System Support for Bandwidth Management and Content Adaptation in Internet Applications

David Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan*, Hari Balakrishnan
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139
{dga, bansal, dcurtis, srini, hari}@lcs.mit.edu

Abstract

This paper describes the implementation and evaluation of an operating system module, the Congestion Manager (CM), which provides integrated network flow management and exports a convenient programming interface that allows applications to be notified of, and adapt to, changing network conditions. We describe the API by which applications interface with the CM, and the architectural considerations that factored into the design. To evaluate the architecture and API, we describe our implementations of TCP; a streaming layered audio/video application; and an interactive audio application using the CM, and show that they achieve adaptive behavior without incurring much end-system overhead. All flows including TCP benefit from the sharing of congestion information, and applications are able to incorporate new functionality such as congestion control and adaptive behavior.

1 Introduction

The impressive scalability of the Internet infrastructure is in large part due to a design philosophy that advocates a simple architecture for the core of the network, with most of the intelligence and state management implemented in the end systems [10]. The service model provided by the network substrate is therefore primarily a "best-effort" one, which implies that packets may be lost, reordered or duplicated, and end-to-end delays may be variable. Congestion and accompanying packet loss are common in heterogeneous networks like the Internet because of overload, when demand for router resources, such as bandwidth and buffer space, exceeds what is available. Thus, end systems in the Internet should incorporate mechanisms for detecting and reacting to network congestion, probing for spare capacity when the network is uncongested, as well as managing their available bandwidth effectively.

Previous work has demonstrated that the result of uncontrolled congestion is a phenomenon commonly called "congestion collapse" [8, 13]. Congestion collapse is largely alleviated today because the popular end-to-end Transmission Control Protocol (TCP) [30, 40] incorporates sound congestion avoidance and control algorithms. However, while TCP does implement congestion control [18], many applications including the Web [6, 12] use several logically different streams in parallel, resulting in multiple concurrent TCP connections between the same pair of hosts. As several researchers have shown [2, 3, 27, 28, 42], these concurrent connections compete with – rather than learn from – each other about network conditions to the same receiver, and end up being unfair to other applications that use fewer connections. The ability to share congestion information between concurrent flows is therefore a useful feature, one that promotes cooperation among different flows rather than adverse competition.

In today's Internet is the increasing number of applications that do not use TCP as their underlying transport, because of the constraining reliability and ordering semantics imposed by its in-order byte-stream abstraction. Streaming audio and video [25, 34, 41] and customized image transport protocols are significant examples. Such applications use custom protocols that run over the User Datagram Protocol (UDP) [29], often without implementing any form of congestion control. The unchecked proliferation of such applications would have a significant adverse effect on the stability of the network [3, 8, 13].

Many Internet applications deliver documents and images or stream audio and video to end users and are *interactive* in nature. A simple but useful figure-of-merit for interactive content delivery is the end-to-end download latency; users typically wait no more than a few seconds before aborting a transfer if they do not observe progress. Therefore, it would be beneficial for content providers to adapt *what* they disseminate to the state of the network, so as not to exceed a threshold latency. Fortunately, such content adaptation is possible for most applications. Streaming audio and video applications typically encode information in a range of formats corresponding to different encoding (transmission) rates and degrees of loss resiliency. Image encoding formats accommodate a range of qualities to suit a

* Carnegie Mellon University, Pittsburgh, PA; srini@seshan.org

variety of client requirements.

Today, the implementor of an Internet content dissemination application has a challenging task: for her application to be safe for widespread Internet deployment, she must either use TCP and suffer the consequences of its fully-reliable, byte-stream abstraction, or use an application-specific protocol over UDP. With the latter option, she must re-implement congestion control mechanisms, thereby risking errors not just in the implementation of her protocol, but also in the implementation of the congestion controller. Furthermore, neither alternative allows for sharing congestion information across flows. Finally, the common application programming interface (API) classes for network applications—Berkeley sockets, streams, and Winsock [31]—do not expose any information about the state of the network to applications in a standard way[1]. This makes it difficult for applications running on existing end host operating systems to make an informed decision, taking network variables into account, during content adaptation.

1.1 The Congestion Manager

Our previous work provided the rationale, initial design, and simulation of the Congestion Manager, an end-system architecture for sharing congestion information between multiple concurrent flows [3]. In this paper, we describe the implementation and evaluation of the CM in the Linux operating system. We focus on a version of the CM where the only changes made to the current IP stack are at the data sender, with feedback about congestion or successful data receptions being provided by the receiver CM applications to their sending peers, which communicate this information to the CM via an API. We present a summary of the API used by applications to adapt their transmissions to changing network conditions, and focus on those elements of the API that changed in the transition from the simulation to the implementation.

We evaluate the Congestion Manager by posing and answering several key questions:

Is its callback interface, used to inform applications of network state and other events, effective for a diverse set of applications to adapt without placing a significant burden on developers?

Because most robust congestion control algorithms rely on receiver feedback, it is natural to expect that a CM receiver is needed to inform the CM sender of successful transmissions and packet losses. However, to facilitate deployment, we have designed our system to take advantage of the fact that several protocols including TCP and other applications already incorporate some form of application-specific feedback, providing the CM with the loss and timing information it needs to function effectively.

Using the CM API, we implement several case studies both in and out of the kernel, showing the applicability of the API to many different application architectures. Our implementation of a layered streaming audio/video application demonstrates that the CM architecture can be used to implement highly adaptive congestion controlled applications. Adaptation via the CM helps these applications achieve better performance and also be fair to other flows on the Internet.

We have also modified a legacy application—the Internet audio tool *vat* from the MASH toolkit [23]—to use the CM to perform adaptive real-time delivery. Since less than one hundred lines of source code modification was required to CM-enable this complex application and make it adapt to network conditions, we believe it demonstrates the ease with which the CM makes applications adaptive.

Is the congestion control correct?

As a trusted kernel module, the CM frees both transport protocols and applications from the burden of implementing congestion management. We show that the CM behaves in the same network-friendly manner as TCP for single flows. Furthermore, by integrating flow information between both kernel protocols and user applications, we ensure that an *ensemble* of concurrent flows is not an overly aggressive user of the network.

In today's off-the-shelf operating systems, does the CM place any performance limitations upon applications?

We find that our implementation of TCP (which uses the CM for its congestion control) has essentially the same performance as standard TCP, with the added benefits of integrated congestion management across flows, with only small (0-3%) CPU overhead.

In a CM system where no changes are made to the receiver protocol stack, UDP-based applications must implement a congestion feedback mechanism, resulting in more overhead compared to the TCP applications. However, we show that these applications remain viable, and that the architectural change and API calls reduce worst-case throughput by 0 - 25%, even for applications that desire fine-grained information about the network on a per-packet basis.

To our knowledge, this is the first implementation of a general application-independent system that combines integrated flow management with a convenient API to enable content adaptation. The end-result is that applications achieve the desirable congestion control properties of long-running TCP connections, together with the flexibility to adapt data transmissions to prevailing network conditions.

The rest of this paper is organized as follows. Section 2 describes our system architecture and implementation. Section 3 describes how network-adaptive applications can be engineered using the CM, while Section 4 presents the results of several experiments. In Section 5,

[1] Utilities like `netstat` and `ifconfig` provide some information about devices, but not end-to-end performance information that can be used for adapting content.

we discuss some miscellaneous details and open issues in the CM architecture. We survey related work in Section 6 and conclude with a summary in Section 7.

2 System Architecture and Implementation

The CM performs two important functions. First, it enables efficient multiplexing and congestion control by integrating congestion management across multiple flows. Second, it enables efficient application adaptation to congestion by exposing its knowledge of network conditions to applications. Most of the CM functionality in our Linux implementation is in-kernel; this choice makes it convenient to integrate congestion management across both TCP flows and other user-level protocols, since TCP is implemented in the kernel.

To perform efficient aggregation of congestion information across concurrent flows, the CM has to identify which flows potentially share a common bottleneck link *en route* to various receivers. In general, this is a difficult problem, since it requires an understanding of the paths taken by different flows. However, in today's Internet, all flows destined to the same end host take the same path in the common case, and we use this group of flows as the default granularity of flow aggregation[2]. We call this group a *macroflow*: a group of flows that share the same congestion state, control algorithms, and state information in the CM. Each flow has a sending application that is responsible for its transmissions; we call this a *CM client*. CM clients are in-kernel protocols like TCP or user-space applications.

The CM incorporates a *congestion controller* that performs congestion avoidance and control on a per-macroflow basis. It uses a window-based algorithm that mimics TCP's additive-increase/multiplicative decrease (AIMD) scheme to ensure fairness to other TCP flows on the Internet. However, the modularity provided by the CM encourages experimentation with other non-AIMD schemes that may be better suited to specific data types such as audio or video.

While the congestion controller determines what the current window (rate) ought to be for each macroflow, a *scheduler* decides how this is apportioned among the constituent flows. Currently, our implementation uses a standard unweighted round-robin scheduler.

In-kernel CM clients such as a TCP sender use CM function calls to transmit data and learn about network conditions and events. In contrast, user-space clients interact with the CM using a portable, platform-independent API described in Section 2.1. A platform-dependent CM library, `libcm`, is responsible for interfacing between the kernel and these clients, and is described in Section 2.2. These components are shown in

[2] This is not strictly true in the presence of network-layer differentiated services. We address this issue later in this section and in Section 5.

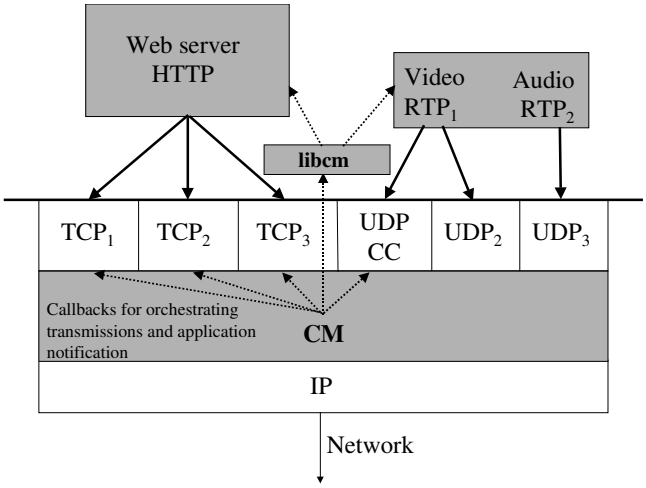

Figure 1. Architecture of the congestion manager at the data sender, showing the CM library and the CM. The dotted arrows show callbacks, and solid lines show the datapath. UDP-CC is a congestion-controlled UDP socket implemented using the CM.

Figure 1.

When a client opens a CM-enabled socket, the CM allocates a flow to it and assigns the flow to the appropriate macroflow based on its destination. The client initiates data transmission by requesting permission to send data. At some point in the future depending on the available rate, the CM issues a callback permitting the client to send data. The client then transmits data, and tells the CM it has done so. When the client receives feedback from the receiver about its past transmissions, it notifies the CM about these and continues.

When a client makes a request to send on a flow, the scheduler checks whether the corresponding macroflow's window is open. If so, the request is granted and the client notified, upon which it may send some data. Whenever any data is transmitted, the sender's IP layer notifies the CM, allowing it to "charge" the transmission to the appropriate macroflow. When the client receives feedback from its remote counterpart, it informs the CM of the loss rate, number of bytes transmitted correctly, and the observed round trip time. On a successful transmission, the CM opens up the window according to its congestion management algorithm and grants the next, if any, pending request on a flow associated with this macroflow. The scheduler also has a timer-driven component to perform background tasks and error handling.

2.1 CM API

The CM API is specified as a set of functions and callbacks that a client uses to interface with the CM. It specifies functions for managing state, for performing data transmissions, for applications to inform the CM of losses, for querying the CM about network state, and for constructing and splitting macroflows if the default per-destination aggregation is unsuitable for an application. The CM API is discussed in detail in [3], which presents the design rationale for the Congestion Manager. Here we provide an overview of the API and a discussion of those features which changed during the transition from simulation to implementation.

2.1.1 State management

All CM applications call `cm_open()` before using the CM, passing the source and destination addresses and transport-layer port numbers, in the form of a `struct sockaddr`. The original CM API required only a destination address, but the source address specification was necesary to handle multihomed hosts. `cm_open` returns a flow identifier (`cm_flowid`), which is used as a handle for all future CM calls. Applications may call `cm_mtu(cm_flowid)` to obtain the maximum transmission unit to a destination. When a flow terminates, the application should call `cm_close(cm_flowid)`.

2.1.2 Data transmission

There are three ways in which an application can use the CM to transmit data. These allow a variety of adaptation strategies, depending on the nature of the client application and its software structure.

(i) **Buffered send.** This API uses a conventional `write()` or `sendto()` call, but the resulting data transmission is paced by the Congestion Manager. We use this to implement a generic congestion-controlled UDP socket (without content adaptation), useful for bulk transmissions that do not require TCP-style reliability or fine-grained control over what data gets sent at a given point in time.

(ii) **Request/callback.** This is the preferred mode of communication for adaptive senders that are based on the ALF (Application-Level Framing [11]) principle. Here, the client does not send data via the CM; rather, it calls `cm_request(cm_flowid)` and expects a notification via the `cmapp_send(cm_flowid)` callback when this request is granted by the CM, at which time the client transmits its data. This approach puts the sender in firm control of deciding what to transmit at a given time, and allows the sender to adapt to sudden changes in network performance, which is hard to do in a conventional buffered transmission API. The client callback is a grant for the flow to send up to MTU bytes of data. Each call to `cm_request()` is an implicit request for sending up to MTU bytes, which simplifies the internal implementation of the CM. This API is ideally suited for an implementation of TCP, since it needs to make a decision at each stage about whether to retransmit a segment or send a new one. In the implementation, the `cmapp_send` callback now provides the client with the ID of the flow that may transmit. To allow for client programming flexibility, the client may now specify its callback function via `cm_register_send()`.

(iii) **Rate callback.** A self-timed application transmitting on a fixed schedule may receive callbacks from the CM notifying it when the parameters of its communication channel have changed, so that it can change the frequency of its timer loop or its packet size. The CM informs the client of the rate, round-trip time, and packet loss rate for a flow via the `cmapp_update()` callback. During implementation, we added a registration function, `cm_register_update()` to select the rate callback function, and the `cm_thresh(down,up)` function: If the rate reduces by a factor of `down` or increases by a factor of `up`, the CM calls `cmapp_update()`. This transmission API is ideally suited for streaming layered audio and video applications.

2.1.3 Application notifications

One of the goals of our work was to investigate a CM implementation that requires no changes at the receiver. Performing congestion management requires feedback about transmissions: TCP provides this feedback automatically; some UDP applications may need to be modified to do so, but without any system-wide changes. Senders must then inform the CM about the number of sent and received packets, type of congestion loss if any, and a round-trip time sample using the `cm_update( cm_flowid, nsent, nrecd, lossmode, rtt)` function. The CM distinguishes between "persistent" congestion as would occur on a TCP timeout, versus "transient" congestion when only one packet in a window is lost. It also allows congestion to be notified using Explicit Congestion Notification (ECN) [32], which uses packet markings rather than drops to infer congestion.

To perform accurate bookkeeping of the congestion window and outstanding bytes for a macroflow, the CM needs to know of each successful transmission from the host. Rather than encumber clients with reporting this information, we modify the IP output routine to call `cm_notify(cm_flowid, nsent)` on each transmission. (The IP layer obtains the `cm_flowid` using a well-defined CM interface that takes the flow parameters (addresses, ports, protocol field) as arguments.) However, if a client decides not to transmit any data upon a `cmapp_send()` callback invocation, it is expected to call

`cm_notify(dst, 0)` to allow the CM to permit some other flows on the macroflow to transmit data.

2.1.4 Querying

If a client wishes to learn about its (per-flow) available bandwidth and round-trip time, it can use the `cm_query()` call, which returns these quantities. This is especially useful at the beginning of a stream when clients can make an informed decision about the data encoding to transmit (e.g., a large color or smaller greyscale image).

2.2 `libcm`: The CM library

The CM library provides users with the convenience of a callback-based API while separating them from the details of how the kernel to user callbacks are implemented. While direct function callbacks are convenient and efficient in the same address space, as is the case when the kernel TCP is a client of the CM, callbacks from the kernel to user code in conventional operating systems are more difficult. A key decision in the implementation of `libcm` was choosing a kernel/user interface that maximizes portability, and minimizes both performance overhead and the difficulty of integration with existing applications. The resulting *internal* interface between `libcm` and the kernel is:

1. `select()` on a single per-application CM control socket. The write bit indicates that a flow may send data, and the exception bit indicates that network conditions have changed.

2. Perform an `ioctl` to extract a list of all flow IDs which may send, or to receive the current network conditions for a flow.

Note that client programs of the CM do not see this interface; they see only the standard `cm_*` functions provided by `libcm`. The use of sockets or signals does change the way the application's event handling loop interacts with `libcm`; after passing the socket into `libcm`, the library performs the appropriate `ioctl`s and then calls back into the application.

2.2.1 Implementation alternatives

We considered a number of mechanisms with which to implement `libcm`. In this section, we discuss our reasons for choosing the control-socket+select+ioctl approach.

While much research has focused on reducing the cost of crossing the user/kernel boundary (extensible kernels in SPIN [7], fast, generic IPC in Mach [5], etc.) many conventional operating systems remain limited to more primitive methods for kernel-to-user notification, each with their own advantages and disadvantages. While functionality like the Mach port set-based IPC would be ideal for our purposes, pragmatically we considered four common mechanisms for kernel to user communication: Signals, system calls, semaphores, and sockets. A discussion of the merits of each follows.

Signals have several immediate drawbacks. First, if the CM were to appropriate an existing signal for its own use, it might conflict with an application using the same signal. Avoiding this conflict would require the standardization of a new signal type, a process both slow and of questionable value, given the existence of better alternatives. Second, the cost to an application to receive a signal is relatively high, and some legacy applications may not be signal-safe. While the new POSIX 1003.1b [17] soft realtime signals allow delivering a 32-bit quantity with a signal, applications would need to follow up a signal with a system call to obtain all of the information the kernel wished to deliver, since multiple flows may become ready at once. For these reasons, we consider mandating the use of signals the wrong course for implementing the kernel to user callbacks. However, we provide an *option* for processes to receive a `SIGIO` when their control socket status changes, akin to POSIX asynchronous I/O.

System calls that block do not integrate well with applications that already have their own event loop, since without polling, applications cannot wait on the results of multiple system calls. A system call is able to return immediately with the data the user needs, but the impediments it poses to application integration are large. System calls would work well in a threaded environment, but this presupposes threading support, and the select-based mechanism we describe below can be used in a threaded system without major additional overhead.

Semaphores suffer from the immediate drawback that they are not commonly used in network applications. For an application that uses `semop` on an array of semaphores as its event loop, a CM semaphore might be the best implementation avenue, for many of the same reasons that we chose sockets for network-adaptive applications. However, most network applications use socket sets instead of semaphore sets, and sockets have a few other benefits, which we discuss next.

Sockets provide a well-defined and flexible interface for applications in the form of the `select()` system call, though they have a downside similar to that of signals: an application wishing to receive a notification via a socket in a non-blocking manner must `select()` on the socket, and then perform a system call to obtain data from the socket. However, a select-based interface meshes well with many network applications that already have a select-loop based architecture. Utilizing a control socket also helps restrict the code changes caused by the CM to the networking stack.

Finally, we decided to use a single control socket instead of one control socket per flow to avoid unnecessary overhead in applications with large numbers of open socket descriptors, such as `select()`-based web-

servers and caches. Because some aspects of select scale linearly with the number of descriptors, and many operating systems have limits on the number of open descriptors, we deemed doubling the socket load for high-performance network applications a bad idea.

2.2.2 Extracting data from the socket

Select provides notification that "some event" has occured. In theory, 7 different events could be sent by abusing the read, write, and exception bits, but applications need to extract more information than this. The CM provides two types of callbacks. Generally speaking, the first is a "permission to send" callback for a particular flow. To maintain even distribution of bandwidth between flows, a loose ordering should be preserved with these messages, but exact ordering is unimportant provided no flows are ignored until the application receives further updates (thereby starving the flows). If multiple permission notifications occur, the application should receive *all* of them so it can send data on all available flows. The second callback is a "status changed" notification. If multiple status changes occur before the application obtains this data from the kernel, then only the *current* status matters.

The weak ordering and lack of history prompted us to choose an `ioctl`-based query instead of a `read` or message queue interface, minimizing the state that must be maintained in the kernel. Status updates simply return the current CM-maintained network state estimate, and "who can send" queries perform a select-like operation on the flows maintained by the kernel, requiring no extra state, instead of a potentially expensive per-process message queue or data stream. Returning all available flows has an added benefit of reducing the number of system calls that must be made if several flows become ready simultaneously.

3 Engineering Network-adaptive Applications

In this section, we describe several different classes of applications, and describe the ways those applications can make use of the CM. We explore two in-kernel clients, and several user-space data server programs, and examine the task of integrating each with the CM.

3.1 Software Architecture Issues

Typical network applications fall into one of several categories:

- Data-driven: Applications that transmit prespecified data, such as a single file, then exit.
- Synchronous event-driven: Self-timed data delivery servers, like streaming audio servers.
- Asynchronous event-driven: File servers (http, ftp) and other network-clocked applications.

The CM library provides several options for adaptive applications that wish to make use of its services:

1. Data-driven applications may use the buffered API to efficiently pace their data transmissions.
2. An application may operate in an entirely callback-based manner by allowing `libcm` to provide its own event loop, calling into the application when flows are ready. This is most useful for applications coded with the CM in mind.
3. Signal-driven applications may request a SIGIO notification from the CM when an event occurs.
4. Applications with select-based event loops can simply add the CM control socket into their select set, and call the `libcm` dispatcher when the socket is ready. Rate-clocked applications (or polling-based applications) can perform a similar non-blocking select test on the descriptor when they awaken to send data, or, if they `sleep`, can replace the sleep with a timed blocking `select` call.
5. Applications may poll the CM on their own schedule.

The remainder of this section describes how particular clients use different CM APIs, from the low-bandwidth `vat` audio application, to the performance-critical kernel TCP implementation. Note that all UDP-based clients must implement application level data acknowledgements in order to make use of the CM.

3.2 TCP

We implemented TCP as an in-kernel CM client. TCP/CM offloads all congestion control to the CM, while retaining all other TCP functionality (connection establishment and termination, loss recovery and protocol state handling). TCP uses the request/callback API as low-overhead direct function calls in the same protection domain. This gives TCP the tight control it needs over packet scheduling. For example, while the arrival of a new acknowledgement typically causes TCP to transmit new data, the arrival of three duplicate ACKs causes TCP to retransmit an old packet.

Connection creation. When TCP creates a new connection via either `accept` (inbound) or `connect` (outbound), it calls `cm_open()` to associate the TCP connection with a CM flow. Thereafter, the pacing of outgoing data on this connection is controlled by the CM. When application data becomes available, after performing all the non-congestion-related checks (e.g., the Nagle algorithm [40], etc.) data is queued and `cm_request()` is called for the flow. When the CM scheduler schedules the flow for transmission,

the `cmapp_send()` routine for TCP is called. The `cmapp_send()` for TCP transmits any retransmission from the retransmission queue. Otherwise, it transmits the data present in the transmit socket buffer by sending up to one maximum segment size of data per call. Finally, the IP output routine calls `cm_notify()` when the data is actually sent out.

TCP input. The TCP input routines now feedback to the CM. Round trip time (RTT) sample collection is done as usual using either RFC 1323 timestamps [19] or Karn's algorithm [21] and is passed to CM via `cm_update()`. The smoothed estimates of the RTT (`srtt`) and round-trip time deviation are calculated by the CM, which can now obtain a better average by combining samples from different connections to the same receiver. This is available to each TCP connection via `cm_query()`, and is useful in loss recovery.

Data acknowledgements. On arrival of an ACK for new data, the TCP sender calls `cm_update()` to inform the CM of a successful transmission. Duplicate acknowledgements cause TCP to check its dupack count (`dup_acks`). If `dup_acks` < 3, then TCP does nothing. If `dup_acks` $== 3$, then TCP assumes a simple, congestion-caused packet loss, and calls `cm_update` to inform the CM. TCP also enqueues a retransmission of the lost segment and calls `cm_request()`. If `dup_acks` > 3, TCP assumes that a segment reached the receiver and caused this ACK to be sent. It therefore calls `cm_update()`. Unlike duplicate ACKs, the expiration of the TCP retransmission timer notifies the sender of a more serious batch of losses, so it calls `cm_update` with the CM_LOST_FEEDBACK option set to signify the occurrence of persistent congestion to the CM. TCP also enqueues a retransmission of the lost segment and calls `cm_request()`.

TCP/CM Implementation. The integration of TCP and the CM required less than 100 lines of changes to the existing TCP code, demonstrating both the flexibility of the CM API and the low programmer overhead of implementing a complex protocol with the Congestion Manager.

3.3 Congestion-controlled UDP sockets

The CM also provides congestion-controlled UDP sockets. They provide the same functionality as standard Berkeley UDP sockets, but instead of immediately sending the data from the kernel packet queue to lower layers for transmission, the buffered socket implementation schedules its packet output via CM callbacks. When a CM UDP socket is created, it is bound to a particular flow. When data enters the packet queue, the kernel calls `cm_request()` on the flow associated with the socket. When the CM schedules this flow for transmission, it calls `udp_ccappsend()` in the CM UDP module. This function transmits one MTU from the packet queue, and requests another callback if packets remain. The in-kernel implementation of the CM UDP API adds no data copies or queue structures, and supports all standard UDP options. Modifying existing applications to use this API requires only providing feedback to the CM, and setting a socket option on the socket.

A typical client of the CM UDP sockets will behave as follows, after its usual network socket initialization:

```
flow = cm_open(dst, port)
setsockopt(flow, ..., CM_BUF)
loop:
  <send data on flow>
  <receive data acknowledgements>
  cm_update(flow, sent, received, ...)
```

3.4 Streaming Layered Audio and Video

Streaming layered audio or video applications that have a number of discrete rates at which they can transmit data are well-served by the CM rate callbacks. Instead of requiring a comparatively expensive notification for each transmission, these applications are instead notified only in the rare event that their network conditions change significantly. Layered applications open their usual UDP socket, and call `cm_open()` to obtain a control socket. They operate in their own clocked event loop while listening for status changes on either their control socket or via a SIGIO signal. They use `cm_thresh()` to inform the CM about network changes for which they should receive callbacks.

3.5 Real-time Adaptive Applications

Applications that desire last-minute control over their data transmission (i.e. those that do not want any buffering inside the kernel) use the request callback API provided by the CM. When given permission to transmit via the `cmapp_send()` callback from the CM, they may use `cm_query()` to discover the current network conditions and adapt their content based on that. Other servers may simply wish to send the most up-to-date content possible, and so will defer their data collection until they know they can send it. The rough sequence of CM calls that are made to achieve this in the application are:

```
flow = cm_open(dst)
cm_request(flow)
<receive cmapp_send() callback from libcm>
cm_query(flow, ...)
<send data>
<receive data acks>
cm_update(flow, sent, lost, ...)
```

Other options exist for applications that wish to exploit the unique nature of their network utilization to reduce the overhead of using the services of the Congestion Manager. We discuss one such option below in the manner in which we adapted the `vat` interactive audio application to use the CM.

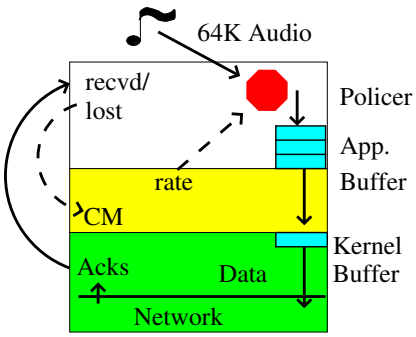

Figure 2. The adaptive vat architecture

3.6 Interactive Real-time Audio

The vat application provides a constant bit-rate source of interactive audio. Its inability to downsample its audio reduces the avenues it has available for bandwidth adaptation. Therefore, the best way to make vat behave in a network-friendly and backwards compatible manner is to preemptively drop packets to match the available network bandwidth. There are, of course, complications. Network applications experience two types of variation in available network bandwidth: long term variations due to changes in actual bandwidth, and short term variations due to the probing mechanisms of the congestion control algorithm. Short-term variation is typically dealt with by buffering. Unfortunately, buffering, especially FIFO buffering with drop-tail behavior, the de-facto standard for kernel buffers and network router buffers, can result in long delay and significant delay variation, both of which are detrimental to vat's audio quality. `Vat` , therefore, needs to act like an ALF application, managing its own buffer space with drop-from-head behavior when the queue is full.

The resulting architecture is detailed in figure 2. The input audio stream is first sent to a policer, which provides long-term adaptation via preemptive packet dropping. The policer outputs into the application level buffer, which can be configured in various sizes and drop policies. This buffer feeds into the kernel buffer on-demand as packets are available for transmission.

4 Evaluation

This section describes several experiments that quantify the costs and benefits of our CM implementation. Our experiments show that using the Congestion Manager in the kernel has minimal costs, and that even the worst-case overhead of the request/callback user-space API is acceptably small.

The tests were performed on the Utah Network Testbed [22] using 600MHz Intel Pentium III processors, 128MB PC100 ECC SDRAM, and Intel EtherExpress Pro/100B Ethernet cards, connected via 100Mbps

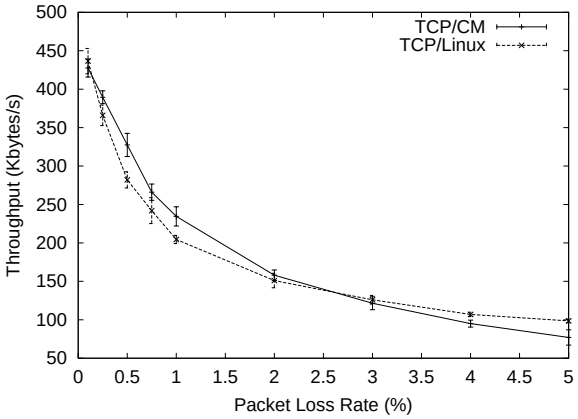

Figure 3. Comparing throughput vs. loss for TCP/CM and TCP/Linux. Rates are for a 10Mbps link with a 60ms RTT.

Ethernet through an Intel Express 510T switch, with Dummynet channel simulation. CM tests were run on Linux 2.2.9, with Linux and FreeBSD clients.

To ensure the proper behavior of a flow, the congestion control algorithm must behave in a "TCP-compatible" [8] manner. The CM implements a TCP-style window-based AIMD algorithm with slow start. It shares bandwidth between eligible flows in a round-robin manner with equal weights on the flows.

Figure 3 shows the throughput achieved by the Linux TCP implementation (TCP/Linux) and TCP with congestion control performed by the CM (TCP/CM). The linux kernel against which we compare has two algorithmic differences from the Congestion Manager: It starts its initial window at 2 packets, and it assumes that each ACK is for a full MTU. The Congestion Manager instead performs byte-counting for its AIMD algorithm. The first issue is Linux-specific, and the last is a feature of the CM.

4.1 Kernel Overhead

To measure the kernel overhead, we measured the CPU and throughput differences between the optimized TCP/Linux and TCP/CM. The midrange machines used in our test environment are sufficiently powerful to saturate a 100Mbps Ethernet with TCP traffic.

There are two components to the overhead imposed by the congestion manager: The cost of performing accounting as data is exchanged on a connection, and a one-time connection setup cost for creating CM data structures. A microbenchmark of the connection establishment time of a TCP/CM vs. TCP/Linux indicates that there is no appreciable difference in connection setup times.

We used long (megabytes to gigabytes) connections with the `ttcp` utility to determine the long-term costs

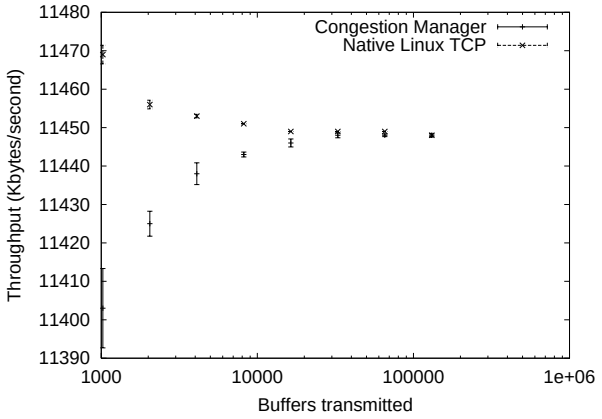

Figure 4. 100Mbps TCP throughput comparison. Note that the absolute difference in the worst case between the Congestion Manager and the native TCP is only 0.5% and that the Y axis begins at 11 megabytes per second.

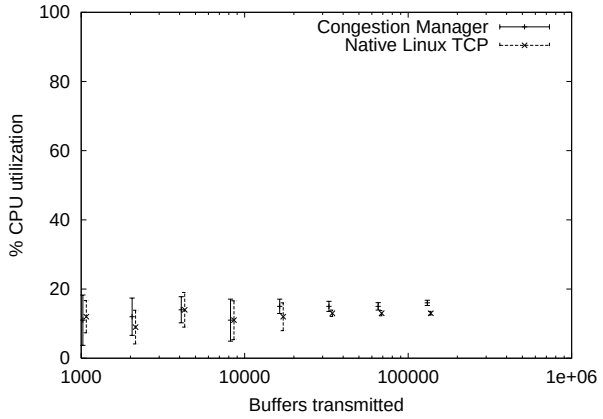

Figure 5. CPU overhead comparison between TCP/Linux and TCP/CM. For long connections, the CPU overhead converges to slightly under 1% for the unoptimized implementation of the CM.

imposed by the congestion manager. The impact of the CM on extremely long term throughput was negligible: in a 1 gigabyte transfer, the congestion manager achieved identical performance (91.6 Mbps) as native Linux. On shorter runs, the throughput of the CM diverged slightly from that of Linux, but only by 0.5%. The throughput rates are shown in figure 4. The difference is due to the CM using an initial window of 1 MTU and Linux using 2 MTU, not CPU overhead.

Because both implementations are able to saturate the network connection, we looked at the CPU utilization during these transmissions to determine the steady-state overhead imposed by the Congestion Manager. In figure 5 we see that the CPU difference between TCP/Linux and TCP/CM converges to slightly less than 1%.

4.2 User-space API Overhead

The overhead incurred by our adaptation API occurs primarily because the applications must process their ACKs in user-space instead of in the kernel. Therefore, these programs incur extra data copies and user/kernel boundary crossings. To quantify this overhead, our test programs sent packets of specified sizes on a UDP socket, and waited for acknowledgement packets from the server. We compare these programs to a webserver-like TCP client which sendt data to the server, and performed a `select()` on its socket to determine if the server has sent any data back. To facilitate comparison, we disabled delayed ACKs for the one TCP test to ensure that our packet counts were identical.

Figure 6 shows the wall-clock time required to send and process the acknowledgement for a packet, based on transmitting 200,000 packets. For comparison, we include TCP statistics as well, where the TCP programs set the maximum segment size to achieve identical network performance. The "nodelay" variant is TCP without delayed acks. The tests were run on a 100Mbps network on which no losses occured.

ALF/noconnect	1 `cm_notify` (ioctl)
ALF	1 `cm_request` (ioctl) 1 extra socket
Buffered	1 `recv`, 2 `gettimeofday`
TCP/CM	–baseline–

Table 1. Cumulative sources of overhead for different APIs using the Congestion Manager relative to sending data with TCP.

Table 1 breaks down the sources of overhead for using the different APIs. Using the CM with UDP requires that applications compute the round-trip-time (RTT) of their packets, requiring a system call to `gettimeofday`, and requires that they process their ACKs in user-space, requiring a system call to `recv` and the accompanying data copy into their address space. The ALF API further requires that the application obtain an additional control socket and select upon it, and that it make an explicit call to `cm_request` before transmitting data. Finally, if the kernel is unable to determine the flow to which to charge the transmission, as with an unconnected UDP socket, the application must explicitly call `cm_notify`

These test cases represent the worst-case behavior of serving a single high-bandwidth client, because no aggregation of requests to the CM may occur. The CM programs can achieve similar reductions in processing time by using delayed acks, so the real API overhead can be determined by comparing the ALF/noconnect case to the TCP/CM case. For 168 byte packets,

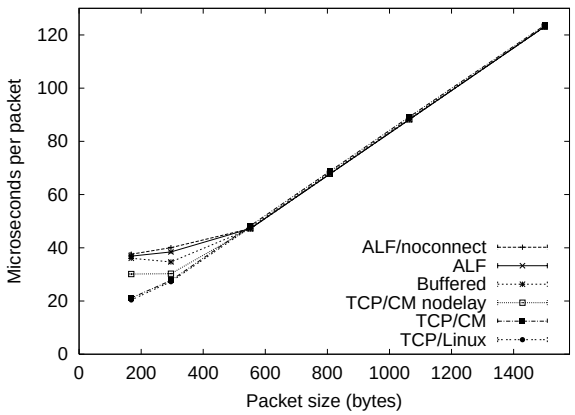

Figure 6. API throughput comparison on a 100Mbps link. The worst-case throughput reduction incurred by the CM is 25% from TCP/CM nodelay to ALF/noconnect.

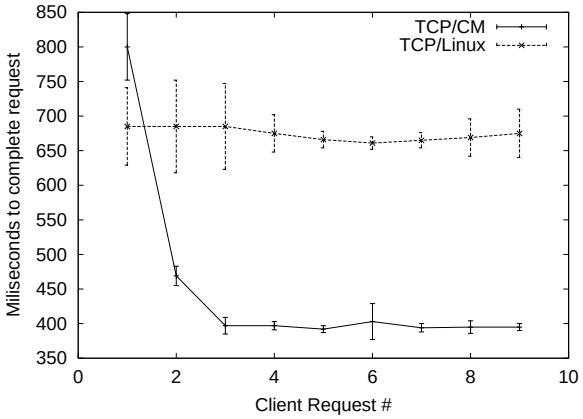

Figure 7. Sharing TCP state: The client requests the same file 9 times with a 500ms delay between request initiations. By sharing congestion information and avoiding slow-start, the CM-enabled server is able to provide faster service for subsequent requests, despite a smaller initial congestion window.

ALF/noconnect results in a 25% reduction in throughput relative to TCP without delayed ACKs.

4.3 Benefits of Sharing

One benefit of integrating congestion information with the CM is immediately clear. A client that sequentially fetches files from a webserver with a new TCP connection each time loses its prior congestion information, but with concurrent connections with the CM, the server is able to use this information to start subsequent connections with more accurate congestion windows. Figure 7 shows a test we performed across the vBNS between MIT and the University of Utah, where an unmodified (non-CM) client performed 9 retrievals of the same 128k file with a 500ms delay between retrievals, resulting in a 40% improvement in the transfer time for the later requests. (Other file sizes and delays yield similar results, so long as they overlap. The benefits are comparatively greater for smaller files). The CM requires an additional RTT (75ms) for the first transfer, because Linux sets its initial congestion window to 2 MTUs instead of 1. This pattern of multiple connections is still quite common in webservers despite the adoption of persistent connections: Many browsers open 4 concurrent connections to a server, and many client/server combinations do not support persistent connections. Persistent connections [28] provide similar performance benefits, but suffer from their own drawbacks, which we discuss in section 6.

4.4 Adaptive Applications

In this section, we demonstrate some of the network adaptive behaviors enabled by the CM.

As noted earlier, applications that require tight control over data scheduling use the request/callback (ALF) API, and are notified by the CM as soon as they can transmit data. The behavior of an adaptive layering application run across the vBNS using this API is shown in figure 8. This application chooses a layer to transmit based upon the current rate, but sends packets as rapidly as possible to allow its client to buffer more data. We see that the CM is able to provide sufficient information to the application to allow it to adapt properly to the network conditions.

For self-clocked applications that base their transmitted data upon the bandwidth to the client (such as conventional layered audio servers), the CM rate callback mechanism provides a low-overhead mechanism for adaptation, and allows clients to specify threshholds for the notification callbacks. Figure 9 shows application adaptation using rate callbacks for a connection between MIT and the University of Utah. Here, the application decides which of the four layers it should send based on notifications from the CM about rate changes.

From figures 8 and 9, we see from the increased oscillation rate in the transmitted layer that the ALF application is more responsive to smaller changes in available bandwidth, whereas the rate callback application relies occasionally on short-term kernel buffering for smoothing. There is an overhead vs. functionality trade-off in the decision of which API to use, given the higher overhead of the ALF API, but applications face a more important decision about the behavior they desire.

Some applications may be concerned about the overhead from receiver feedback. To mitigate this, an application may delay sending feedback; we see this in a minor and inflexible way with TCP delayed acks. In

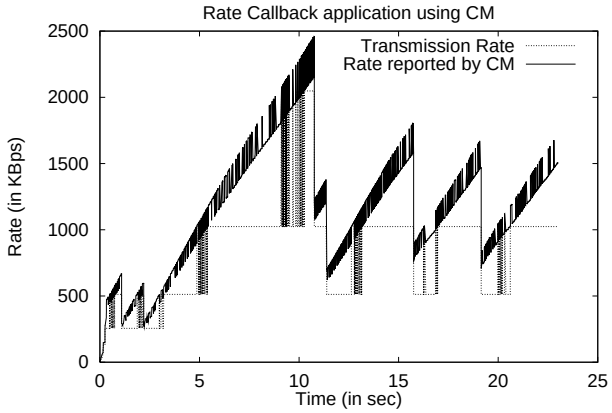

Figure 8. Bandwidth perceived by an adaptive layered application using the request callback (ALF) API.

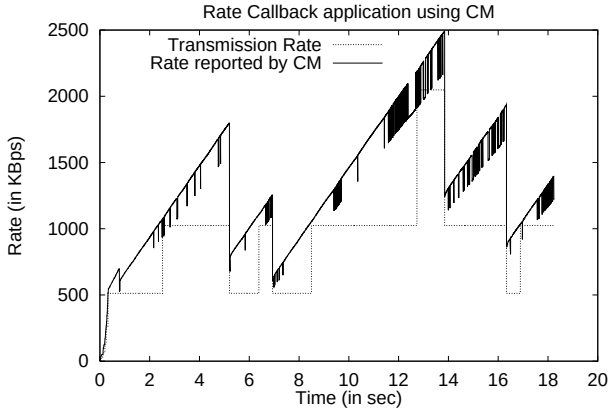

Figure 9. Bandwidth perceived by an adaptive layered application using the rate callback API.

figure 10, we see that delaying feedback to the CM causes burstiness in the reported bandwidth. Here, the feedback by the receiver was delayed by min(500 acks, 2000ms). The initial slow start is delayed by 2s waiting for the application, then the update causes a large rate change. Once the pipe is sufficiently full, 500 acks come relatively rapidly, and the normal, though bursty, non-timeout behavior resumes.

5 Discussion

We have shown several benefits of integrated flow management and the adaptation API, and have explored the design features that make the API easy to use. This section describes an optimization useful for busy servers, and discusses some drawbacks and limitations of the current CM architecture.

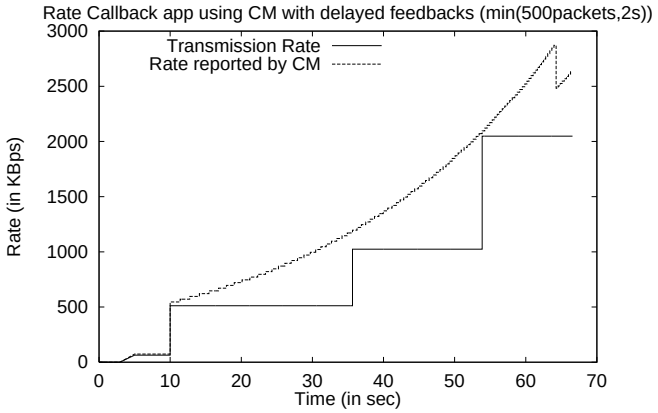

Figure 10. Adaptive layered application using rate callback API with delayed feedback

Optimizations. Servers with large numbers of concurrent clients are often very sensitive to the overhead caused by multiple kernel boundary crossings. To reduce this overhead, we can batch several sockets into the same `cm_request` call with the `cm_bulk_request` call, and likewise for `query`, `notify`, and `update` calls.

By multiplexing control information for many sockets on each CM call, the overhead from kernel crossings is mitigated at the expense of managing more complicated data structures for the CM interface. Bulk querying is already performed in `libcm` when multiple flows are ready during a single `ioctl` to determine which flows can send data, but this completes the interface.

Trust issues. Because our goal was an architecture that did not require modifications to receivers, we devised a system where applications provide feedback using the `cm_update()` call. The consequence of this is that there is a potential for misuse, due to bugs or malice. For example, the CM client could repeatedly misinform the CM about the absence of congestion along a path and obtain higher bandwidth. This does not increase the vulnerability of the Internet to such problems, because such abuse is already trivial. More important are situations where users on the same machine could potentially interfere with each other. To prevent this, the Congestion Manager would need to ensure that only kernel-mediated (e.g. TCP) flows belonging to different users can belong in the same macroflow. Our current implementation does not make an attempt to provide this protection. Savage [37] presents several methods by which a malicious receiver can defeat congestion control. The solutions he proposes can be easily used with the CM; we have already implemented byte-counting to prevent ACK division.

Macroflow construction. When differentiated services, or any system which provides different service to flows between the same pair of hosts, start being deployed, the CM would have to reconsider the default

choice of a macroflow. We expect to be able to gain some benefit by including the IP differentiated-services field in deciding the composition of a macroflow.

Finally, we observe that remote LANs are not often the bottleneck for an outside communicator. As suggested in [42, 36] among others, aggregating congestion information about remote sites with a shared bottleneck and sharing this information with local peers may benefit both users and the network itself. A macroflow may thus be extended to cover multiple destination hosts behind the same shared bottleneck link. Efficiently determining such bottlenecks remains an open research problem.

Limitations. The current CM architecture is designed only to handle unicast flows. The problem of congestion control for multicast flows is a much more difficult problem which we deliberately avoid. UDP applications using the CM are required to perform their own loss detection, requiring potential additional application complexity. Implementing the Congestion Manager protocol discussed in [3] would eliminate this need, but remains to be studied.

6 Related work

Designing adaptive network applications has been an active area of research for the past several years. In 1990, Clark and Tennenhouse [11] advocated the use of *application-level framing* (ALF) for designing network protocols, where protocol data units are chosen in concert with the application. Using this approach, an application can have a greater influence over deciding how loss recovery occurs than in the traditional layered approach. The ALF philosophy has been used with great benefit in the design of several multicast transport protocols including the Real-time Transport Protocol (RTP) [38], frameworks for reliable multicast [14, 33], and Internet video [24, 35].

Adaptation APIs in the context of mobile information access were explored in the Odyssey system [26]. Implemented as a user-level module in the NetBSD operating system, Odyssey provides API calls by which applications can manage system resources, with upcalls to applications informing them when changes occur in the resources that are available. In contrast, our CM system is implemented in-kernel since it has to manage and share resources across applications (e.g., TCP) that are already in-kernel. This necessitates a different approach to handling application callbacks. In addition, the CM approach to measuring bandwidth and other network conditions is tied to the congestion avoidance and control algorithms, as compared to the instrumentation of the user-level RPC mechanism in Odyssey. We believe that our approach to providing adaptation information for bandwidth, round-trip time, and loss rate complements Odyssey's management of disk space, CPU, and battery power.

The CM system uses application callbacks or *upcalls* as an abstraction, an old idea in operating systems. Clark describes upcalls in the Swift operating system, where the motivation is a lower layer of a protocol stack synchronously invoking a higher-layer function across a protection boundary [9]. The Mach system used the notion of *ports*, a generic communication abstraction for fast inter-process communication (IPC). POSIX specifies a standard way of passing "soft real-time signals" that can be used to send a notification to a user-level process, but it restricts the amount of data that can be communicated to a 32-bit quantity.

Event delivery abstractions for mobile computing have been explored in [1], where "monitored" events are tracked using polling and "triggered" events (e.g., PC card insertion) are notified using IPC. This work defines a language-level mechanism based on C++ objects for event registration, delivery, and handling. This system is implemented in Mach using ports for IPC.

Our approach is to use a `select()` call on a control socket to communicate information between kernel and user-level. The recent work of Banga *et al.* [4] to improve the performance of this type of event delivery can be used to further improve our performance.

The Microsoft Winsock implementation is largely callback-based, but here callbacks are implemented as conventional function calls since Winsock is a user-level library within the same protection boundary as the application [31]. The main reason we did not implement the CM as a user-level daemon was because TCP is already implemented in-kernel in most UNIX operating systems, and it is important to share network information across TCP flows.

Quality-of-service (QoS) interfaces have been explored in several operating systems, including Nemesis [16]. Like the exokernel approach [20] and SPIN [7], Nemesis enables applications to perform as much of the processing as possible on their own using application-specific policy, supported by a set of operating system abstractions different from those in UNIX. Whereas Nemesis treats local network-interface bandwidth as the resource to be managed, we take a more end-to-end approach of discovering the end-to-end performance to different end-hosts, enabling sharing across common network paths. Furthermore, the API exported by Nemesis is useful for applications that can make resource reservations, while the CM API provides information about network conditions. Some "web switches" [**?**] provide traffic shaping and QoS based upon application information, but do not provide integrated flow management or feedback to the applications creating the data.

Multiple concurrent streams can cause problems for TCP congestion control. First, the ensemble of flows probes more aggressively for bandwidth than a single flow. Second, upon experiencing congestion along the path, only a subset of the connections usually reduce their window. Third, these flows do not share any information between each other. While we propose a gen-

eral solution to these problems, application-specific solutions have been proposed in the literature. Of particular importance are approaches that multiplex several logically distinct streams onto a single TCP connection at the application level, including Persistent-connection HTTP (P-HTTP [28], part of HTTP/1.1 [12]), the Session Control Protocol (SCP) [39], and the MUX protocol [15]. Unfortunately, these solutions suffer from two important drawbacks. First, because they are application-specific, they require each class of applications (Web, real-time streams, file transfers, etc.) to reimplement much of the same machinery. Second, they cause an undesirable coupling between logically different streams: if packets belonging to one stream are lost, another stream could stall even if none of its packets are lost because of the in-order "linear" delivery forced by TCP. Independent data units belonging to different streams are no longer independently processible and the parallelism of downloads is often lost.

7 Conclusion

The CM system enables applications to obtain an unprecedented degree of control over what they can do in response to different network conditions. It incorporates robust congestion control algorithms, freeing each application from having to re-implement them. It exposes a rich API that allows applications to adapt their transmissions at a fine-grained level, and allows the kernel and applications to integrate congestion information across flows.

Our evaluation of the CM implementation shows that the callback interface is effective for a variety of applications, and does not unduly burden the programmer with restrictive interfaces. From a performance standpoint, the CM itself imposes very little overhead; that which remains is mostly due to the unoptimized nature of our implementation. The architecture of programs implemented using UDP imposes some additional overhead, but the cost of using the CM after this architectural conversion is quite small.

Many systems exist to deliver content over the Internet using TCP or home-grown UDP protocols. We believe that by providing an accessible, robust framework for congestion control and adaptation, the Congestion Manager can help improve both the implementation and performance of these systems.

The Congestion Manager implementation for Linux is available from our web page, `http://nms.lcs.mit.edu/projects/cm/`.

Acknowledgements

We would like to thank Suchitra Raman and Alex Snoeren for their helpful feedback and suggestions, and the Flux research group at the University of Utah for providing their network testbed[3]. Finally, we would like to thank our shepherd, Peter Druschel, and the anonymous reviewers for their numerous helpful comments. This work was supported by grants from DARPA (Grant No. MDA972-99-1-0014), NSF, IBM, Intel, and NTT Corporation.

References

[1] Badrinath, B. R., and Welling, G. Event Delivery Abstraction for Mobile Computing. Tech. Rep. LCSR-TR-242, Rutgers University, 1995.

[2] 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* (San Francisco, CA, Mar. 1998).

[3] Balakrishnan, H., Rahul, H. S., and Seshan, S. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. ACM SIGCOMM* (Sep 1999).

[4] Banga, G., Mogul, J. C., and Druschel, P. A scalable and explicit event delivery mechanism for unix. In *Proc. of the USENIX 1999 Annual Technical Conference* (June 1999).

[5] Barrera, J. S. A fast Mach network IPC implementation. In *Proc. of the Second USENIX Mach Symposium* (Nov. 1991), pp. 1–12.

[6] Berners-Lee, T., Fielding, R., and Frystyk, H. *Hypertext Transfer Protocol–HTTP/1.0.* Internet Engineering Task Force, May 1996. RFC 1945.

[7] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility, safety, and performance in the SPIN operating system. pp. 267–284.

[8] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and Zhang, L. *Recommendations on Queue Management and Congestion Avoidance in the Internet.* Internet Engineering Task Force, Apr 1998. RFC 2309.

[9] Clark, D. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)* (Dec. 1985), pp. 171–180.

[10] Clark, D. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM* (Aug. 1988).

[11] Clark, D., and Tennenhouse, D. Architectural Consideration for a New Generation of Protocols. In *Proc. ACM SIGCOMM* (September 1990).

[3] Supported by NSF grant ANI-00-82493, DARPA/AFRL grant F30602-99-1-0503 and Cisco.

[12] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners-Lee, T. *Hypertext Transfer Protocol—HTTP/1.1.* Internet Engineering Task Force, Jan 1997. RFC 2068.

[13] Floyd, S., and Fall, K. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Trans. on Networking 7*, 4 (Aug. 1999).

[14] Floyd, S., Jacobson, V., McCanne, S., Liu, C. G., and Zhang, L. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proc. ACM SIGCOMM* (Sept. 1995).

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

[16] I. Leslie and D. McAuley and R. Black and T. Roscoe and P. Barham and D. Evers and R. Fairbairns and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications 14*, 7 (September 1996), 1280–1297.

[17] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Programming Interface (API) — Amendment 1: Realtime Extension [C Language]*, 1994. Std 1003.1b-1993.

[18] Jacobson, V. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM* (Aug 1988).

[19] Jacobson, V., Braden, R., and Borman, D. *TCP Extensions for High Performance.* Internet Engineering Task Force, May 1992. RFC 1323.

[20] Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (Saint-Malô, France, October 1997), pp. 52–65.

[21] Karn, P., and Partridge, C. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems 9*, 4 (Nov. 1991), 364–373.

[22] Lepreau, J., Alfeld, C., Andersen, D., and Maren, K. V. A large-scale network testbed. Unpublished, in 1999 SIGCOMM works-in-progress. `http://www.cs.utah.edu/flux/testbed/`, Sept. 1999.

[23] The MASH Project Home Page. `http://www-mash.cs.berkeley.edu/mash/`, 1999.

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

[25] Microsoft Windows Media Player. `http://www.microsoft.com/windows/mediaplayer/`.

[26] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J., Flinn, J., and Walker, K. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles* (Oct 1997).

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

[28] Padmanabhan, V. N., and Mogul, J. C. Improving HTTP Latency. In *Proc. Second International WWW Conference* (Oct. 1994).

[29] Postel, J. B. *User Datagram Protocol.* Internet Engineering Task Force, August 1980. RFC 768.

[30] Postel, J. B. *Transmission Control Protocol.* Internet Engineering Task Force, September 1981. RFC 793.

[31] Quinn, B., and Shiute, D. *Windows Sockets Network Programming.* Addison-Wesley, Jan. 1999.

[32] Ramakrishnan, K., and Floyd, S. *A Proposal to Add Explicit Congestion Notification (ECN) to IP.* Internet Engineering Task Force, Jan 1999. RFC 2481.

[33] Raman, S., and McCanne, S. Scalable Data Naming for Application Level Framing in Reliable Multicast. In *Proc. ACM Multimedia* (Sept. 1998).

[34] Real Networks. `http://www.real.com/`.

[35] Rejaie, R., Handley, M., and Estrin, D. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. IEEE INFOCOM* (March 1999).

[36] Savage, S., Cardwell, N., and Anderson, T. The Case for Informed Transport Protocols. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS VII)* (Mar 1999).

[37] Savage, S., Cardwell, N., Wetherall, D., and Anderson, T. TCP Congestion Control with a Misbehaving Receiver. In *ACM Computer Comm. Review* (Oct 1999).

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

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

[40] Stevens, W. R. *TCP/IP Illustrated, Volume 1.* Addison-Wesley, Reading, MA, Nov 1994.

[41] Tan, W., and Zakhor, A. Real-time Internet Video Using Error Resilient Scalable Compression and TCP-friendly Transport Protocol. *IEEE Trans. on Multimedia* (May 1999).

[42] Touch, J. *TCP Control Block Interdependence.* Internet Engineering Task Force, April 1997. RFC 2140.