Unified Congestion Control for Unreliable Transport Protocols

by

Hariharan Shankar Rahul

Bachelor of Technology (Computer Science and Engineering)
Indian Institute of Technology, Madras, India (1997)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

~~August 1999~~
[September 1999]

Author .... H Rahul ....
Department of Electrical Engineering and Computer Science
August 13, 1999

Certified by .. Hari Balakrishnan ....
Hari Balakrishnan
Assistant Professor
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Unified Congestion Control for Unreliable Transport Protocols

by

Hariharan Shankar Rahul

Submitted to the Department of Electrical Engineering and Computer Science
on August 13, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

There is an increasing number of Internet applications that do not require reliable data transfer, and hence do not solicit feedback from receivers in the form of acknowledgments. However, receiver feedback about received and lost packets is essential for proper network behavior in the face of congestion. In this thesis, we propose an application-independent feedback scheme for unreliable flows running over the User Datagram Protocol. Our scheme is lightweight, adaptive and robust in the face of network losses. We implement our scheme in the context of an end-to-end congestion management infrastructure, the Congestion Manager. The Congestion Manager integrates congestion management across multiple transports and applications, and exports a general programming interface that allows applications to adapt to congestion. We evaluate the performance of our scheme through both simulation, as well as wide-area measurements using our user-level implementation of the Congestion Manager. Our feedback scheme performs efficiently across a variety of bottleneck bandwidths, and in the presence of significant congestion and cross traffic.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Acknowledgments

I am deeply indebted to my advisor, Prof. Hari Balakrishnan for his guidance and suggestions throughout the duration of this thesis. His enthusiasm for research, efficiency, and excellent writing and presentation skills have been a source of inspiration.

The research described in this thesis is joint work with my advisor and Dr. Srini Seshan of the IBM T. J. Watson Research Laboratory. I thank them for their collaboration and look forward to further interaction with them in the future.

Prof. John Guttag was kind enough to support me when I was still in search of a research direction, and has been a wellspring of advice. I would like to express my gratitude to him. I would also like to acknowledge the members of the SDS group, especially Ulana Legedza and Dave Wetherall for helping me find my feet during my first term at MIT.

My stay at MIT has been enriched by the many friends I have made and hope to keep in the coming years. Bodhi Priyantha, Suchitra Raman and Elliot Schwartz have been great officemates. I will always remember the discussions we have had, academic and otherwise. Chandra Boyapati, Raj Iyer, Dina Katabi and Joanna Kulik made LCS a really fun place to work and play.

I would like to thank my uncle, aunt and grandmother for providing me a home away from home. Saving the best for last, I must express my sincerest gratitude to my parents for always supporting, loving and believing in me. I dedicate this thesis to them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The rapid growth in the popularity of the Internet (shown in Figure 1-1 has been a prominent trend in networking and communication in the 1990s. Many of the currently popular applications like Web transfer [3], electronic mail [26] and ftp [27] require reliable data transfer and use the Transmission Control Protocol (TCP) [15], which provides applications with a reliable, ordered, exactly-once byte stream abstraction over a network that can drop, reorder or duplicate packets. An important consequence of using TCP as the transport protocol is that it solicits feedback from receivers not only for reliability, but also to regulate the sending rate of sources depending on congestion in the network.

However the diversity and richness of content sought by users of any public medium, including the Internet, increases with its popularity. While the chief content type on the Internet has been hyperlinked text and images, there has been an increase in audio and video traffic in recent times. It is anticipated that traffic due to video and audio will constitute a significant proportion of all future Internet traffic. These types of traffic have real-time requirements which are not well served by TCP; delaying data for in-order delivery can often cause it to be worthless for these application. Hence, these applications use the User Datagram Protocol (UDP) [36] which promises best-effort delivery.

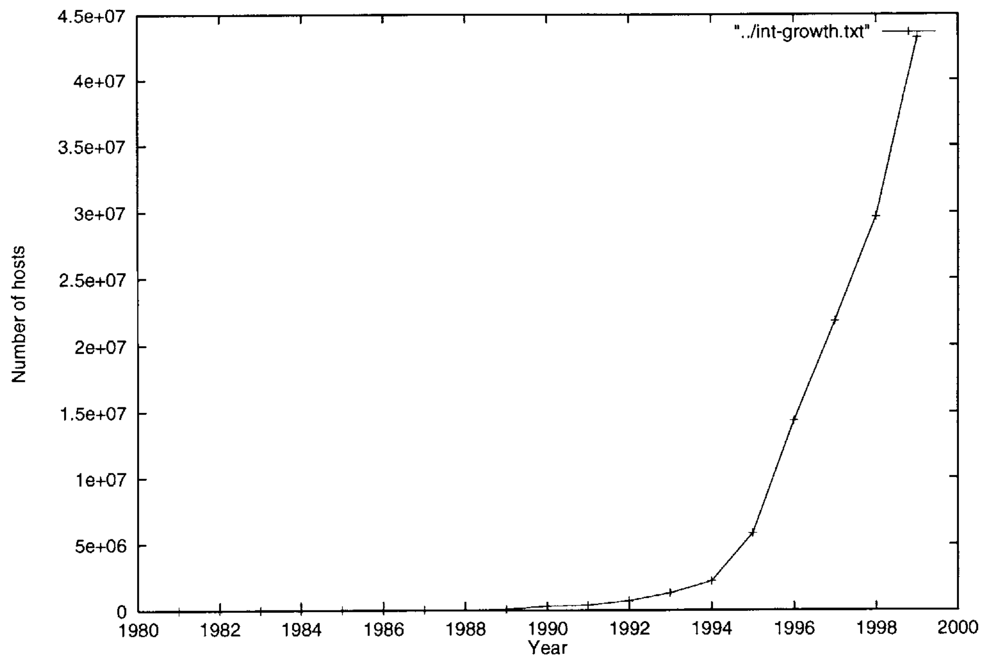

Figure 1-1: Rapid growth of number of hosts on the Internet with time (Data obtained from [14]).

Many applications that use UDP do not require per-packet reliability and hence do not solicit feedback from receivers for this purpose. However, receiver feedback is essential for senders to detect and adapt to network congestion. Open-loop operation is dangerous as the stability of the Internet critically depends on traffic adapting to network congestion, and inelastic sources can cause widespread network collapse.

Current Internet protocol stacks do not provide any convenient method for UDP applications to detect or adapt to congestion. In many cases, application writers are forced to implement this mechanism independently for each application or use inappropriate protocols like TCP. The goal of our work, therefore, is to provide a scheme to allow detection and adaptation to network congestion by applications which do not solicit feedback. We have the following design goals for our feedback scheme.

Application independence: This will allow application writers to focus on the mechanisms for congestion adaptation, rather than congestion detection. Additionally, a scheme which can be implemented independent of individual applications can exploit global knowledge, for example, it can integrate congestion information across flows to the same destination host.

Adaptation: It is important that the feedback scheme work across and adapt to a wide range of round-trip times and bottleneck bandwidths.

Robustness: The feedback scheme should work correctly and efficiently when control packets are lost. This is critical as the feedback protocol is especially important during epochs of congestion and loss in the network.

1.2 Brief Description

Our feedback scheme is designed to meet the goals listed in the previous section. It consists of a feedback module at the sender, and a response module at the receiver which are implemented independent of all applications. Applications notify the sender feedback module whenever data is transmitted, and the receiver response module whenever data is received. The sender feedback module sends periodic probes to the receiver module and solicits responses from the receiver module. The frequency of the probes adapts to the round-trip time of the connection. While this induces a bias towards connections with short round-trip times, we adopt this policy in order to mimic the behavior of TCP, which is the most widely deployed transport protocol. Our feedback scheme can tolerate losses of both probes and responses, and reacts correctly in the face of infrequent feedback.

We implement our feedback scheme in the context of the Congestion Manager [1], which provides an infrastructure for unified end-to-end congestion management. It integrates congestion information across all flows with the same destination address and detects and reacts to congestion events by appropriately adjusting its rate estimates. It exports a simple, general and flexible programming interface which exposes congestion information. The API is designed to allow the Congestion Manager to detect and react to network congestion, while giving applications maximum flexibility in adapting to congestion.

The chief contributions of this thesis are:

- The design of a robust, adaptive and application-independent feedback scheme for UDP flows.
- A user-level UNIX implementation of the Congestion Manager.
- Evaluation of our feedback scheme through simulation and wide-area Internet experiments.

1.3 Roadmap of Thesis

The rest of this thesis is organized as follows. Chapter 2 presents some background material in congestion management, describes related research and compares it to our framework. Chapter 3 presents the design criteria and our feedback protocol in Chapter 3. It also outlines the motivation and design of the Congestion Manager, as well as the details of our user-level implementation. We evaluate the feedback scheme through simulation studies as well as the performance of our implementation and present the results in Chapter 4. We conclude with a summary of our contributions and directions for future work in Section 5.

Chapter 2

Background and Related Work

This chapter discusses the related work in congestion control for unicast and multicast. We begin with a description of the Internet model, motivate the need for congestion control, and outline the basic principles in Section 2.1. Section 2.2 surveys the related work in congestion control algorithms and protocols.

2.1 The Need for Congestion Control

Figure 2-1 schematically shows the connectivity model of the network. Individual networks are interconnected through routers that *store* incoming packets and *forward* them along one of several output links. The rate of outgoing packets is limited by the bandwidth of the outgoing links which can in general be less than the bandwidth of

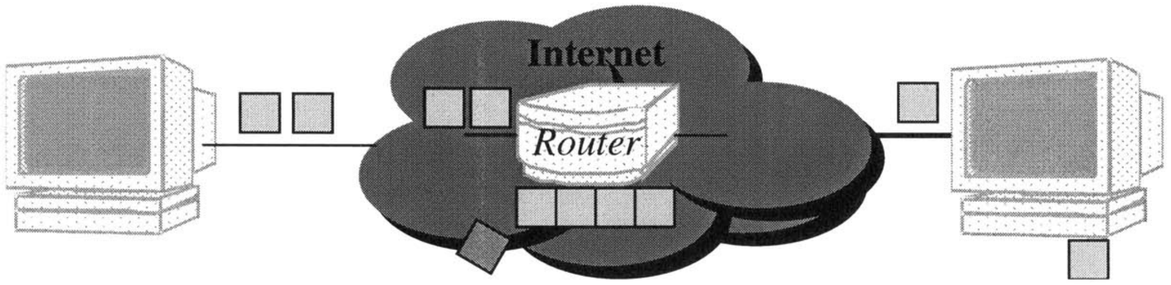

Figure 2-1: Schematic connectivity model of the Internet. Shared infrastructure and overload causes congestion.

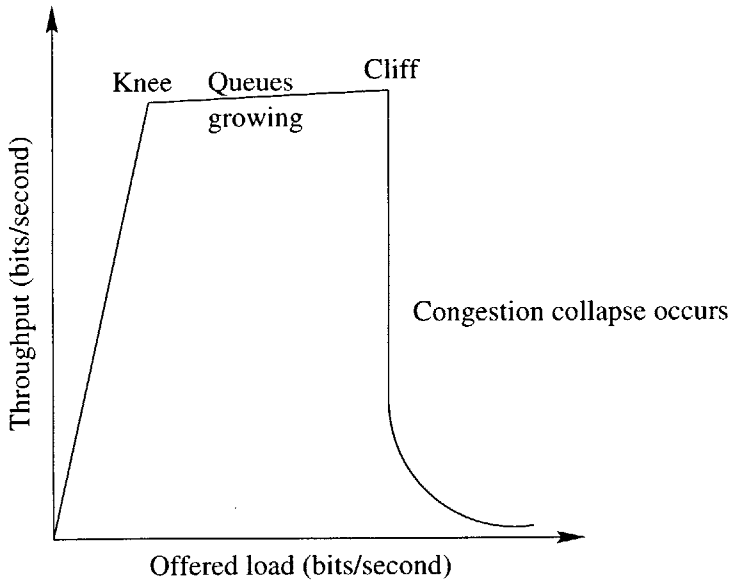

Figure 2-2: Variation of network throughput with offered load.

the incoming links. In order to accommodate short surges in network traffic, routers have queues (buffers) for temporary storage of packets. Queue lengths grow when the incoming rate exceeds the outgoing bandwidth and packets are dropped when the router detects congestion or the queues are full. Thus there is a contention among different flows for the shared resources of *link bandwidth* and *router buffers.* Network congestion occurs when demand for these shared resources exceeds what is available.

Overload severely degrades network throughput as shown in Figure 2-2 which plots throughput as a function of offered load. At low levels of load (to the left of the knee), throughput increases linearly with offered load as the network is not fully utilized. Throughput is maximum when the offered load is close to the bottleneck bandwidth and plateaus as queue lengths increase at the bottleneck router. As offered load is increased further, the throughput suddenly drops to a negligible amount at a *cliff* as all competing flows send data but no useful payload reaches the receiver as most packets are dropped.

A seemingly obvious solution to avoid network congestion is to overprovision the network to account for the maximum possible demand. However, the observed variance in demand is too high to allow overprovisioning without greatly reducing the average utilization of the network resources thus making this solution economically infeasible [34]. As a result, it is necessary to come up with *congestion control* mechanisms to maintain the network operating point to the left of the knee, and ensure that router queues are not overflowed. This is in addition to end-to-end *flow control* which attempts to ensure that the sender sends data at most as fast as the receiver can process it. Flow control is normally achieved by negotiation between the sender and the receiver.

2.1.1 Characteristics of Congestion Control Algorithms

For the following discussion, we will assume that the network is shared by n users. Time is divided into discrete slots at the beginning of which users set their load level based on feedback during the previous slot. The offered load of the ith user during time slot t is $X_i(t)$. The chief criteria for any algorithm for allocation of network resources among flows, as described in [5] are:

Efficiency: This is defined by the closeness of the total load on the resource to its knee. If X_{goal} denotes the desired load level at the knee, the resource is operating efficiently as long as $X(t) = \sum_{i=1}^{n} X_i(t)$ is sufficiently close to X_{goal}.

Fairness: When multiple users share a resource, all users in a similar class ought to have the equal share of the bottleneck. If allocation is not exactly equal, the degree of fairness is measured by the fairness index defined as:

$$F(X(t)) = \frac{(\sum_{i=1}^{n} X_i(t))^2}{n(\sum_{i=1}^{n} X_i(t)^2)}$$

Distributedness: This is necessary as a centralized scheme requires complete knowledge of the state of the network as well as individual flows,and cannot scale to a large network such as the Internet.

Convergence: Convergence is measured by the time taken till the system reaches its goal state from an arbitrary starting state. The system oscillates around the goal state once it is reached. Ideally, the system reaches the goal state quickly and has a small amplitude of oscillation around it.

2.1.2 Increase and Decrease Algorithms

Chiu and Jain [5] study different controls, linear and non-linear, for adjusting the values $X_i(t)$ to achieve the resource management goals listed in Section 2.1. They conclude that non-linear controls are extremely sensitive to system parameters and hence not robust. The linear control chosen is of the form

$$X_i(t+1) = a + bX_i(t)$$

In order to satisfy the goals listed in Section 2.1, it can be shown (if users are synchronized) that the appropriate equations are:

Additive Increase: $X_i(t+1) = X_i(t) + a,\ a > 0$

Multiplicative Decrease: $X_i(t+1) = bX_i(t),\ 0 \leq b < 1$

Thus, under certain assumptions, Additive-Increase/Multiplicative-Decrease (AIMD) converges to fairness and efficiency, and is the algorithm of choice for congestion control.

2.2 Related Work

This section discusses past work in the design of congestion control protocols. Several congestion controlled protocols for unicast have been proposed in the literature, of these we describe schemes which require router support, TCP [25] and Rate Adaptation Protocol [30] in some detail. We also briefly discuss other proposals for unicast congestion control, as well as multicast congestion control in Sections 2.2.4 and 2.2.5

These protocols can be window- or rate-based. Window-based protocols are generally acknowledgment-clocked; i.e., the arrival of a new acknowledgment triggers the sending of a new packet. Thus, the number of outstanding packets in the network is carefully controlled and can at most be equal to the window size. Moreover, the ack-processing and packet transmission together incur the cost of only one interrupt on the sender. However, window-based protocols tend to send packets in a burst. As an alternative, protocols where the sending rate is explicitly controlled have been proposed. However, in pure rate-based protocols, packet transmissions are triggered solely by timers at the sender which can sometimes make it a bottleneck. Moreover, a purely rate-regulated sender could cause an unbounded number of packets outstanding in the network.

2.2.1 Router Support

DECbit was one of the earliest congestion control schemes. In DECbit, the network provides feedback to allow senders to adjust the amount of traffic they transmit into the network. Routers monitor the average queue length in some defined time interval. If the average queue length exceeds a threshold, the router sets a congestion-indication bit in packets flowing in the forward direction. Receivers echo this bit in their acknowledgment to the sender. The sender monitors and stores these congestion-indication bits for a certain number of packets. If the fraction of congestion-indication bits which are set exceeds a threshold, the window size is decreased multiplicatively, else the window size is increased additively. The scheme is intended to ensure that the network operates in congestion-avoidance mode to the left of the knee in Figure 2-2. A more recent proposal suggests that routers send Explicit Congestion Notifications (ECN) [28] to senders during periods of congestion. For example, Random Early Detection (RED) gateways [12] can achieve this by marking packets with a certain probability when the queue length exceeds a threshold.

Another method to ensure proper network behavior is isolation of flows using a mechanism like fair queueing [8] so that conforming flows are not penalized by an aggressive sender. Mechanisms such as integrated services [6] and differentiated services [20] also attempt to provide guarantees per flow or traffic class and achieve a similar effect. However, all these schemes rely on modifications to routers which is a substantial change to the current infrastructure. In contrast, we focus on a scheme which requires only end-system changes.

2.2.2 Transmission Control Protocol

The Transmission Control Protocol (TCP) is the most popular protocol for reliable data transmission on the Internet today. In addition to congestion control, it also performs the functions of loss recovery and connection management. In this discussion, we will describe only the congestion control algorithms used in TCP.

Initialization:

```
cwnd = 1;
ssthresh = 65536;     // Transition point from slow start to
                      // congestion avoidance
```

Acknowledgement Arrival:

```
if (cwnd < ssthresh) // Slow Start
    cwnd += MSS;
else                  // Congestion Avoidance (Additive Increase)
    cwnd += MSS*MSS/cwnd;
```

Duplicate Acknowledgements:

```
cwnd /= 2;            // Multiplicative Decrease
ssthresh = cwnd;
```

Timeout:

```
ssthresh = cwnd/2;
cwnd = 1;
```

Figure 2-3: Pseudocode for congestion control and avoidance of TCP. Note that `cwnd` and `ssthresh` are measured in bytes and TCP additionally updates its estimate of the round-trip time upon ACK arrival.

TCP congestion management is based on the fundamental concepts of AIMD and is described in the seminal paper by Van Jacobson [15]. TCP uses a sliding-window protocol where the window determines the maximum number of packets which can be outstanding in the network. TCP performs flow control by ensuring that the sender's transmission window is less than the receiver's advertised window.

TCP performs congestion control by constantly adapting the window size based

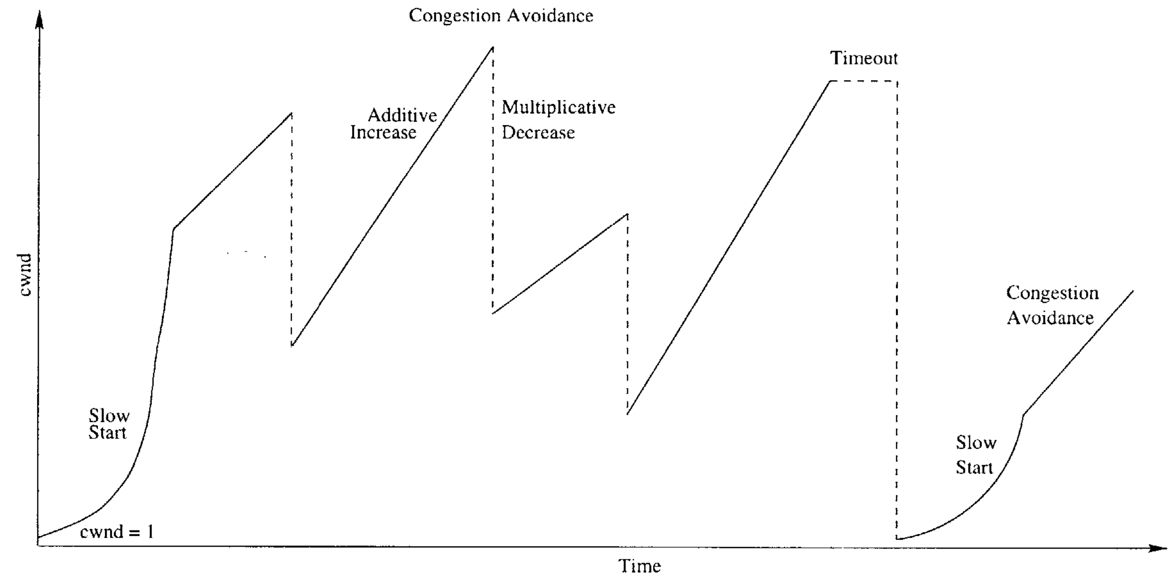

Figure 2-4: Evolution of the sender window as a function of time.

on its estimate of congestion in the network. Packet losses are assumed as indicators of congestion in the network. This is a justifiable assumption in most modern wired networks. There are two distinct modes in which TCP operates:

Slow Start: TCP is in slow start when a connection first starts sending data, or restarts after an idle period. The congestion window `cwnd` is initialized to one segment size. Each arriving ack increases the size of the congestion window by a segment size (MSS). When the congestion window size exceeds a slow start threshold `ssthresh`, TCP moves into the congestion avoidance phase.

Congestion Avoidance: TCP is expected to operate in congestion avoidance mode in its steady state. The congestion window is increased by one MSS for each successfully transmitted window. A packet loss causes halving of the congestion window. Additionally, `ssthresh` is set to half the size of the congestion window at the time of a loss. Persistent losses (detected by a timeout) reduce the congestion window size to one MSS and cause the connection to move back into slow start.

Figure 2-3 shows pseudocode describing the behavior of TCP. The evolution of the sender window as a function of time is plotted in Figure 2-4.

The actual sender window is set to the minimum of the congestion window and the receiver advertised window. In the steady state, TCP transmits one window of packets every round-trip. Since the congestion window tends to the product of the bottleneck bandwidth and round-trip delay, this implies that the long-term transmission rate of a TCP sender is equal to the bottleneck bandwidth. As described earlier, the use of the same window for both loss recovery and congestion control couples the two functions, and does not allow the use of TCP solely for congestion control. However, the prevalence of TCP has made it imperative that any new congestion control algorithm or protocol is fair to existing TCP flows in the network. This has led to the notion of TCP-friendliness [9]. The TCP-friendly equation relates the throughput T of a TCP connection with round-trip R, sending packets of size B bytes and an observed probability of packet loss p as:

$$T = \frac{1.5\sqrt{2/3} * B}{R\sqrt{p}}$$

It is believed that proposed protocols for congestion control should show a similar relationship between throughput and loss probability.

However, TCP does not share control information across connections, and hence concurrent connections to the same destination (a common case in Web traffic) compete, instead of cooperating for network bandwidth. HTTP/1.1 [10] addresses this issue by multiplexing several transfers onto a single long-lived TCP connection. However, this imposes a total order on all the packets belonging to logically different streams, and a packet loss on one stream can stall other streams even if none of their packets are lost. Our congestion-management infrastructure multiplexes congestion control information across flows to the same destination, and avoids undesirable coupling of logically independent streams.

2.2.3 Rate Adaptation Protocol

Rate Adaptation Protocol (RAP) is an end-to-end TCP friendly protocol which uses an AIMD algorithm. It is similar to our work in that it separates network congestion-control from application-level reliability. It is designed for unicast playback of real-time streams and other semi-reliable rate-based applications. While the internal rate-control algorithms used in RAP are similar to those used in our work, there are some key differences. The receiver-side in RAP acknowledges every packet transmitted by the sender. This might be too heavyweight for applications which do not require per-packet reliability. Moreover, RAP performs congestion control on a per-application basis whereas our congestion infrastructure unifies congestion information on a per-destination basis. Additionally, RAP does not provide a convenient and general API for applications to adapt to network congestion.

2.2.4 Other Unicast Congestion Control Protocols

Several other protocols have been proposed in the area of unicast congestion control. However, most of them do not fairly coexist with TCP flows. There are also some commercial media streaming players which are used over the Internet. Examples are Real Player [29] and Microsoft Windows Media Player [19]. While these claim to adapt to network congestion, no details or analysis are available to substantiate these claims.

2.2.5 Multicast Congestion Control

Recently, congestion control protocols have been proposed for real-time multimedia applications. RLM [17] describes a congestion control algorithm for layered streams. There is also a large body of work in congestion control for reliable multicast applications [31]. Unlike these application-specific proposals, our work attempts to provide a unified congestion-management layer and an API that allows applications to independently adapt to congestion. The problem of scalable feedback for multicast video distribution has been addressed in IVS [4]. The real-time transport protocol (RTP) and the associated real-time control protocol (RTCP) [33] describe end-to-end transport and monitoring functions for real-time streams. RTCP specifies packet formats for sender and receiver reports, as well as methods to calculate the interval between these packets. One key issue is that this interval is required to be at least 5 seconds, which does not allow fine-grained adaptation to congestion by the sender. This is unlike our scheme where the frequency of feedback adapts to the round-trip time of the connection.

2.3 Conclusion

The chapter began with a discussion of the Internet model and motivated the need for congestion control to prevent collapse of the network. We then presented a basic theoretical foundation of congestion-control algorithms. In Sections 2.2.1, 2.2.2 and 2.2.3, we discussed router support, TCP and RAP respectively. Section 2.2.4 discussed some other proposals for unicast congestion control and Section 2.2.5 presented proposals for multicast congestion control. We observe that many of these current approaches have problems like coupling congestion control with loss recovery, requiring changes in network infrastructure, or application-specificity.

Chapter 3

Congestion Feedback

This chapter describes the design and implementation of our feedback protocol for congestion-controlled unreliable transports.

Senders need periodic feedback about the network state so that they can adapt their rate of transmission. Feedback allows transmitters to increase their rate when the network has spare capacity, and decrease it when the network is congested. As described in Chapter 2, AIMD is used as the algorithm for updating rate estimates whenever feedback is obtained.

Packet loss or explicit notification by routers is used as the signal of congestion by most rate adaptation algorithms, and its absence is assumed to indicate the availability of bandwidth. Packet drops are usually detected by obtaining acknowledgments of packets from receivers and detecting missing packets.

Protocols like TCP utilize acknowledgments for both reliability and congestion-feedback information. However, implementation of an acknowledgment scheme is unnecessary for many applications which do not require per-packet reliability and essentially desire only an estimate of the allowable transmission rate. Hence, our goal is to design a scheme that provides *application-independent* feedback and can be conveniently used for updating transmission rates.

The rest of this chapter is organized as follows. Section 3.1 elaborates on the criteria for a good feedback scheme. We present the details of our feedback scheme in Section 3.2. Our feedback scheme is implemented in the context of the Congestion

Manager, which we describe in Section 3.3. We discuss the implementation of the Congestion Manager in Section 3.4 and summarize in Section 3.5.

3.1 Design Criteria

This section lists and describes the requirements for a good congestion feedback scheme.

(R-1) Incremental deployment: As Figure 1-1 shows, a chief characteristic of the Internet is its immense scale. Surveys show that there are over 43 million hosts in the Internet, as of January 1999 [14]. Hence, there is a significant number of legacy systems and protocol implementations on the Internet. As a consequence, protocols must be designed and deployed in an evolutionary manner in order to have an impact on the Internet [11]. There are two facets to this issue: (a) developing a clear deployment path for new protocols from the current state of the world, and (b) ascertaining that the new protocol does not interact detrimentally with existing implementations. In our context, this implies that our congestion feedback scheme must interact seamlessly with unmodified network hosts, and must also be friendly to other congestion-controlled protocols (e.g. TCP) widely deployed on the Internet. Furthermore, we do not require or assume any router mechanisms such as ECN, although these are valuable as hints.

(R-2) Network load: Our problem domain deals with applications that do not require per-packet reliability, and hence the feedback scheme requires control information which is not directly useful to such applications. The scheme would be impractical if the traffic generated to exchange this control information between hosts is a significant proportion of all transmissions as this could potentially generate network congestion. Also, the rate adaptation scheme must react conservatively in the absence of feedback, since network congestion could have occured.

(R-3) Host load: The performance of applications which do not solicit feedback from receivers should not be adversely affected by processing related to feedback. At the sender, this implies that the state maintained per flow should be small so as to scale with the number of flows. Additionally, the sending of control packets normally incurs context switching overhead and possibly timer interrupts. The scheme should hence send as few control packets as possible to reduce this cost. At the receiver, the arrival of each data packet updates some state information for the flow. The arrival of each control packet also triggers an interrupt, some processing and a control packet transmission in response. Our scheme reduces the per-flow state as well as the processing in response to control packets at the receiver.

(R-4) Adaptation: In the absence of router support, feedback about congestion is normally available within a round-trip of its onset. Protocols like TCP adapt at an interval which is a function of the mean and variance of the round-trip time. The feedback of frequency solicited from receivers must therefore depend on the round-trip time (unlike protocols like RTCP). In particular, the frequency of feedback must at least be the granularity at which the congestion-control scheme adapts to congestion. A lower frequency could cause the congestion-control scheme to reduce its sending rate unnecessarily leading to inefficient utilization of network resources.

(R-5) Loss resilience: Control packets that carry feedback information can be lost in the network. The feedback protocol, in concert with the algorithm for congestion control, must be able to accommodate and detect the loss of these control packets in order to maintain network stability. Since congestion feedback is useful only if it is timely, it is not worthwhile to use a reliable transport protocol like TCP which ensures eventual arrival (often delayed) of control packets. Additionally, a protocol like TCP imposes additional state and processing overhead on end-hosts.

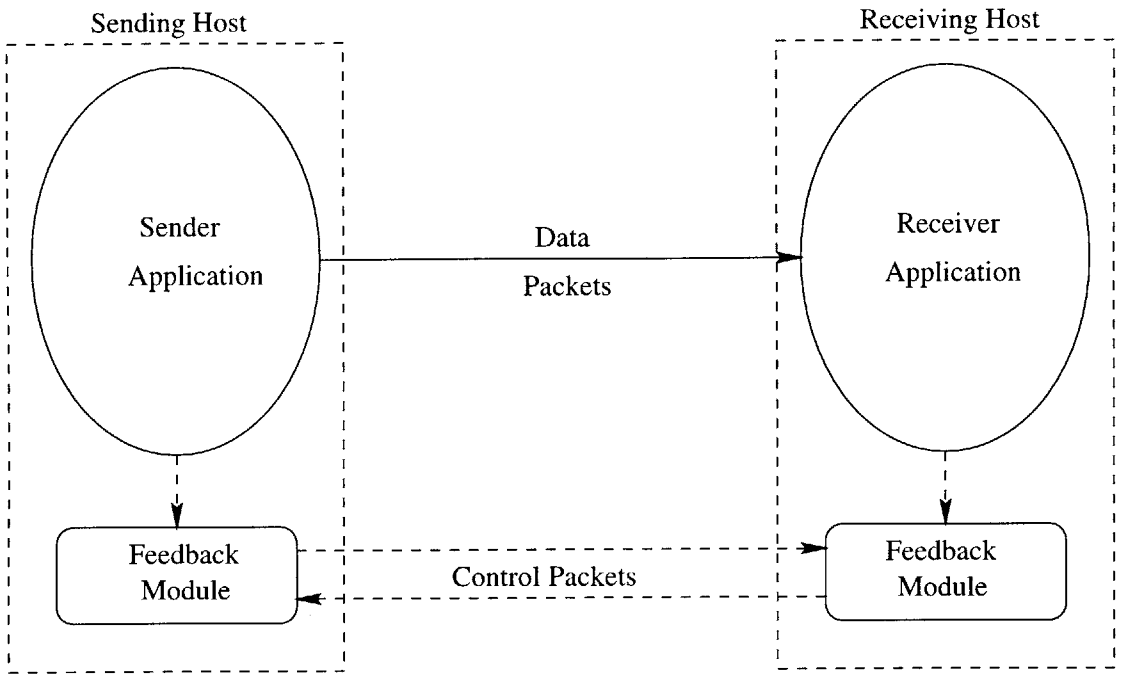

Figure 3-1: Schematic architecture of a generic feedback scheme.

3.2 Design of the Feedback Protocol

This section presents the details of our protocol for application independent feedback. Figure 3-1 describes our schematic architecture. Sender and receiver applications only exchange data packets; they inform the feedback modules about the transmission or arrival of data packets. The feedback modules use this information to exchange appropriate control packets.

One possible feedback scheme is for the receiver layer to acknowledge every k^{th} data packet with the total number of bytes received since the previous acknowledgment. However, this is undesirable since it requires the overhead of timers at the receiver feedback module to send an acknowledgment in case sufficient data packets are not received. Additionally, as described in requirement R-4, these timers will require round-trip time information which is generally not available at the receiver. Moreover, since the sender feedback module is not guaranteed to be on the data path, it does not have the ability to number each packet with a unique sequence number and this scheme cannot unambiguously detect losses.

S1. Sender Initialization

```
int nxmitted, probenum;
Queue PendingProbes;

nxmitted = 0;
probenum = 0;
PendingProbes.init();
```

S2. Sender action on a sending `nsent` bytes

```
nxmitted += nsent;
```

S3. Sending a probe to the receiver

```
probe = <PROBE, probenum>;
Send probe;
PendingProbes.insert({seq=probenum, time=now(), nsent=nxmitted});
nxmitted = 0;
probenum++;
Set timer for next probe;
```

S4. Sender action on receiving a response `<RESPONSE, lastprobe, thisprobe, count>`

```
// Delete state for probes for which responses
// will never again be received
while (PendingProbes.head().seq <= lastprobe) do
  PendingProbes.delete();
totsent = 0;
// At this point, the first entry in PendingProbes
// must have seq = lastprobe + 1
while (PendingProbes.head().seq <= thisprobe) do {
  totsent += PendingProbes.head().nsent;
  timesent = PendingProbes.head().time;
  PendingProbes.delete();
}
totrecd = count;
rtt     = now() - timesent;
Update congestion control algorithm with
totsent, totrecd, rtt;
```

Figure 3-2: Sender-side pseudocode for handling probes/responses. Each entry in the PendingProbes Queue has three elements: `seq`, `time` and `nsent`.

R1. Receiver Initialization

```
int lastprobe, narrived;

lastprobe = -1;
narrived  = 0;
```

R2. Receiver action on receiving `nrecd` bytes

```
narrived += nrecd;
```

R3. Receiver response to probe `<PROBE, thisprobe>`

```
response = <RESPONSE, lastprobe, thisprobe, narrived>;
response.send();
lastprobe = thisprobe;
narrived = 0;
```

Figure 3-3: Receiver-side pseudocode for handling probes/responses.

Instead, we choose to design a scheme where receivers are purely reactive, while senders send periodic probes to elicit feedback. The frequency of probing is a function of the round-trip time estimated for the destination. While this implies that the sending of each probe is triggered by a timer interrupt at the sender, we believe that this cost is acceptable since it is incurred for several packet transmissions across multiple flows to this destination. Receivers respond to arriving probe packets with appropriately constructed response packets. Each sender probe has a unique sequence number, and each response packet contains the numbers of the probes to which it replies as well as the number of bytes received in the relevant interval. The sender feedback module can correctly associate probes and responses even in the presence of network losses, and thereby determine (using its internal state) whether data packets have been dropped.

Figures 3-2 and 3-3 describe the behavior of the sender and receiver respectively (using C++-like pseudocode) upon receipt of probe and response packets. Note that the pseudocode uses a first-in first-out `Queue` datatype to store the state associated with each transmitted probe. The datatype supports the following operations:

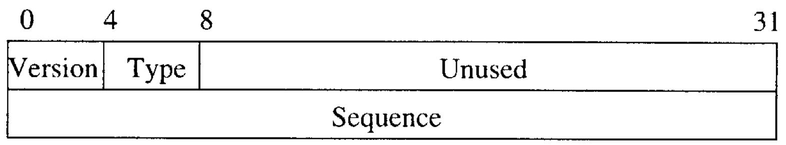

Figure 3-4: Format of the probe packet. `Type` is `PROBE (1)`. `Sequence` is the incrementing probe sequence number.

`init`: Initialize the state of the queue.

`head`: Returns the element in the front of the queue.

`insert`: Adds an element at the rear of the queue.

`delete`: Removes an element from the front of the queue.

Also, the operator `<...>` is used to indicate composition or decomposition of a message into its individual elements.

The module for sender feedback maintains two integer variables `nxmitted` and `probenum`, as well as a queue of state for unacknowledged probes. `nxmitted` tracks the number of bytes transmitted by the sender since the last probe was sent and `probenum` is the sequence number of the next probe packet to be transmitted. Whenever the transmitting application informs the sender feedback module that `nsent` bytes have been sent, `nxmitted` is incremented by the value of `nsent`. When the timer associated with the probe fires, the sender stamps the outgoing probe packet with sequence number `probenum` (Figure 3-4 shows the format of a probe packet). This sequence number is logically separate from the sequence number space of the different transports and applications using the CM. It then adds the time this probe was sent, as well as the value of `nxmitted` to its queue of outstanding probe messages. `probenum` is then incremented, `nxmitted` is set to 0 and a transmission timer for the next probe is set.

The receiver needs to maintain two integer variables `narrived` and `lastprobe` per transmitting host. `narrived` tracks the number of bytes received since the last probe and `lastprobe` is the sequence number of the last probe received. Whenever

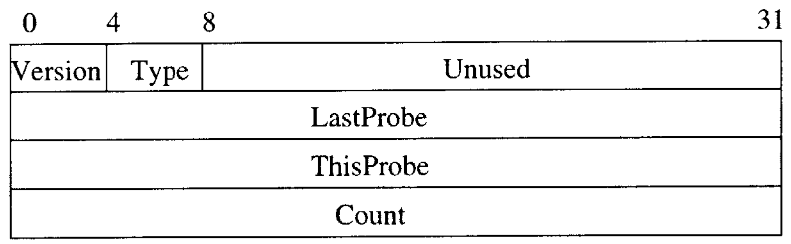

Figure 3-5: Format of the response packet. `Type` is `RESPONSE (2)`. `ThisProbe` is the sequence number of the probe triggering the response and `LastProbe` is the sequence number of the previous probe received. `Count` is the number of bytes received between `LastProbe` and `ThisProbe`.

the receiving application informs the feedback module of the arrival of `nrecd` bytes, the value of `narrived` is incremented by `nrecd`. Upon the arrival of a new probe with number `thisprobe`, a response packet (with the format shown in Figure 3-5) is constructed with entries corresponding to `lastprobe`, `thisprobe` and `narrived`. `lastprobe` is now set to `thisprobe` and `narrived` is set to 0 to maintain the invariant properties.

Upon receipt of a response packet, the sender can determine the number of bytes transmitted between `lastprobe` and `thisprobe`. From its queue of outstanding probes, the sender can also estimate the round-trip time as probes are expected to be acknowledged immediately by receivers. It can then call `cm_update` with the appropriate parameters to update congestion information. Additionally, the sender can delete the state associated with all probes with sequence number less than or equal to `thisprobe`. This is possible since the sender now knows that future response packets will have a value of `lastprobe` at least as large as the current value of `thisprobe`.

Note that the algorithm is resilient to losses of probes as well as responses since each response packet unambiguously defines the set of probes it responds to, and thus satisfies requirement R-5. The scheme also meets requirements R-2 and R-3 since it is lightweight in terms of processing at the sender and receiver, as well as in the number and size of control (probe and response) packets

The careful reader would have noted that the scheme is susceptible to the effects of reordering in the network. This is inevitable as individual data and control packets are not numbered and hence the scheme will interpret reordering of probe and data packets as a transient loss. Similarly, a packet loss between two probes can be masked by packet duplication in the network during the same epoch[1]. However, we do not believe that this is a serious concern as packet duplication is uncommon in the Internet today [23].

3.3 Congestion Manager

We now describe the design of the Congestion Manager (CM) [1]. The CM provides the framework for the implementation of our congestion feedback scheme described in Section 3.2. The motivation, high-level architecture and the adaptation algorithm of the CM are described in Section 3.3.1. We then detail the design of the CM API in Section 3.3.2.

3.3.1 Architecture

The CM is an end-to-end framework for managing network congestion. It is motivated by the desire to

- Efficiently multiplex concurrent flows between the same pairs of hosts such as web connections. Since these flows share the same path, they can share knowledge about the network state instead of competing for bandwidth.

- Enable applications which do not use TCP to adapt to congestion.

The CM unifies congestion management across all flows to a particular destination, independent of the particular application or transport instance.

The sender architecture of the protocol stack with the CM is shown in Figure 3-6. As the figure shows, the congestion manager integrates congestion and flow information across multiple applications and protocol instances. Currently, the CM aggregates all flows to a particular destination address as these are highly likely to

[1]If the receiving application is capable of detecting duplicate packets, then it can avoid notifying the receiver feedback module of duplicate packets to prevent this occurrence.

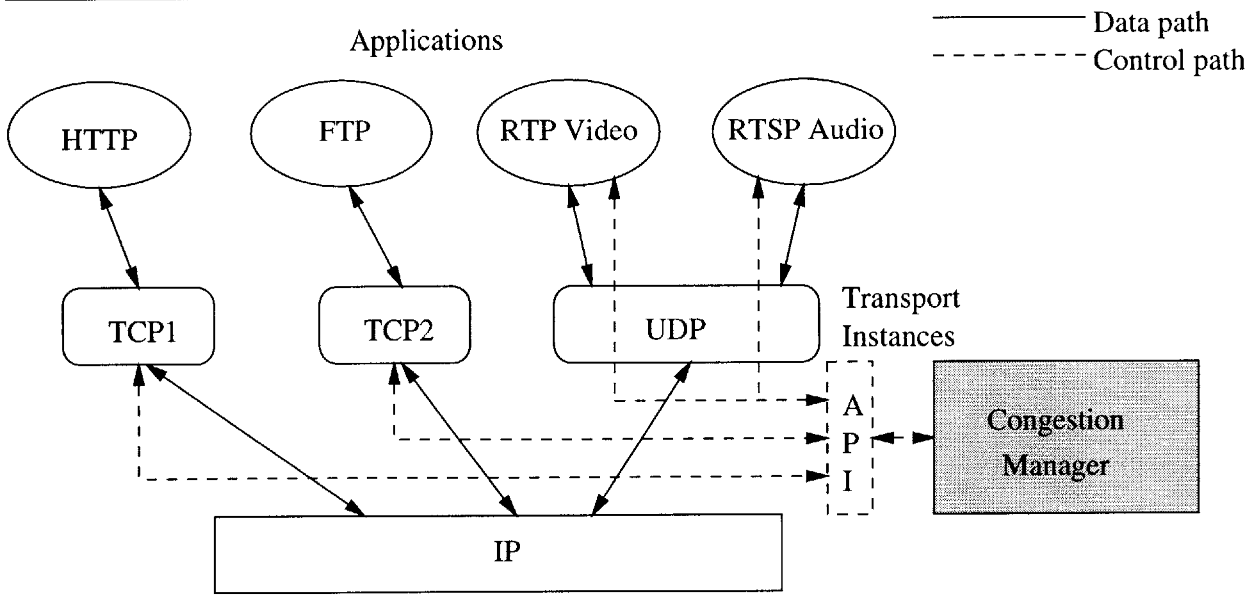

Figure 3-6: Sender Protocol Stack architecture with the Congestion Manager.

share the same path. In the future, it could aggregate information on a per-subnet or per-domain basis too.

The internal algorithm used by the CM to react to congestion is window-based with traffic shaping. The window growth and decay follow the AIMD algorithms of TCP shown in Figure 2-3. This is a pragmatic choice as the protocol is demonstrably TCP-friendly. A key difference is that the CM shapes outgoing traffic to prevent bursts of traffic. It achieves this by allowing transmission of the sender window gradually over its estimate of the smoothed round-trip time.

An important feature of the congestion management algorithm of the CM is *exponential aging* to deal with infrequent feedback about network state. The different alternatives for a sender in such a case are to:

- Stop transmission until feedback is received or the sender times out. While this is a conservative strategy and does not adversely affect other flows, it leads to long recovery times and can cause inefficient utilization of the network.

- Continue sending at the estimated rate until feedback is received. This is too aggressive as it can cause the network to remain in an unstable state for extended periods of time. In fact, this particular behavior has often been cited as a key reason against rate-based protocols.

- Decay the rate by a constant factor (half in this case) for every time interval in which no feedback is received. This allows the sender to continue transmitting data at a conservative rate while waiting for feedback. A key parameter of this method is the time interval between decays, a natural choice being the smoothed round-trip time. However, this choice is inappropriate since its value in fact increases during periods of congestion. Instead the minimum round-trip time observed to the particular destination host is chosen as the half-life for the rate decay. This is the method adopted in the CM.

Since the CM unifies congestion management across several flows to the same destination, an important consideration is the policy for apportioning bandwidth among flows. The CM currently apportions bandwidth to different flows according to preconfigured weights, and defaults to equal sharing of available bandwidth among all flows. However, this can be problematic in networks with differentiated or integrated services which can differentiate between flows based on port numbers, flow identifiers etc. The CM intends to accommodate this by incorporating some mechanism to infer this differentiation through observed loss rates and throughputs of different flows.

3.3.2 Application Programming Interface

Since the CM unifies congestion management functionality across multiple flows with potentially different characteristics, a key requirement is that it export a simple interface that is sufficiently general to accommodate these requirements. We explain the design considerations which motivate the API, and then describe the individual functions.

The guiding principles in the design of the API are to:

(a) **put the application in control.** While it is the responsibility of the CM to estimate congestion and available bandwidth in the network, the end-to-end argument [32] and the Application Level Framing (ALF) [7] guideline suggest that the adaptation to congestion should be performed at the application level. The CM API achieves this by not buffering any data and only telling applications when they can send. Thus applications make non-blocking requests for transmission which are granted using CM upcalls. This is in contrast to an API with a single `send()` call like that of Berkeley sockets [35].

(b) **accommodate diversity in traffic patterns.** The CM API should be usable effectively by several types of traffic, for example, TCP bulk transfers, Web traffic, layered audio or video sources, as well as flows that are capable of transmission at a continuum of rates.

(c) **accommodate diversity in application styles.** The CM API must suit different application styles, instead of forcing application writers to use a particular method. Most applications that transmit data are either *synchronous* or *asynchronous*. Data transmission in asynchronous applications is triggered by external events, such as file writes or frame captures. On the other hand, there is a class of synchronous applications where data transmission is triggered by firing of a timer. The CM exports interfaces which can be conveniently used by both these classes of applications.

(d) **learn from the application.** Since some applications obtain feedback from receivers, the API provides functions for the CM to leverage this feedback instead of independently rediscovering network conditions.

The function `cm_open(daddr, dport)` is called when a flow to the destination address `daddr` and port `dport` is initiated by an application. This returns the flow identifier `id` used in subsequent calls to the CM. When a flow is terminated, the application calls `cm_close(id)` allowing the CM to update its internal state associated with that flow. An application can also invoke `cm_query(id, *rate, *srtt)` to obtain the current estimate of the round-trip time as well as its share of the bandwidth.

The CM exports two types of callbacks for use by synchronous and asynchronous applications. Asynchronous applications make a `cm_request(id)` call whenever they

```
typedef int cmid_t;
```

Query

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

Control

```
cmid_t cm_open(addr daddr, int dport);
void   cm_request(cmid_t id);
void   cm_notify(cmid_t id, int nsent);
void   cm_update(cmid_t id, int nrecd, int nlost, 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(cmid_t id);
void   cmapp_update(cmid_t id, double rate, double srtt);
```

Figure 3-7: Data structures and functions for the sender-side CM API.

Control

```
cmid_t cmr_open(addr saddr, int sport);
void   cmr_notify(cmid_t id, int nrecd);
```

Figure 3-8: Data structures and functions for the receiver-side CM API.

desire to send data. After some time interval, the CM makes an application callback with `cmapp_send(id)`. This callback is a grant for the application to send upto a maximum transfer unit (MTU) [18] worth of bytes. Since the actual data buffer is never passed as an argument in these calls, the application can decide to send different data if it senses congestion or excess bandwidth in the network. On the other hand, synchronous applications essentially need to set the frequency of their timers as a function of the sustainable bandwidth of the network. Hence the CM exports an additional callback `cmapp_update(id, rate, srtt)` which is invoked whenever the CM's estimate of the bandwidth available to the flow with identifier `id` changes. Note that the `cmapp_update()` call is also beneficial for asynchronous applications which can use it to adaptively choose from multiple different data qualities.

Additionally, the CM sender exports a `cm_notify(id, nsent)` function. This is called by applications to inform the CM that `nsent` bytes were sent by flow `id`. The CM updates its estimates of actual data sent when this function is called.

For the sake of completeness, we also discuss the `cm_update()` function call which applications use to provide feedback to the CM about network conditions. Note that this thesis discusses applications that do not obtain feedback from receivers and hence never invoke `cm_update()`. The call `cm_update(id, nsent, nrecd, lossmode, rtt)` informs the CM that the estimated round-trip time is `rtt`, `nsent` bytes were sent by the flow `id`, of these `nrecd` were received. The value of `lossmode` can be `PERSISTENT` (e.g. a TCP timeout), `TRANSIENT` (e.g. a TCP triple-duplicate acknowledgement), or `ECN` (Explicit Congestion Notification). This function allows the CM to update its estimates of the congestion window and round-trip time.

On the receiver side, whenever a packet with source address `saddr` and port `sport` associated with a new flow is received, the application invokes `cmr_open(saddr, sport)` which returns a flow identifier `rid`. When a packet of size `nrecd` bytes is received on a flow with identifier `rid`, the receiver invokes `cmr_notify(rid, nrecd)`.

The CM API for the sender and receiver are summarized in Figures 3-7 and 3-8 respectively. It must be noted that while the entire API is discussed here for completeness, the rest of the thesis will focus on applications which do not incorporate any feedback. In particular, these kinds of applications will not use the `cm_update` call and hence the CM must incorporate mechanisms to obtain the feedback.

3.4 User-level Implementation

This section discusses our user-level implementation of the Congestion Manager. We choose to implement the CM at user-level since it achieves (a) rapid prototyping and (b) portability across flavors of Unix. Additionally, a kernel implementation normally requires privileged access to a machine before it can be installed and deployed.

The CM at the sender requires to schedule probe and data transmissions. This can be achieved by writing the application in polling or interrupt mode. Polling ensures that transmissions are scheduled at the exact time at the cost of loading the processor with constant and often unproductive loop checks. On the other hand, writing the application to be triggered by timer interrupts significantly reduces processor load, but can cause several scheduled events to miss their deadlines. Thus, while polling is detrimental to host performance, it can produce better network performance, whereas interrupts have the opposite effect. However, since a user process has no control over when it is scheduled by the operating system, a polling-based system can potentially miss several deadlines too. Hence, the CM is predominantly implemented using timer interrupts with judicious use of polling where delays are expected to be short.

We describe the internal architecture of the CM sender and receiver in Section 3.4.1, and an example application in Section 3.4.2.

3.4.1 Architecture of the CM

The CM sender and receiver are implemented as daemons, one of each per end-host. Applications link to a CM library which exports the CM API.

The CM library registers applications to receive callbacks for `cmapp_send` and `cmapp_update` using USR1 and USR2 interrupts respectively, which are raised by the CM sender daemon. The library also allows applications to invoke the API functions on the CM daemons in a manner similar to regular functions. When an API function

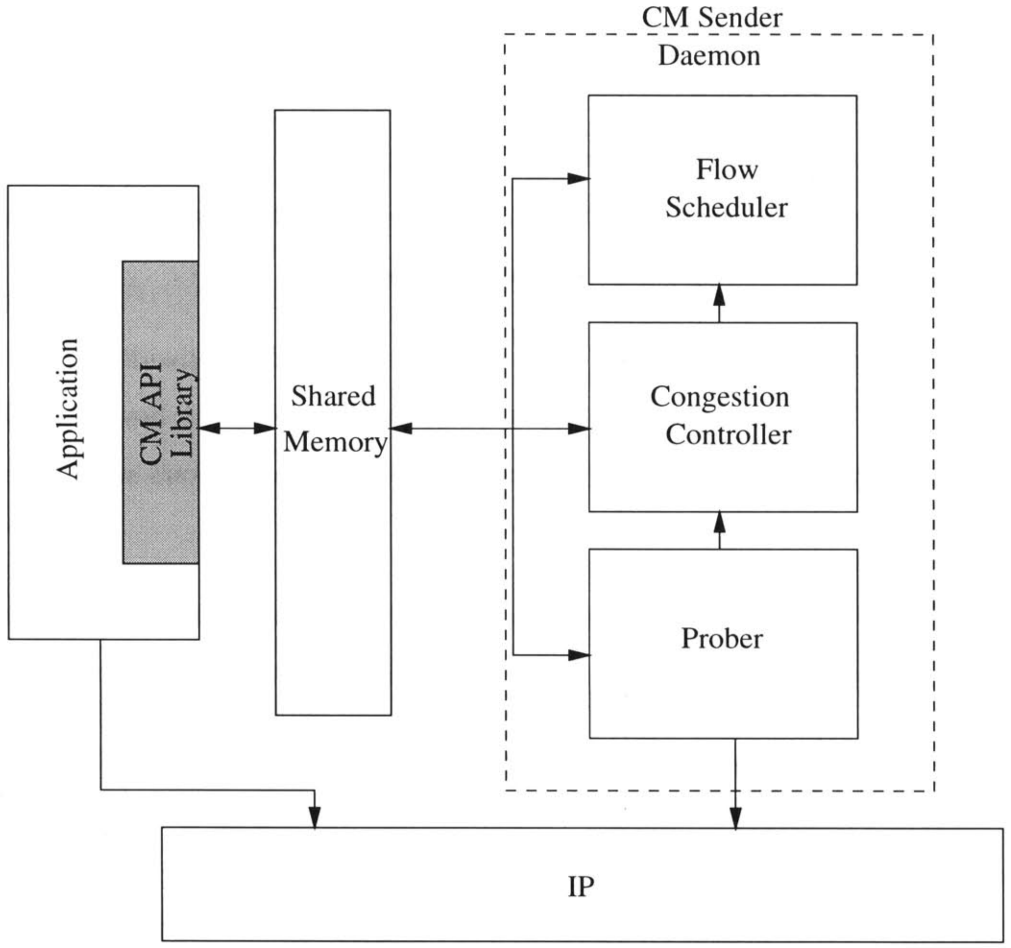

Figure 3-9: Architecture of applications and the CM daemon at the sender.

is invoked, the parameters are written by the library implementation in a region of shared memory and the CM daemon is woken up by a write on a well-known socket. The CM daemon then updates its internal state and writes return parameters back in shared memory. Meanwhile, the application blocks on a read on another well known socket till it is woken up by a socket write by the daemon. This event occurs when the daemon has executed the library function and written the return values in shared memory. The library implementation can now read these values and return them to the application.

The CM sender daemon is composed of three intercommunicating modules which together implement the CM functionality described in Sections 3.2 and 3.3. They are:

(a) Congestion Controller: This calculates the total sustainable rate between the end-hosts based on its estimate of congestion in the network. For applications without feedback, it obtains these estimates using the feedback protocol implemented in the *Feedback* module described below.

(b) Scheduler: This module uses the estimate of available bandwidth from the Congestion Controller and apportions it among the different flows. Application transmissions are scheduled by this module and applications are notified using callbacks.

(c) Feedback: This component schedules the transmission of probes and updates the Congestion Controller module upon receipt of responses from the receiver.

These modules communicate with each other through well-known interfaces as specified in [2]. The architecture of the sender-side functionality of the CM is depicted in Figure 3-9.

The CM receiver functionality is likewise implemented as a library that allows receivers to invoke the `cmr_open()` and `cmr_notify()` functions on the CM receiver daemon. The CM receiver daemon additionally implements the response functionality to provide feedback to senders whenever probes arrive.

3.4.2 An Example Application

In this section, we describe the sequence of steps executed when a sample application uses the CM API. The application we describe is a layered audio server. These servers have multiple encodings whose quality is usually proportional to their data rates, and the goal is to provide the best sustainable audio quality to clients. Thus, they need to estimate the available bandwidth in the network in order to use an appropriate encoding. We consider a case where the audio client does not provide feedback to the audio server.

Figure 3-10 presents pseudocode for a simple layered audio server. When the server first receives a request from a client, it invokes the `cm_open()` function allowing the CM sender daemon to set up the internal state associated with this flow. The `cm_open()` call also implicitly registers the application for `cmapp_update()` callbacks. It then uses the `cm_query()` function to obtain its share of the bandwidth estimated by the CM. This allows the server to choose an initial encoding which can be sustained by the network. Subsequently, the CM calls the `cmapp_update()` function whenever the share of bandwidth available to the application changes. This can happen either due to the increase and decrease algorithms built into the CM, or the addition or removal of flows (decided by the Scheduler module). The server can implement the callback function to choose the best possible source encoding with a data rate less than the CM estimate. The implementation shown always chooses the best possible encoding when `cmapp_update` is invoked. Another implementation might decide to impose a minimum time interval between rate changes to reduce variations in buffering requirements at the receiver. Thus, while the reaction to congestion is implemented in the CM, applications can adapt to it in a variety of ways.

The audio client notifies the CM receiver whenever it receives a packet from the server. Simultaneously, the Feedback module of the CM sender daemon sends probe packets to the CM receiver. It then passes the information from the elicited responses to update the Congestion Control module which can recalculate its rate and round-trip time estimates.

3.5 Summary

This chapter described the design of a congestion feedback protocol and its implementation in the context of the CM. This protocol uses probes by an active sender and responses from a reactive receiver, and is adaptive, lightweight and resilient to network losses. We also discussed the chief design components and a user-level implementation of the Congestion Manager.

Global declarations

```
struct {
  int rate;               // Rate in bits per second
  double inter_pkt_time;  // Time between packets in ms
} Encoding;

Encoding codes[NUM_CODES];// The list of server encodings in ascending
                          // order of rates
Encoding current_code = codes[0];
cmid_t id;
```

Server code

```
id = cm_open(daddr, dport); // Initialize CM state

// Seed the server with an appropriate initial encoding.
// This assumes that codes[0].rate is always
// lesser than the rate returned by cm_query
cm_query(&rate, &srtt);
for (int i = 0; i < NUM_CODES; i++)
  if (codes[i].rate <= rate) current_code = codes[i];
  else break;

while (TRUE) {      // Main loop
  double next_time = current_time() + current_code.inter_pkt_time;
  wait(next_time);  // Block till next_time
}
```

Timer Handler

```
// Called when wait is unblocked at next_time
void timeout() {
  int pktsize = (current_code.rate * current_code.inter_pkt_time)/8000;
  Send packet of size pktsize bytes from encoding current_code;
  cm_notify(id, pktsize);
}
```

cmapp_update()

```
void cmapp_update(cmid_t id, double rate, double srtt) {
  for (int i = 0; i < NUM_CODES; i++)
    if (codes[i].rate <= rate) current_code = codes[i];
    else return;
}
```

Figure 3-10: Pseudocode for a layered adaptive audio server. Details of locking critical sections for correctness have been elided in the interest of simplicity.

Chapter 4

Performance Evaluation

Chapter 3 described the design and implementation of our feedback scheme. In this chapter, we evaluate the behavior of our feedback scheme through simulation as well as performance measurements in real networks. Section 4.1 describes the simulation environment and the results of our simulations. The performance of our UNIX implementation is evaluated in Section 4.2.

4.1 Simulation

Simulation is an excellent aid to research in networking, and we have employed it to understand our feedback scheme. The chief advantages of simulation which are not achievable in experiments on the Internet are:

Repeatability: The conditions of links and routers on the Internet changes constantly, and hence experimental conditions can never be reproduced exactly when testing an implementation. However, a simulator allows all parameters to be accurately controlled so that the effect of changes in individual variables can be studied.

Fine-grained control: Simulation allows us to vary the parameters of the network such as topology, bottleneck bandwidth and latencies to understand the impact of their variation on our scheme. This is not achievable with the Internet.

Section 4.1.1 describes our choice of simulator and our enhancements to it. We then elaborate on our experiments in Section 4.1.2.

4.1.1 Simulation Environment

We use the UCB/LBNL/VINT network simulator `ns` (Version 2) [21] for our experiments. `ns` is a discrete event-driven simulator initiated as a version of REAL [16]. `ns` is primarily written in C++ [37] with MIT's Object Tcl (OTcl) [38] used as a command and configuration language. It has an object-oriented architecture which allows modular addition of new components, and augmentation of functionality of existing components using inheritance. Simulations are configured and executed in OTcl which is an object-oriented extension of the Tool Command Language (Tcl) [22]. This is achieved easily because of the coupling between C++ and OTcl objects using split objects. Instance variables in the two languages can be bound, thus ensuring that they expose the same state in both realms.

The simulator `ns` provides abstractions for network links, queues and nodes. It also has implementations of several popular network protocols and applications (e.g. TCP and its variants, telnet, ftp), router algorithms (e.g. Drop-tail, RED) and traffic models (e.g. Exponential, Pareto, Constant Bit Rate, Web). New protocols and applications are obtained by inheriting from the `Agent` and `Application` classes respectively, and possibly reimplementing the `send` and `recv` methods. `Agent` and `Application` are split objects to allow their instantiation and access in OTcl.

We have implemented the CM in `ns` with a round-robin scheduler which apportions bandwidth equally among all flows to a particular destination. We have developed `Prober` and `Responder` classes inherited from the class `Agent`. Objects belonging to these classes are dynamically instantiated and connected whenever a flow to a new destination address was opened. We have also implemented a congestion-controlled UDP class (UDPCC) class `UdpCCApp` as a split object inherited from `Application`. The UDPCC application registers with the CM for `cm_appupdate` callbacks and transmits at the maximum rate allowed by the CM for this flow. The application sends packets of a fixed size, and hence the inter-packet time varies when the rate is changed.

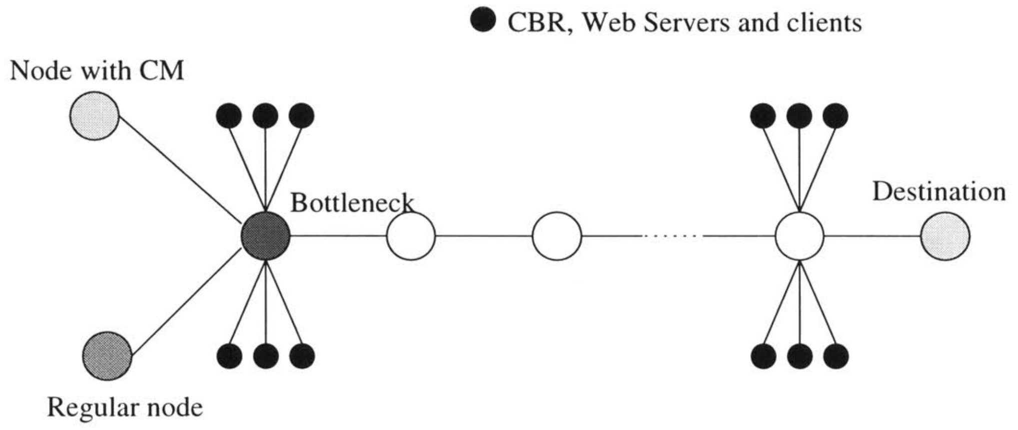

Figure 4-1: Network topology used for simulation experiments.

As the name implies, the flow is unreliable since acknowledgments are not solicited and congestion-controlled because the sending rate is determined by the CM.

4.1.2 Simulation Results

Figure 4-1 depicts the network topology used for our simulations. We perform our experiments in the presence of significant constant bit-rate (CBR) and web cross traffic. All reported values are the average of 10 iterations.

We now compare the performance of the `UdpCCApp` class with that of the Newreno variant of TCP [13]. Figures 4-2 and 4-3 depict sequence traces of regular TCP and a UDPCC application for bottlenecks of 200 kbps and 400 kbps respectively[1] (Feedback is solicited using 4 probes per round trip time). These traces show that the UDPCC application using the CM closely emulates the performance of regular TCP. Figure 4-4 also shows the relative performance of TCP and UDPCC applications for a range of bottleneck bandwidths. The results show that UDPCC and TCP show comparable throughputs over a wide range of bandwidths and that the feedback scheme is quite robust in the face of cross-traffic. Additionally, packet traces show that both probe and response packets are dropped during congestion epochs and the congestion-control scheme reacts appropriately in the presence of infrequent feedback.

[1]bps denotes bits per second and Bps denotes Bytes per second

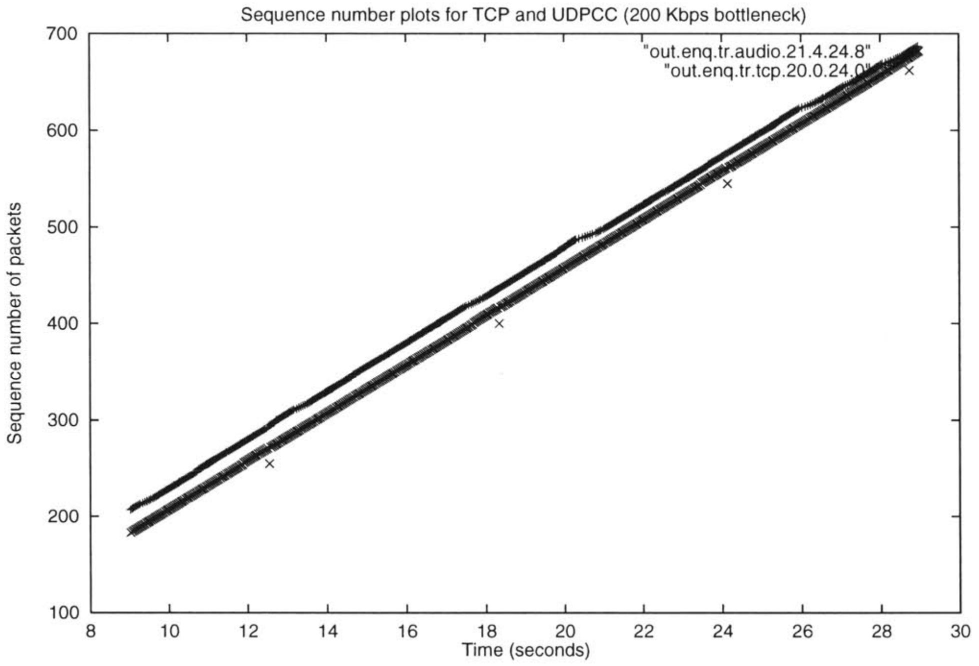

Figure 4-2: Sequence traces of TCP and UDPCC for a bottleneck bandwidth of 200 Kbps. The lower line shows the TCP trace.

We also compare the performance of a pair of UDPCC streams with a competing TCP stream in the presence of competing cross-traffic. The results for the topology of Figure 4-1 with bottleneck bandwidths of 200 kbps and 400 kbps are shown in Figures 4-5 and 4-6 respectively. Since the CM combines state across all flows to a particular destination, the two UDPCC flows behave like a single flow, and are expected to consume approximately as much bandwidth as the single TCP connection from the other node. Additionally, since the CM round-robin scheduler apportions the estimated bandwidth equally between the two UDPCC flows, they should deliver approximately the same throughput. The experimental data shown in Figure 4-7 shows this over a range of bottleneck bandwidths.

4.2 Implementation Evaluation

While simulation is an excellent aid, implementation is essential especially in the context of Internet research for the following reasons:

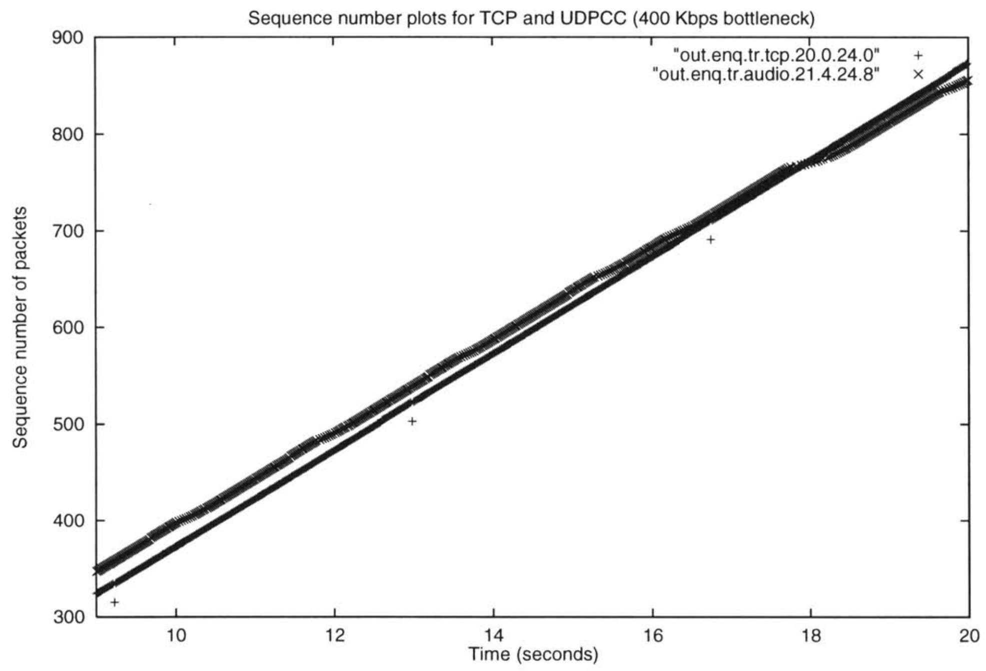

Figure 4-3: Sequence traces of TCP and UDPCC for a bottleneck bandwidth of 400 Kbps. The lower line shows the TCP trace.

Insufficient knowledge: The topology and link properties on the Internet constantly change with time and span a very large range. There is also a variety of protocol implementations with significantly different features and bugs. An absence of a good traffic model is another reason why it is difficult to simulate the Internet. These issues are elaborated in greater detail in [24].

System effects: While a simulator isolates the implementation of our scheme from operating system vagaries, in reality, effects like the timer granularity and process scheduling by the operating system have an important influence on the behavior of our scheme. For example, most operating systems have a clock granularity of 10 ms. This implies that a timer-driven sender transmitting 1500 byte packets cannot transmit at greater than 1.5 MBps. Additionally, if the operating system does not schedule the sending application at the appropriate time, future transmissions are delayed.

Bottleneck Bandwidth (kbps)	UDPCC Throughput (kbps)	TCP Throughput (kbps)
200	67.6	72.8
400	173.2	148.4
600	207.2	199.2
800	284.8	329.6
1000	312.8	358.4
1200	454.4	450.0
1400	468.8	527.2

Figure 4-4: UDPCC and TCP throughput as a function of bottleneck bandwidth.

We have developed interoperating user-level implementations of the CM for BSD/OS 3.0 and Linux 2.0.35, and performed wide-area experiments over the Internet backbone using hosts in Stanford University and University of California at Berkeley. As described earlier, a key performance issue we faced in the user-level implementation was that we did not have control over the process scheduling or timer granularity of the operating system. While the process scheduling issue could be resolved by appropriately setting the priority of the sending process, this requires privileged access to the system and is hence not a universal solution. Instead, we decided to compensate for missed deadlines (due to late timer firing or scheduling) by sending packets in small bursts whenever the sender actually could transmit data. While this is not completely desirable, we rationalized this behavior by verifying that TCP too tends to send short bursts of packets when growing its window.

Figures 4-8 and 4-9 show the sequence traces for competing TCP and UDPCC flows between sources at Stanford and Berkeley, and a destination at MIT. Both the flows show approximately the same throughput. However, the UDPCC trace is not as smooth as the TCP trace in either case. We believe that this happens since UDPCC has much more infrequent feedback (2 probes per round-trip) than TCP (an acknowledgment every alternate packet). Over 10 iterations, the average throughputs of UDPCC and TCP to Stanford were 2.87 Mbps and 2.93 Mbps respectively. The UDPCC and TCP throughputs to Berkeley were 0.51 Mbps and 0.57 Mbps respec-

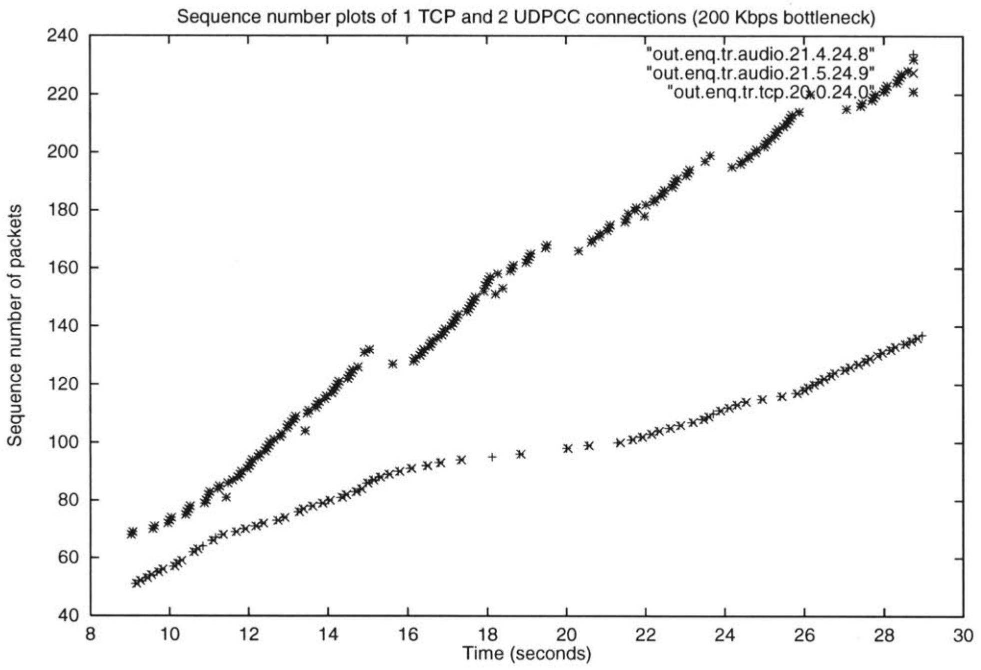

Figure 4-5: Sequence traces of TCP and 2 UDPCC flows for a bottleneck bandwidth of 200 Kbps. The upper line shows the TCP trace. The traces for the two UDPCC flows are almost identical.

tively.

We also performed wide-area experiments with two competing UDPCC flows and the results are shown in Figures 4-10 and 4-11. These experiments were performed at different times than the previous set of experiments. The throughput from Berkeley averaged over 10 iterations was 0.181 Kbps and 0.183 Kbps respectively, while that from Stanford averaged 0.635 Kbps and 0.634 Kbps respectively. We see that the two sending UDPCC applications achieve almost the same throughput in both cases. This is because the round-robin scheduler of the CM apportions the estimated bandwidth equally among all the participating flows.

4.3 Conclusion

In this chapter, we presented the results of simulation and implementation of our feedback scheme. We evaluated the efficiency of our scheme by comparing its throughput

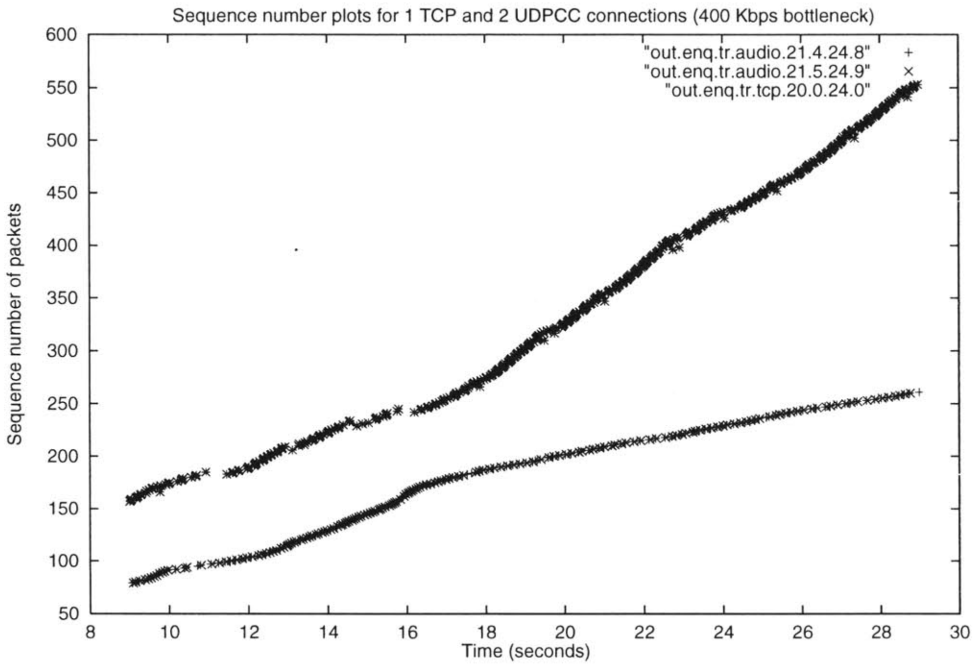

Figure 4-6: Sequence traces of TCP and 2 UDPCC flows for a bottleneck bandwidth of 400 Kbps. The upper line shows the TCP trace. The traces for the two UDPCC flows are indistinguishable.

with competing TCP flows, and its fairness by comparing the throughput of competing UDPCC flows. The results demonstrate that the feedback scheme allows applications to compete fairly and efficiently with existing transports on the Internet.

Bottleneck (kbps)	UDPCC1 Throughput (kbps)	UDPCC2 Throughput (kbps)	TCP Throughput (kbps)
200	30	32	66
400	71	76	146
600	106.4	101.2	221.2
800	148.4	143.3	322.4
1000	182.4	174.4	400.4
1200	243.6	235.2	453.2
1400	262.4	258.0	523.2

Figure 4-7: Throughput of TCP and 2 UDPCC flows as a function of bottleneck bandwidth.

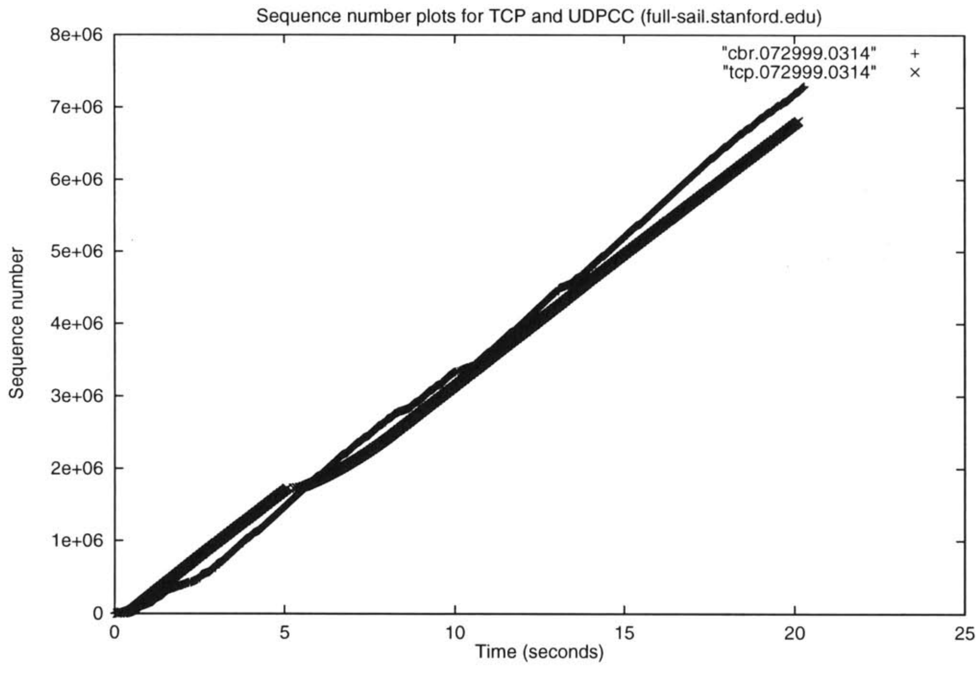

Figure 4-8: Sequence traces of TCP and UDPCC for a sender on full-sail.stanford.edu. The slightly longer line shows the TCP trace.

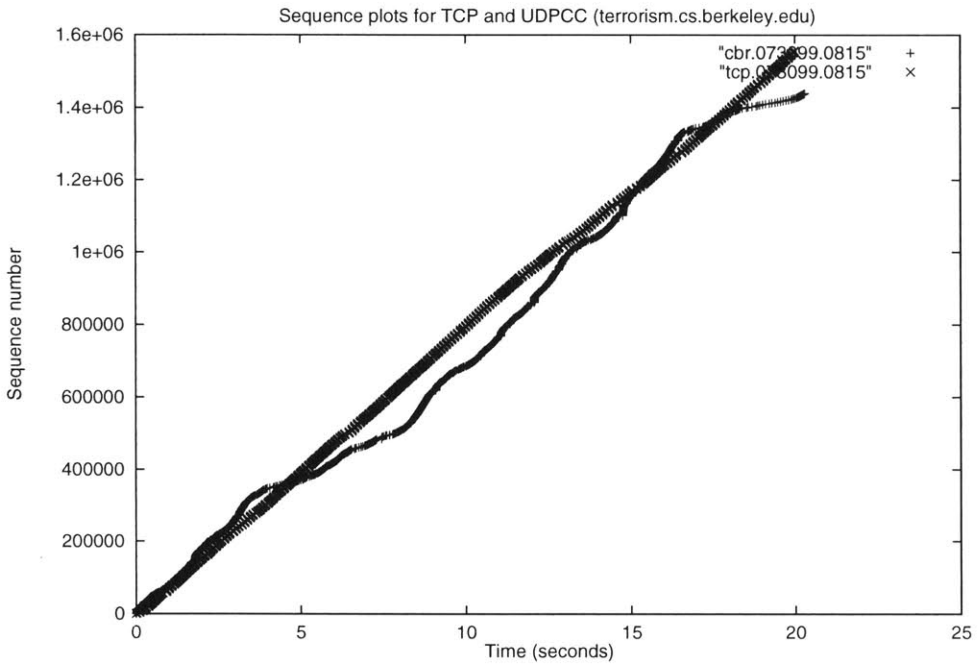

Figure 4-9: Sequence traces of TCP and UDPCC for a sender on terrorism.cs.berkeley.edu. The relatively straight line shows the TCP trace.

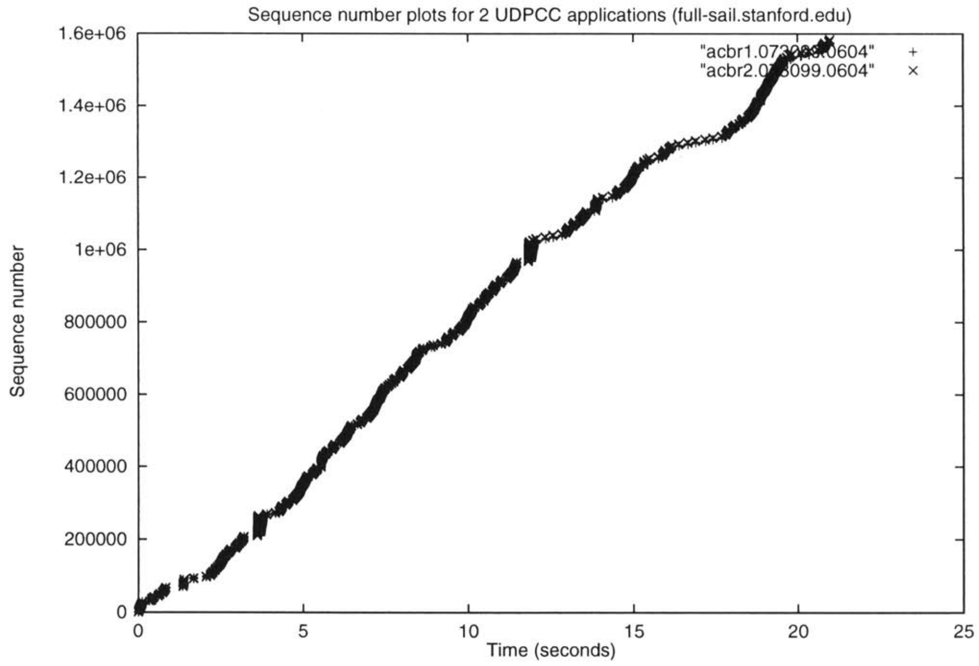

Figure 4-10: Sequence traces of competing UDPCC flows for a sender on full-sail.stanford.edu. The traces are virtually identical.

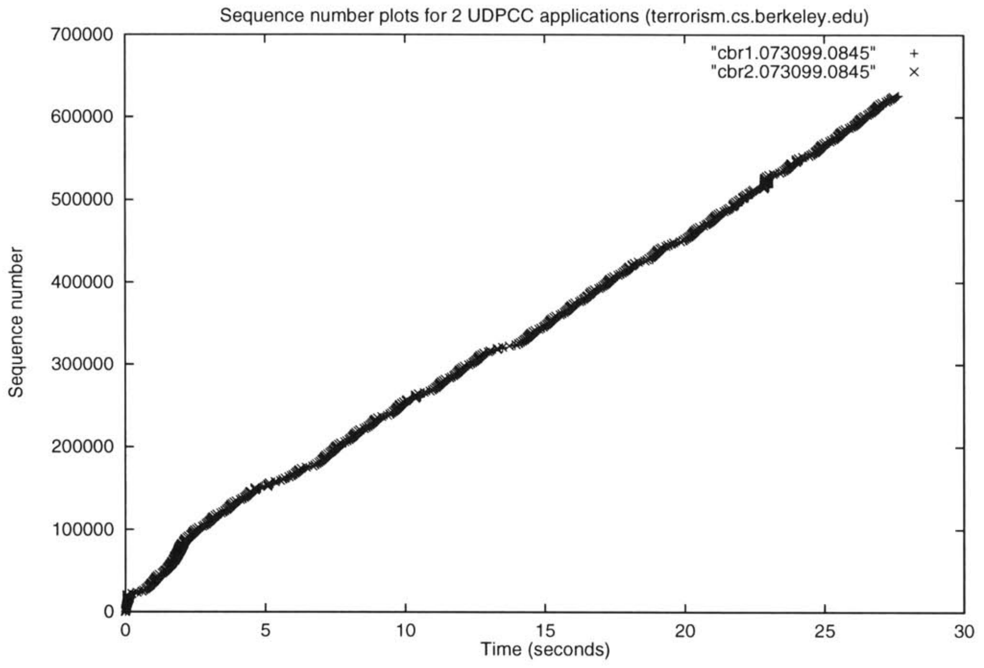

Figure 4-11: Sequence traces of competing UDPCC flows for a sender on terrorism.cs.berkeley.edu. The traces are indistinguishable.

Chapter 5

Conclusions and Future Work

We conclude our thesis with a summary of our contributions and directions for future work.

5.1 Summary

This thesis solves the problem of designing a congestion feedback scheme for applications that do not require reliable packet transmissions. A feedback scheme that is independent of individual applications frees programmers from reimplementing such a framework for every application, and allows them to focus on encoding application data to adapt to network congestion.

We proposed an end-to-end scheme for congestion feedback based on periodic probes from a sender module to solicit congestion information responses from receivers. Applications only need to notify the sender or receiver module when data is either transmitted or received. The scheme is designed to operate with relatively low overhead on the network and end-hosts, and is robust in the face of losses of both probe and response packets. It also adapts to the round-trip time of the flow in order to mimic the behavior of TCP.

We implemented our feedback module in the context of an end-to-end congestion management infrastructure, the Congestion Manager. The Congestion Manager provides unified congestion control across multiple applications and transports, and exports an API which allows applications to adapt to congestion without having to monitor and understand individual congestion events. We evaluated the performance of our feedback scheme through simulations in the VINT **ns** simulator, as well as wide-area Internet experiments.

The chief contributions of our work are:

An application-independent congestion feedback scheme for UDP flows. We draw on the principles of protocols like TCP, RAP and RTCP to design a feedback protocol that is lightweight, general and resistant to network losses.

User-level implementation of a Congestion Manager. Applications link to a library which implements the CM API and allows them to implement mechanisms for adaptation to congestion when it is detected by the CM.

Evaluation of the feedback scheme. We simulated the performance of our scheme across a variety of bottleneck bandwidths and in the presence of cross traffic. We also performed transfers across the Internet backbone using our user-level implementation. Our scheme shows fair and efficient performance against competing TCP connections. It also allows fair sharing of the estimated bandwidth amongst multiple UDP flows using the CM.

5.2 Future Work

We now outline a few directions for future work based on the work described in this thesis.

A kernel implementation of the Congestion Manager. In Chapter 4, we outlined the problems associated with a user-space implementation, despite its portability and ease of prototyping. The issues of coarse-grained timers and inappropriate scheduling can be addressed effectively by a kernel implementation. This should, in turn, enhance application performance.

Feedback frequency. In this thesis, we used sender probes at a frequency of 2 or 4 probes per round trip. It would be interesting to understand the impact of this frequency on end-to-end performance. Additionally, it might be possible to adapt this frequency dynamically to network conditions.

Receiver modifications. Our current feedback protocol requires some modifications to receivers in order to function correctly. A scheme that requires even fewer or no modifications to receivers might have greater chances of adoption and deployment.

Bibliography

[1] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. ACM SIGCOMM '99*, September 1999. To appear.

[2] H. Balakrishnan and S. Seshan. The Congestion Manager. Internet Draft, IETF, June 1999. Expires Dec 1999.

[3] T. Berners-Lee et al. The World Wide Web. *Communications of the ACM*, 37(8):76–82, Aug 1994.

[4] J.C Bolot, T. Turletti, and I. Wakeman. Scalable Feedback for Multicast Video Distribution in the Internet. In *Proc. ACM SIGCOMM*, London, England, Aug 1994.

[5] D-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[6] D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms. In *Proc. ACM SIGCOMM*, August 1992.

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

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulations of a Fair-Queueing Algorithm. *Internetworking: Research and Experience*, V(17):3–26, 1990.

[9] K. Fall and S. Floyd. Promoting the use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7, August 1999.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, Jan 1997. RFC-2068.

[11] S. Floyd. Internet Research: Comments on Formulating the Problem. ftp://ftp.ee.lbl.gov/papers/assumptions.ps, 1998. Unpublished manuscript.

[12] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

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

[14] Internet Software Consortium. http://www.isc.org/dsview.cgi?domainsurvey/report.html, 1999.

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

[16] S. Keshav. REAL: A Network Simulator. Technical Report 88/472, Computer Science Division, Univ. of California at Berkeley, 1988.

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

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

[19] Intelligent Streaming. http://www.microsoft.com/windows/windowsmedia/features/intellistream/default.asp, 1998.

[20] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. Internet Draft, IETF, November 1997.

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

[22] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, Reading, MA, 1994.

[23] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. ACM SIGCOMM '97*, September 1997.

[24] V. Paxson and S. Floyd. Why We Don't Know How to Simulate the Internet. In *Proc. Winter Simulation Conf.*, 1997.

[25] J. B. Postel. *Transmission Control Protocol.* Information Sciences Institute, Marina del Rey, CA, September 1981. RFC-793.

[26] J. B. Postel. *Simple Mail Transfer Protocol.* Information Sciences Institute, Marina del Rey, CA, August 1982. RFC-821.

[27] J. B. Postel and J. Reynolds. *File Transfer Protocol (FTP).* Information Sciences Institute, Marina del Rey, CA, Oct 1985. RFC-821.

[28] K.K. Ramakrishnan and S. Floyd. A Proposal to Add Explicit Congestion Notification (ECN) to IPv6 and to TCP. Internet Draft draft-kksjf-ecn-00.txt, November 1997. Work in progress.

[29] SureStream[tm] - Delivering Superior Quality and Reliability. http:// www.real.com/devzone/library/whitepapers/surestrm.html, 1998.

[30] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. IEEE INFOCOM '99*, 1999.

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

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

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

[34] S. Shenker. Fundamental Design Issues for the Future Internet. *IEEE Journal on Selected Areas in Communications*, 13(7):1176–1188, Sep 1995.

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

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

[37] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1997.

[38] D. Wetherall and C. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Proc. Tcl/Tk Workshop*, July 1995.

5230- 8