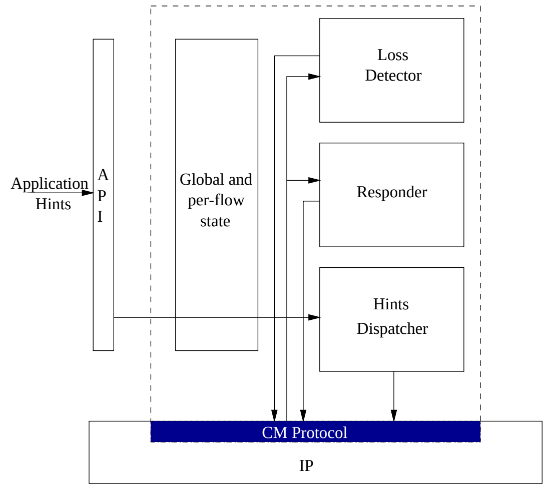

Figure 3: Internal organization of the Congestion Manager at the receiver.

- **CM adaptation API.** An API for applications and transport protocols to learn about and adapt 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 of TCP transfers, and enables applications such as audio servers to effectively transmit the best among several available source encodings.

2 CM Architecture

In this section, we give a brief description of the overall CM architecture. The CM has two modules, one at the data sender and the other at the receiver. The sender orchestrates data transmissions, while the receiver maintains loss statistics and responds to occasional sender probes. Most of the complexity is at the sender.

Figure 2 shows a schematic description of the components of the CM at the sender. The *Congestion Controller* adjusts the aggregate transmission rate between sender and receiver based on its estimate of congestion in the network. It obtains feedback about its past transmissions from applications themselves, as well as from the *Prober*, which sends periodic probes to the receiver CM. The *Flow Scheduler* apportions available bandwidth amongst the different flows and notifies applications when they are permitted to send data. Applications schedule transmissions by invoking scheduler functions. The CM components communicate using well-defined interfaces; this allows us to change any one of them without affecting the rest of the system.

To communicate with the receiver CM module, the CM uses a protocol that attaches a CM header to outgoing packets. This protocol is used to determine if the receiver is CM-enabled via a simple two-way handshake, and if so, exchange information about losses and other interesting statistics. If the receiver is not CM-enabled, the CM sender does not attach this header. However, it continues to implement its internal algorithms to enable efficient multiplexing, proper congestion behavior, and application adaptation.

Figure 3 depicts the CM components at the receiver. Application hints for apportioning bandwidth are communicated by the API to the *Hints Dispatcher*, which transmits them to the CM sender. The receiver CM module strips the CM header and dispatches the packet to the receiver application. The *Loss Detector* maintains loss statistics based on information in the CM header and informs the sender when it detects congestion. The *Responder* maintains statistics of the number of bytes received by each flow and participates in the probing protocol.

The ability to function even when the receiver is not CM-enabled is ensured by the CM protocol, and the CM algorithms at the sender often function with application feedback (as opposed to CM protocol feedback from the receiver CM module). Thus, while the full benefits of the architecture are observed when both senders and receivers are modified, there are substantial benefits even when only the sender is CM-enabled. This is especially true for those applications that have a feedback mechanism (e.g., TCP) and use the CM API to inform the sender.

The rest of this paper describes the CM API, its algorithms and protocol, and simulation results of some CM applications.

3 The CM API

Using the CM API, flows can determine their share of the available bandwidth, request and schedule their transmissions, inform the CM about successful transmissions, and be informed when the CM's estimate of path bandwidth changes. Thus, the CM frees applications from having to maintain information about the state of congestion and available bandwidth along any path.

3.1 Design Rationale

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 [26] 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 Application Level Framing (ALF) [6], and leads to the following API.

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 [29]. 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 API. Here, an application that wishes to send data invokes `cm_request(id)`. (The `id` is obtained by the application using the `cm_open(dst)` call, where `dst` is the destination address.) After some time, depending on the past transmissions and allowed rate, the CM triggers an application callback using `cmapp_send()`, which is a grant for the application to send up to `mtu` bytes. The application can transmit any `mtu` (or less) bytes soon after this, and it does not matter if those bytes are different from the ones for which the original request was made. The application uses `cm_mtu(id)` to get the path MTU (Maximum Transmission Unit), which can either be statically configured or discovered using path MTU discovery [19].

(In our original design, `cm_request() and cmapp_send()` used the number of requested and permitted bytes as arguments. This would have potentially given applications greater control, at the expense of complicating the scheduler and making traffic shaping harder to accomplish. Ultimately, because we could not see any clear benefits of this additional control for application writers, we eliminated these arguments. We are grateful to Steve McCanne for convincing us to pursue this better alternative.)

Our initial design only allowed for the ALF-oriented API based on callbacks described above. However, early experience and discussions convinced us that not all applications would want to use ALF, and that a conventional buffered send mode was worth supporting as well. This is straightforward; such streams, invoke `cm_send(id, data, length)` and the CM buffers data for eventual transmission.

To learn about per-flow available bandwidth and the round-trip time, applications use the CM's `cm_query(id, &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 API described above is appropriate because their transmissions are scheduled by the CM. On the other hand, synchronous transmitters are timer-driven and would use the CM to adapt the frequency of their internal timers and the amount of data transmitted at each timer event. Such applications can use the `cmapp_update(rate, srtt)` callback function informing them of changes in rates. Thus, there are two callback functions implemented by the CM: `cmapp_send()` in response to a previous request call, and `cmapp_update()` 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 `cmapp_send()` to schedule its transmissions and `cmapp_update()` 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. They can use `cm_update(id, nsent, nrecd, lossmode, rtt)` call to inform the CM that `nsent` bytes were sent of which `nrecd` were received, that the loss event was `PERSISTENT` (e.g., a TCP timeout), `TRANSIENT` (e.g., TCP duplicate acknowledgments), or `ECN` (on Explicit Congestion Notification), and that the observed RTT sample was `rtt`. The feedback could be through ACKs as in TCP, through RTCP [27] in the case of real-time applications, or through any other protocol. The CM uses this information to update its congestion window and round-trip time estimates.

The CM also exposes a notification function, `cm_notify()` that must be invoked by the IP output routine at the sender whenever any bytes are sent for a flow. This allows the CM to update its estimate of the number of outstanding bytes for the flow.

At the receiver, the CM can learn from application hints about the relative proportion of the available bandwidth to allocate to different flows. This allows receivers to express their preference for certain types of traffic over others, for example, images over text. We are currently completing the details of this part of the API.

An application calls `cm_close(id)` when a flow is terminated allowing the CM to destroy the internal state associated with that flow and repartition available bandwidth. If an application forgets to invoke `cm_close()`, its associated flow state is cleaned up by the CM after a timeout.

The CM API is summarized in Figure 4.

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.

[1]It really does not matter what version of HTTP, but as we will see in Section 6, the use of persistent connections in P-HTTP has some drawbacks.

```
typedef int cmid_t;
```

Query

```
void cm_query(cmid_t id, double *rate, double *srtt);
```

Control

```
cmid_t cm_open(addr dst);
int    cm_mtu(cmid_t id);
void   cm_request(cmid_t id);
void   cm_notify(addr dst, int nsent);
void   cm_update(cmid_t id, int nrecd,
                 int nlost, int lossmode,
                 double rtt);
void   cm_close(cmid_t id);
```

Buffered transmission

```
void   cm_send(cmid_t id, char *data, int len);
```

Application callback

```
void   cmapp_send();
void   cmapp_update(double rate, double srtt);
```

Figure 4: Data structures and functions for the sender-side CM API.

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 [34] protocol.

Multiple representations of different sizes exist for several of these objects. The sender uses the `cm_query()` call and the `cmapp_update()` 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 [14] to incorporate these ideas.

The Web server uses TCP to disseminate data, which in turn uses the CM to perform congestion management; thus, 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, we modify TCP to call `cm_open()` when it establishes a connection. When data arrives from the application (e.g., Web server), TCP/CM calls `cm_request()` to schedule their transmission. When an ACK arrives from the network acknowledging `nrecd` bytes of data, TCP/CM calls `cm_update()` 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 `cmapp_send()` to the TCP/CM send routine, allowing for up to 1 MTU's worth of data to be sent, provided the receiver-advertised window permits transmission. When the IP output routine sends this data, it calls `cm_notify()` to update the CM's estimate of the number of outstanding bytes. `nsent` could be smaller than the amount permitted.

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 and timeout based loss recovery remain unchanged, as does end-to-end flow control based on advertised windows. In our implementation and experiments, we use the Newreno variant of TCP/CM [13] 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 than before.

[2] "TCP over CM"

3.2.2 Audio server for layered audio streams

Many Internet audio servers support a variety of audio sampling rates and audio encodings to allow the client to trade-off 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 server to *automatically* adapt its choice of audio encoding to the congestion state of the network.

When requested to transmit audio to a client, the server calls `cm_open()` and uses `cm_query()` to determine how soon it may transmit data. It then begins transmitting audio at the highest quality encoding that does not exceed the rate returned by `cm_query()`. Although some streaming servers solicit feedback about network conditions from their clients, many do not. For servers that do not, feedback is obtained using the CM's probing protocol (Section 4.2.2). 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 `cmapp_update()` callback. The audio server's implementation of `cmapp_update()` then adjusts its data encoding using the new rate information. Via these simple interactions with the CM, the audio server can automatically adjust audio quality to reflect the quality of reception. Note that the CM does not shape such traffic by forcing transmissions at particular times; instead, it shapes all other traffic around those events.

4 CM Algorithms and Protocol

In this section, we present the CM's internal algorithms and protocols. We first present the architecture of the CM at the sender. Then, we describe the corresponding organization of the CM at the receiver. We conclude this section by discussing issues that arise in non-best-effort networks, including ones with service differentiation and reservations.

4.1 Stable Congestion Control

One of the key features of the CM is that it ensures proper congestion behavior of an ensemble of flows by sharing congestion information between them. This implies that its mechanisms for reacting to network congestion and probing for spare capacity must be sound and robust. An attractive feature of the CM framework is that it provides a good platform for experimenting with and deploying new congestion control algorithms.

It is hard to characterize our scheme as rate-based or window-based; it is best characterized as a window-based

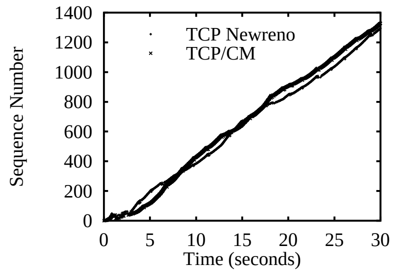

Figure 5: Sequence traces for TCP Newreno and TCP/CM, showing TCP/CM's true emulation of TCP Newreno congestion control.

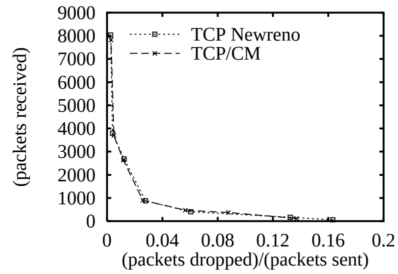

Figure 6: CM's rate control is TCP-friendly.

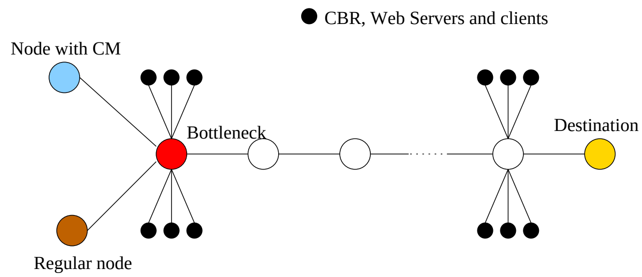

Figure 7: Topology for simulations.

scheme that modulates transmissions using a rate-based traffic shaper to reduce bursts. It is thus a hybrid window-rate scheme—while it uses a TCP-like window-based mechanism, it also shapes outgoing traffic using a rate estimate that is the ratio of the window size to the smoothed round-trip time. Furthermore, it changes to an exponentially decaying rate-based scheme when feedback is absent, as explained in Section 4.4.

Our primary consideration in the design of the congestion control module is that it be stable and friendly to existing TCP traffic in the network. The CM maintains a congestion window that 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. This does lead to a bias against long round-trip flows in a congested network [12, 35, 9], but we felt that an accurate emulation of TCP's increase algorithm is currently the safest deployment alternative. Upon a loss, the congestion window is halved, and when persistent congestion occurs (e.g., a TCP timeout), the rate drops to a small value forcing slow start [15] to occur.

We chose to implement a hybrid scheme instead of a pure TCP-like window-based scheme for two main reasons. First, this avoids bursts of transmissions that window-based schemes (e.g., TCP) are prone to, which makes it 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 using rate-based aging leads to more consistent performance without compromising network stability in these situations.

We conducted several experiments to validate the soundness of the CM's algorithm and tune it to perform well. Results from one set of experiments, for two connections—TCP Newreno [13] and TCP/CM—running over a network with random Web-like background traffic are shown in Figure 5. 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 TCP Newreno.

We now argue that our experimental data demonstrates that TCP/CM competes fairly with TCP Newreno. Figure 5 shows the sequence number plots for TCP/CM and Newreno for a particular transfer and topology (the topology itself is shown in Figure 7). We observed similar behavior over a wide range of bottleneck bandwidths and topologies. Figure 6 shows the throughput (number of successfully received packets) as a function of the loss rate (ratio of the number of dropped to sent packets) for TCP/CM and Newreno transfers lasting 20 seconds each. Each point on the graph is the average of ten runs for each bottleneck bandwidth; there were seven different bottlenecks: 100 Kbps, 300 Kbps, 500 Kbps, 1 Mbps, 3 Mbps, 5 Mbps, 10 Mbps; one-way delays of 60 ms (35 ms in some experiments), and queue sizes set to the bandwidth-delay product. These results show that TCP/CM and Newreno have similar throughput-loss relationships, which is evidence that the congestion control is "TCP-friendly."

4.2 Receiver Feedback

One of the fundamental requirements for stable end-to-end congestion control is receiver feedback. Without it, the sender would not 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 a timely manner. The sender's CM uses standard congestion indicators — packet losses and Explicit Congestion Notification (ECN) [10, 23] 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 infrequent.

4.2.1 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

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 8: Sender and receiver side pseudocode for handling probes/responses.

applications, we first need to determine an appropriate feedback frequency.

Unfortunately, it is not easy to determine the appropriate frequency in general. After some simple experiments that measured loss rate as a function of feedback frequency, we (somewhat arbitrarily) decided on a frequency of every one-half RTT. In Section 4.3 we discuss the insertion of a CM packet header that will allow the CM receiver to detect losses and thereby reduce the sensitivity to probe frequency.

4.2.2 Feedback mechanism

The CM uses two forms of feedback to adjust its congestion window and react to congestion: *application notification* and *explicit feedback*. Application notification occurs when the receiver application or transport protocol provides feedback to the sender application. 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 sequence of the last probe it received (i.e., the current one), the sequence of the last probe it responded to, and the number of packets received for each

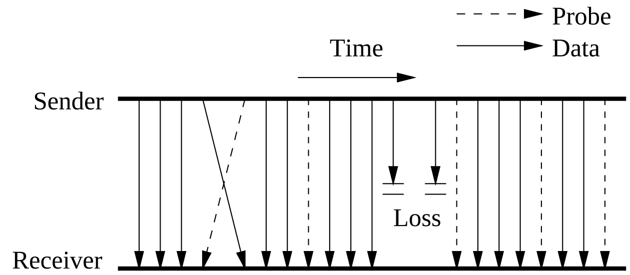

Figure 9: Example of reordering of probe and data packets.

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 8 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.

This pseudocode correctly identifies losses (and infers congestion) when the network does not reorder packets. Unfortunately, when the network does reorder data and probe packets, the packets received between a pair of probes may not be the same as the packets sent between those same probes. If the reordering occurs such that the fewer packets were received between the probes than were sent, the CM will erroneously identify a loss and perform congestion avoidance.

However, unlike TCP which would perform a premature retransmission, the ambiguity in this case is not as serious since the CM only performs congestion control, not retransmissions. Thus, if reordering is mistaken for a loss and we later recognize this mistake, we can undo the changes to the congestion state by updating the CM's congestion window. The problem then is determining how to undo a false window reduction. This can be done by observing the number of received and sent packets, `nrecd` and `nsent`, in successive probes. In particular, if packet reordering had occurred, the sender CM will see probe responses in which `nrecd` $>$ `nsent`, and can use these "extra" packets to identify the previous false window reduction. The reduction can then be reversed by incrementing the congestion window as necessary.

As it turns out, doing this correctly is a little more involved and requires a small amount of additional state. The sender CM begins storing the *cumulative* number of sent and received packets after it receives a response in which `nsent` $\neq$ `nrecd`. For each subsequent response, it compares the cumulative sent and received packets since it started storing responses. If the sent and received count are not equal and multiplicative decrease has not been performed for at least one round trip time, the CM invokes its decrease routine. As soon as the CM receives two successive reports with `nsent` = `nrecd`, the CM clears its memory of stored responses. At this stage, if the cumulative sent and received counts match, then we know that the CM performed an unnecessary window reduction and this action is reversed. The example shown in Figure 9 illustrates how this algorithm identifies reordered packets. In the first period, four packets were sent and only three received. Since a loss was indicated, the CM would perform a multiplicative decrease and store the information about the number of packets sent and received. The next probe indicates that an "extra" packet was received. If no further losses had occurred, the CM would notice that `nsent` = `nrecd` over the entire loss period and reverse the previous window reduction. However, the next probe indicates that only three out of five packets were received. If this probe was more than one round trip after the previous decrease, the CM would perform an additional window decrease. Assuming that no further losses occurred, the CM would not reverse either of the two decreases it had performed. In this scenario, the "extra" packet may have belonged to the first or third loss periods and the CM should have only performed a single window reduction. Since the CM cannot identify this from the information it has, it takes the conservative approach of two window reductions.

However, this solution is complex and does not work well when packet duplication occurs. Duplicate packets may cause losses to be hidden from the CM and wrongly reverse a correct window reduction. We address these issues by incorporating a CM packet header, which solves these problems (Section 4.3).

4.3 CM Packet Header

During our design, a question that repeatedly arose was whether the CM should incorporate a packet header of its own. There are some trade-offs involved in this decision.

- **Loss/congestion detection.** In the absence of a CM header with its own sequence number, detecting loss and congestion is problematic. It increases the reliance on a well-tuned probing scheme, because the CM receiver cannot provide feedback to sender as soon as congesion has occurred. To prevent the sender from transmitting in open-loop until the next probe and response when congestion has already occurred is undesirable. Note that this is not a significant issue with applications like TCP that incorporate their own sequence spaces and congestion detection machinery, using `cm_update()` to inform the CM about congestion.

- **Reordering issues.** A CM header with an incrementing sequence number attached to data packets eases the task of distinguishing losses from reordering.

- **Deployment concerns.** A CM header is certainly cleaner, in the sense that all the information used by the sender and receiver CM modules can be encapsulated in it. However, adding this entails more change, especially at the receivers. This is why we hesitated including it initially.

In our initial design, we decided to incorporate the complicated reordering machinery (Section 4.2.2) and thought that we could arrive at a straightforward solution to the probing frequency problem. This led us to believe that we could get away with the simpler alternative of eliminating a CM header. However, subsequent experience and reflection convinced us that the complexity and inefficiency of the reordering distinguisher was enough to justify the additional change required at receivers to process the CM header. Furthermore, while we still believe that the feedback frequency problem is tractable, we do not think it is trivial.

We convinced ourselves that the addition of a CM header is a significant deployment problem because the CM already requires receiver changes to respond to periodic sender probes and to implement the receiver-side API (both of which require changes to the protocol stack). However, we would like a CM sender to communicate with a receiver that *does not* have a CM, and work well for applications that provide feedback to the sender (which can in turn use `cm_update()` to inform the CM of the state of the network). We achieve this by creating a new CM protocol type identifier (`IPPROTO_CM`) and negotiating the use of the CM header via a two-way handshake between sender and receiver.

The CM uses the packet header format shown in Figure 10 for its messages. This is used in the probe and response packets, in data packets, and in state setup/reset packets.

The `Protocol` field is used by the receiver CM to decide which transport protocol to pass the incoming packet on to. This is needed because the sender CM rewrites the IP protocol field of all outgoing packets to CM-enabled receivers using a new IP protocol type `IPPROTO_CM` (this protocol number needs to be standardized by the IETF).

0	4	8	16	31
Version	Type	Protocol	Checksum	
Sequence				
FlowID				

Figure 10: Format of the CM header. `Type` can be one of `SYN` (1), `SYN-ACK` (2), `RST` (3), `PROBE` (4), or `RESPONSE` (5). `Protocol` is the transport protocol type that the packet should be dispatched to at the receiver. `Sequence` is an incrementing packet sequence number and `FlowID` uniquely identifies the flow.

0	4	8	16	31
Version	Type	Unused	Checksum	
NumFlows			Unused	
ThisProbe				
LastProbe				
FlowID				
Count				
...				
FlowID				
Count				

Figure 11: Format of the CM response header. `NumFlows` is the number of flows for which statistics are included in the response. `ThisProbe` is the sequence number of the probe triggering the response, and `LastProbe` is the sequence number of the previous probe received. `Count` is the number of bytes received between `ThisProbe` and `LastProbe` for the flow with identifier `FlowID`.

When any data is received, the sequence number field increments for every packet that is transmitted to the destination, independent of FlowID. The receiver CM monitors these sequence numbers (and ECN as well) to determine if congestion has occurred. It is robust to reordering in the same way that TCP is, flagging a congestion event to the sender only if a packet at least three packets greater than a missing one arrives.

For type PROBE packets, the sequence number refers to the probe sequence number, which is a different incrementing stream from the data sequence numbers. In response, the receiver sends a RESPONSE packet, which has a very different format from the other types (see Figure 11). The RESPONSE packet carries in it per-flow information of the number of received bytes between two probe sequence epochs—the current sender probe and the previous one received by the responder.

The SYN packet type is used to perform a two-way handshake between sender and receiver. The CM sender uses this to determine if a given receiver is CM-enabled. Observe that a three-way handshake is unnecessary because the receiver's RESPONSE messages do not use an independent sequence number, they only echo the sender's query. If a pair of hosts are both sending and receiving CM-enabled traffic between each other, there are two "connections" of the probe/response protocol in action.

The RST type is used to reset the sender's state after crash recovery or any other loss of synchronization in the sender and receiver states.

When the sender encounters a new receiver, it sends a SYN packet with an initial sequence number using the same mechanism that TCP uses. There are two cases to consider: a CM-enabled receiver and a non-CM receiver.

In the first case, the receiver's IP layer passes the packet on to the receiver CM because of the IPPROTO_CM protocol type. The receiver CM generates a SYN-ACK in a response, echoing the sequence number. If the receiver was not CM-enabled, the SYN would be dropped and an ICMP "protocol not available" message sent to the sender. Upon the receipt of this message or on a timeout (since the sender cannot rely on the ICMP being generated or received), the sender realizes that the receiver is not CM-enabled and proceeds without the CM header.

In the above description, no application data is sent during the round-trip to effect this handshake. This is undesirable, so we permit packets to be sent emulating TCP slow start, and assuming that the receiver does not have a CM (i.e., these packets do not have a CM header). If the receiver is indeed CM-enabled, we discover this when the SYN-ACK arrives and start incorporating a CM header on packets.

The existence of the CM header is transparent to the transport protocols and applications at both the sender and receiver. Thus, when the receiver-side CM receives a packet with data, it adds the payload size (from the IP header) to the number of bytes received on the corresponding FlowID, strips the CM header and passes the packet to the higher layer based on the Protocol field in the CM header.

Finally, PROBE and RESPONSE types are handled as explained in Section 4.2.2.

4.4 Exponential aging

The probing protocol described above periodically elicits a response from the receiver regarding the number of received bytes to infer the state of the network. However, probe messages or responses may be lost during times of congestion, because of which the sender will not have an accurate estimate of the network state.

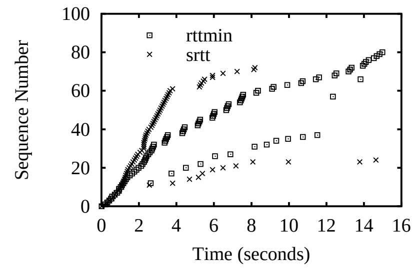

Figure 12: Sequence traces showing that exponential aging based on mean round-trip time causes substantially more losses than the alternative based on minimum round-trip time.

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 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 behavior similar to 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 12 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 as a result of larger 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 con-

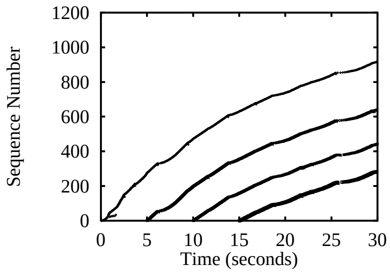

Figure 13: The CM scheduler apportions bandwidth well between simultaneous flows.

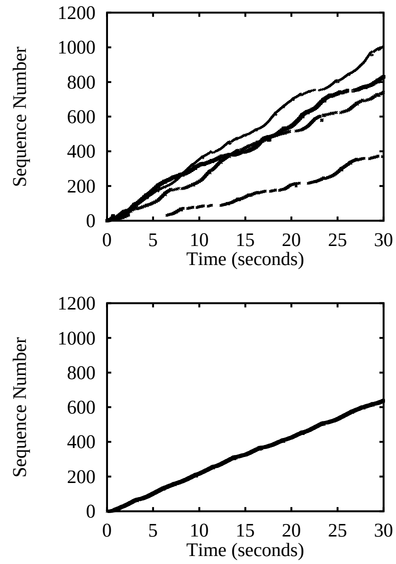

Figure 14: The top graph shows sequence traces for a Web-like workload using 4 concurrent TCP Newreno connections. The performance of these transfers is highly variable and inconsistent. The bottom graph demonstrates the consistent and predictable performance of a Web workload using TCP/CM—the four connections are indistinguishable!.

servative 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 12 which shows the sequence traces of transfers in each mode. Using the smoothed round-trip time, the connection experiences a larger number of losses and does not recover from it, while with the minimum, it does not burst out as many packets.

4.5 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, priorities 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.

This problem may be tractable using *flow segregation*, where the flows are aggregated not by host address but by some combination of address, port numbers, and identifiers. If an application knows *a priori* 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 `cm_update()` hints and the probing protocol, 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.

4.6 Flow Scheduling

One of the advantages of the modular CM design is that one scheduler can be swapped with another without affecting the rest of the system. We currently use a simple Hierarchical Round Robin (HRR) Scheduler [17]. The scheduler apportions bandwidth among flows in proportion to pre-configured weights (and soon based on receiver hints). Figure 13 shows flows starting at different times eventually achieving the same rate allocation from the HRR scheduler.

The scheduler is invoked whenever any application makes a call to transmit data. It schedules the request for a time in the future based on the CM congestion window and on past transmissions, without considering the current request size (indeed, there is no explicit request size in the API). At this time, it calls `cmapp_send()`, causing the application to send up to `mtu` bytes (obtained by the application using `cm_mtu()`.

The scheduler as currently implemented performs only bandwidth allocation, and does not use delay bounds in its scheduling. While this is adequate for applications like TCP, it does not accomodate delay-sensitive applications. We are planning on implementing an H-FSC-like scheduler [31] in the CM.

5 Application Performance

We have implemented the CM in ns [33]. We have also implemented a TCP agent and an audio server application to use the CM, and performed experiments with a variety of topologies. We present and discuss the results for the topology shown in Figure 7.

5.1 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 show that the CM ensures proper behavior in the face of congestion and improves the consistency of application performance.

Figure 14 shows two sets of sequence traces: when TCP Newreno was used, and when TCP/CM was used for the four connections. Using TCP Newreno, the performance of the four connections varies between 99 Kbps and 268 Kbps, a factor of 2.7 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 170 Kbps, without causing as 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 calculated the fairness index [16] for several experiments; for the TCP/CM ensemble, the index was 1.0 while for the Newreno ensemble it was 0.952. Note that the aggregate throughput obtained by the TCP/CM connections ($\approx$ 680 Kbps) is lower than the aggregate throughput obtained by independent TCP Newreno connections ($\approx$ 785 Kbps). This is not 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 correct 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.

5.2 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 propagation delay of 120 ms. 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 chose amongst encodings of 10, 20, 40, 80, 160 and 320 Kbps. It always performed transmissions of 1 KB packets.

The results of the experiment, shown in Figure 15, confirm that the CM, TCP/CM and adaptive audio perform as desired. The TCP Newreno transfer obtained approximately 150 Kbps. The combination of the audio, at about 65Kbps, and the TCP/CM, at about 85 Kbps, was close to the throughput of Newreno. The audio primarily used the 80 Kbps encoding, occasionally switching to the 40 and 160 Kbps encodings. These results show that the CM API enables applications like layered audio to adapt well to network conditions, despite only using coarse-grained layers.

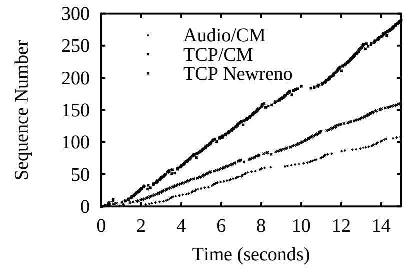

Figure 15: Performance of a layered adaptive audio application using the CM.

6 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, and worse, will lead to instability in a network where most clients operate in this fashion. To solve these problems, researchers have proposed two classes of solutions—*application-level solutions* and *integrated TCP congestion control.*

6.1 Application-level Solutions

Application-level solutions 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) [28] 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

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

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.

6.2 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, 21, 32]. In RFC 2140 [32], 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. In [1, 21], 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. Although these approaches do solve some of the problems associated with the Web scenario, they are transport-specific and do not provide any APIs for application adaptation.

6.3 Real-time Multimedia

There has been some recent work in developing congestion control protocols for such applications. Much of this work has been in the context of multicast video (e.g., IVS [3], RLM [18], etc.). There have also been numerous recent congestion control proposals for various reliable multicast applications (for a survey, see [25]). 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. For example, the RAP protocol [24] is a rate-based congestion control scheme intended for streaming applications. The CM provides a general architecture within which a scheme like RAP could be implemented as the congestion controller. Because the CM is independent of specific transport protocols and facilitates the sharing of information, it integrates congestion management across all flows.

7 Conclusions

This paper motivated and presented the Congestion Manager (CM) architecture for managing Internet congestion. At the sender, the congestion manager maintains network statistics on a per-receiver basis, performs congestion avoidance and control, schedules transmissions, and exposes an API based on ALF principles to enable applications to adapt to congestion. At the receiver, the CM maintains loss statistics, responds to sender probes, and exposes an API for applications to provide hints on apportioning bandwidth.

We showed using ns-based simulations that the CM enabled efficient multiplexing of logically different streams, ensuring proper congestion behavior for an ensemble of flows. Our results showed that the CM enhances the predictability of TCP performance and allows a layered audio application to adapt well to changing bandwidth.

In terms of deployment, the full benefits of the CM architecture require changes to both senders and receivers. However, substantial benefits can be obtained with sender changes alone (e.g., at popular servers), especially for those applications such as TCP that already have their own feedback mechanism. This incremental deployment path encourages us to be optimistic about the CM's long-term prospects.

Acknowledgments

Several people provided useful comments and constructive criticism of the CM design and earlier versions of this paper. Our special thanks to Sally Floyd, Mark Handley, Vern Paxson, and Steve McCanne for extensive discussions that greatly improved (and simplified) the CM API. We thank Mark Allman, Jean Bolot, Sally Floyd, Mark Handley, Tom Henderson, Frans Kaashoek, Shiv Kalyanaraman, Steven McCanne, Greg Minshall, Robert Morris, Vern Paxson, Bodhi Priyantha, Suchitra Raman, Lixia Zhang, and the SIGCOMM reviewers for their comments on this work. This research was supported in part by a research grant from the NTT Corporation. Much of this work was done while Srinivasan Seshan was a Visiting Scientist at the M.I.T. Laboratory for Computer Science.

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. Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic. *Computer Communications Review 21*, 5 (Oct. 1991).

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

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

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

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

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

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

[16] Jain, R. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[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] McCanne, S., Jacobson, V., and Vetterli, M. Receiver-driven Layered Multicast. In *Proc ACM SIGCOMM* (Aug. 1996).

[19] Mogul, J., and Deering, S. *Path MTU Discovery*, Nov 1990. RFC-1191.

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

[21] Padmanabhan, V. *Addressing the Challenges of Web Data Transport*. PhD thesis, Univ. of California, Berkeley, September 1998.

[22] 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.

[23] 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.

[24] Rejaie, R., Handley, M., and Estrin, D. RAP: An End-to-end Rate-based Congestion Control Mechanism for Real-time Streams in the Internet. To appear in Proc. Infocom 99.

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

[26] Saltzer, J., Reed, D., and Clark, D. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems 2* (Nov 1984), 277–288.

[27] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications*. RFC, Jan 1996. RFC-1889.

[28] Spero, S. Session Control Protocol (SCP). http:// www.w3.org/ pub/ WWW/ Protocols/ HTTP-NG/ http-ng-scp.html, 1996.

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

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

[31] 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).

[32] Touch, J. *TCP Control Block Interdependence*. RFC, April 1997. RFC-2140.

[33] VINT Project. http://netweb.usc.edu/vint, 1998.

[34] WebTP Home Page. http://webtp.eecs.berkeley.edu/, 1999.

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

[36] Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D. RSVP: A new resource ReSerVation Protocol. *IEEE Network Magazine* (Sept. 1993), 8–18.