An End-System Architecture for Unified Congestion Management

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

Srinivasan Seshan
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
srini@watson.ibm.com

Abstract

In this position paper we motivate and describe the Congestion Manager (CM), a novel end-system architecture, which enables application adaptation to network congestion. The CM maintains congestion and path related information and allows flows to learn from each other and share information. It also exports an API to enable applications to learn about network status and regulate data transmission. As a result, applications are freed from having to detect network congestion and probe for spare bandwidth. We describe how TCP can be implemented using the CM and conclude that the CM provides an excellent framework for building adaptive Internet applications.

1. Motivation

The success of the Internet to date has been in large part due to the sound principles of additive-increase/ multiplicative-decrease congestion management [4] on which its dominant transport protocol, TCP [12, 23], is based. Because most traffic in the Internet has been dominated by long-running TCP flows, the network has shown relatively stable behavior and has not undergone large-scale collapse in the past decade.

However, Internet traffic patterns are changing rapidly and will be very different in the future. First, Web workloads stress TCP congestion control heavily and in unforeseen ways. Typical Web transfers are characterized by multiple concurrent, short TCP connections. These short Web transfers do not give TCP enough time or information to adapt to the state of the network, while concurrent connections to the same client compete rather than cooperate with each other for scarce resources. Second, there are commercial products being developed today that “accelerate” Web downloads, usually by turning off or changing TCP’s congestion control in unknown and potentially dangerous ways. Third, and perhaps most importantly, increasingly popular real-time streaming applications run over UDP, and in most cases today do not employ any adaptive congestion control.

These trends, coupled with the unknown nature of future popular applications, threaten the long-term stability of the Internet. They make it likely that large portions of the network might suffer congestion-triggered collapse due to unresponsive services. To some, this might sound overly pessimistic, but even the optimists amongst us will grant that to get the best performance applications should *adapt* to congestion. Unfortunately, achieving this for applications and protocols today is easier said than done, for there is no system support to effect this in current operating systems. Our work attempts to fill this void.

Our work overcomes these problems by developing a novel framework for managing network congestion from an end-to-end perspective. Unlike most past work on bandwidth management that focuses on mechanisms in the network to provide QoS to flows or reduce adverse interactions between competing flows [7, 18, 8, 5, 26], we focus on developing an architecture at the end-hosts to enable applications to adapt easily to network congestion. The resulting framework is independent of specific applications and specific transport protocol instances, but provides the ability for different flows to perform *shared state learning*. Here flows learn from each other and share information about the state of congestion along common network paths.

Increasingly, the trend on the Internet is for unicast data servers to transmit a wide variety of data, ranging from best-effort (unreliable) real-time streaming content to reliable Web pages and applets. Our design of the congestion management architecture is motivated by the observation that in the Internet of the future, many logically different streams using different transport protocols will share the path between server and client. They all have to incorporate control protocols that dynamically probe for spare bandwidth and adapt to congestion for the Internet to be stable. Furthermore, they are likely to have different reliability requirements, which motivates us to separate the functions of loss recovery and congestion management that are coupled in protocols like TCP.

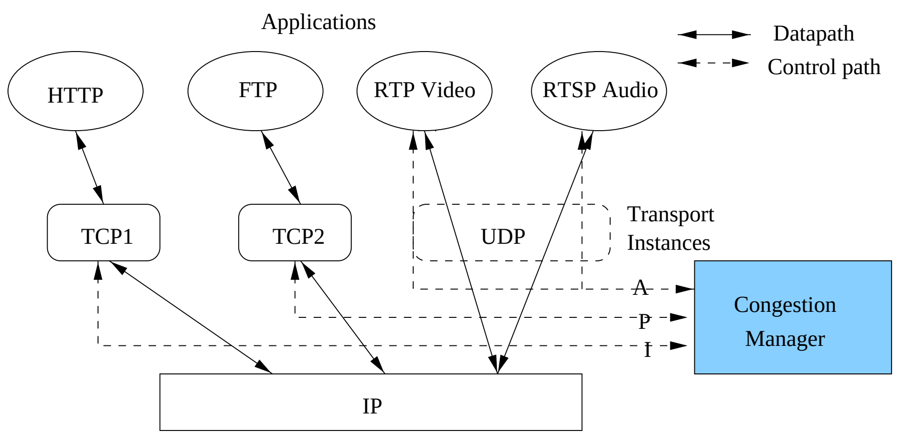

Figure 1. New sender architecture with unified Congestion Manager.

At the core of our architecture is the *Congestion Manager (CM)*, which maintains network statistics and orchestrates data transmissions governed by robust control principles. Rather than have each stream act in isolation and thereby also adversely interact with the others, the CM maintains host and domain-specific path information. Path properties are shared between different streams because applications and transport instances perform transmissions only with the CM's consent. The CM reacts to congestion, carefully probes for spare bandwidth, prioritizes data transmissions, and apportions available capacity between different active flows based on hints from receivers. For example, triggered by hints from the user or client application, the sender could decide to use one-half of the available bandwidth for a video stream, and partition the remaining equally between an audio stream and graphic-laden streams in a collaborative application.

The other important class of functions performed by the CM is motivated by the desire to enable application adaptation to congestion and variable bandwidth. The CM exports a simple API to allow applications to adapt to congestion, while accommodating the heterogeneous nature of streams. This API includes functions to query path status, notify the CM upon data transmission, and update path variables upon congestion or successful transmission. The CM is not on the data path of transmissions, but only on a control path where its API is used by applications. The design of the CM API, articulated in Section 3, is motivated strongly by the end-to-end argument [21], the principle of Application-Level Framing (ALF) [6], and the need for simplicity. The CM API permits the application to have the final say in deciding what to transmit, especially when available bandwidth is smaller than what the application desires. It does so by not buffering any data on its own, relying instead on a request/callback/notification mechanism (explained in Section 3).

The resulting network architecture from the viewpoint of a data sender is shown in Figure 1. By providing congestion management as a service in a transport and application-independent manner, it permits information about network dynamics to be efficiently shared between logically different streams without adverse interactions between them. It frees different transport and application protocols from having to (re-)implement congestion control and management from scratch, and it discourages applications from using an inappropriate transport protocol (e.g., TCP for high-quality audio) simply because the transport implements a congestion control scheme. We claim that the CM provides the right support and a simple API over which adaptive Internet applications can be developed.

Section 2 discusses and critiques existing approaches to these problems. In Sections 3 and 4, we present the design of our CM, discussing its adaptation API and internal mechanisms respectively. We present an example application using the CM in Section 5. We then conclude in Section 6 where we argue that the CM architecture is worth pursuing and discuss our future plans.

2. State-of-the-World Today

Most Web sessions today use multiple concurrent TCP connections. Each connection wastefully performs slow start irrespective of whether other connections are currently active to the same client. Furthermore, upon experiencing congestion along the path to a client, only a subset of the connections (the ones that experience losses) reduce their window. The resulting multiplicative decrease factor for the ensemble of connections is often larger than 0.5, the value used by an individual TCP connection[1]. This is unfair relative to other clients that use fewer connections, and worse, will lead to instability in a network where most clients operate in this fashion. There has also been some recent work in

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

developing application-specific congestion control schemes for real-time multimedia streams. We discuss two classes of solutions to the unicast congestion control problem—*application-level solutions* and *transport-level solutions*.

2.1. Application-level Solutions

Application-level solutions for Web transport multiplex several logically distinct streams onto a single transport (TCP) connection to overcome the adverse effects of independent competing TCP transfers. Examples of this include Persistent-connection HTTP (P-HTTP, part of HTTP/1.1), which is application-specific, and the Session Control Protocol (SCP) [22] and the MUX protocol [9], which are not tied to HTTP.

There are several drawbacks with this class of solutions.

- *Architectural problems:* These solutions are application-specific and attempt to *avoid* the poor congestion management support provided by protocol stacks today. However, congestion is a property of the network path and the right point in the system to manage it is inside the protocol stack, not at the application. If the right support is provided by the system, the need for such solutions can be eliminated.
- *Application-specificity:* These solutions require each class of applications (Web, real-time streams, file transfers, etc.) to reimplement much of the same machinery, or else force them to use protocols like TCP that are not well-suited to the task at hand.
- *Undesirable coupling:* These solutions typically multiplex logically distinct streams onto a single byte-stream abstraction. If packets belonging to one of the streams is lost, another stream could stall even if none of its packets are lost. This is because of the in-order delivery provided by TCP, which forces a linear order over all the transferred bytes even when only a partial order is desired. This violates the ALF principle [6], which states that independent Application Data Units (ADUs) should be independently processible by receivers independent of the order in which they were received.

The WebTP proposal [10] aims to develop a receiver-oriented approach to handle concurrent Web transactions. This includes maintaining congestion parameters at the receivers, which makes it easy to incorporate our equivalent receiver hints for bandwidth partitioning between flows. On the other hand, because the eventual transmissions are performed by the sender, we believe that the CM design is sound and also achieves some of WebTP's benefits. Like the CM, WebTP has also been heavily motivated by ALF as a protocol design principle.

There has been some recent work in developing congestion control protocols for real-time multimedia and streaming applications, especially multicast video (e.g., IVS [3], RLM [15], etc.). There have also been numerous recent congestion control proposals for various reliable multicast applications (for a survey, see [20]). In contrast to these application-specific efforts, our aim is to develop a substrate that manages congestion and allows applications to implement their own adaptation policies. Perhaps, closer in spirit to our goal is the RAP protocol [19], which is a rate-based congestion control scheme intended for streaming applications. While the internal algorithms of the CM are in fact rate-based, its architecture is radically different from RAP. In particular, it is independent of the transport protocol and permits information to be shared between transports in a coherent manner (for example, it integrates congestion management across concurrent rate-based audio flows and window-based TCP flows).

2.2. Transport-level solutions

Motivated by the drawbacks of the above solutions and the desire to improve Web transfer performance, various researchers have proposed modifications to TCP itself. These proposals include Touch's work on TCP control block interdependence, in which he proposed that all TCP connections share portions of the protocol control block for path-related parameters [24]. Balakrishnan *et al.* proposed an integrated congestion control and loss recovery mechanism for TCP [1] called TCP-INT, which was refined and extended by Padmanabhan into a session-oriented variant of TCP [17]. Although these approaches solve most of the problems associated with the Web scenario, they are transport specific, and applications that use protocols other than TCP cannot take advantage of them.

Recently, a transport protocol for heterogeneous packet flows (HPF) has been described in [14]. A key difference between the CM and HPF is that the CM integrates congestion management across an ensemble of flows and provides a different adaptation API, while HPF does not consider the interactions between concurrent active flows.

3. Congestion Manager API

The key design guideline of the CM API is the end-to-end argument [21]. The application is given the flexibility to decide what to send at each point of time, as well as what proportion of bandwidth to allocate to each flow. The CM achieves this by not buffering any application data; instead it allows applications the opportunity to adapt to unexpected network changes until the data is actually sent. This design decision is a direct consequence of the ALF [6] approach to design. The API should accommodate a wide variety of traffic types, including TCP bulk transfers, short connections, real-time flows that can transmit at a continuum of rates, and layered streams. The API should also be flexible enough to permit different application styles; in particular, it should accommodate both synchronous transmitters (which are timer-driven) and asynchronous transmitters (which are triggered by events such as file reads or frame captures). In addition, the API must allow the CM to learn about network conditions from applications.

Data Structures:

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

Query:

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

Control:

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

Application callback:

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

Figure 2. Sender-side CM API

Figure 2 depicts the components exposed to applications by the CM through its API. The API has three types of functions: *query*, *control* and *callback* functions. The query function allows applications to obtain the rate, round trip time estimators and other parameters for a particular destination or group of destinations. A transport protocol like TCP could use `cm_query` to obtain the current mean and variance of the round-trip time for setting timeouts for packet retransmissions. The control functions are invoked either before or after actual data transmission to request and inform the CM about data transmission. The callback functions permit asynchronous notification to applications by the CM.

We now describe the typical order in which an application uses the CM API.Each flow must call `cm_open` call before any other control function calls are made. This allows the CM to initialize internal state corresponding to this flow, and returns a handle for the application to use in subsequent calls. An application uses the non-blocking `cm_request(nbytes)` call to obtain the consent of the CM for sending `nbytes` bytes of data. The parameters indicating the minimum meaningful data transmission size and the maximum tolerable latency are used by the CM for scheduling the request appropriately. An application callback `app_notify(nallowed)` is executed when the request is scheduled by the CM. The application can then transmit up to `nallowed` bytes soon after this. The application notifies the CM about the actual amount of data sent using `cm_notify(nsent)`. The `cm_notify()` function is separated from `cm_request()` in this request/callback/notify scheme to allow for repacketizing of data by applications, especially when they are permitted to send fewer bytes of data than they requested. Synchronous (timer-driven) applications can also use the CM API to adapt the frequency of their internal timers. This is enabled by providing a CM callback `change_rate(newrate)` informing them of changes in rate.

An application calls `cm_update(nrecd, duration, loss_occurred, rtt)` to inform the CM that `nrecd` bytes were received over `duration` seconds, that the observed RTT was `rtt`, and whether any losses occurred. The application could obtain feedback through ACKs as in TCP, through RTCP in the case of real-time applications, or through any other protocol. The CM uses this as a hint to internally update its sustainable sending rate and round-trip time estimates. Finally, an application calls `cm_close()` when a flow is terminated allowing the CM to destroy the internal state associated with that flow and repartition available bandwidth.

4. CM Architecture

In this section, we briefly present the CM's internal algorithms and protocols. The three main components of the CM architecture are its sound rate-control scheme, lightweight receiver feedback algorithm, and flexible flow scheduler.

4.1. Sound Congestion Control

The CM ensures proper congestion behavior by reacting to network congestion and probing for spare capacity using sound and robust mechanisms. Currently, the CM achieves this by using a rate-based additive-increase/multiplicative-decrease scheme. The additive increase component is friendly to comparable TCP flows since the algorithm emulates the TCP increase algorithm. Upon a loss, the rate is halved, and when persistent congestion occurs, the rate drops to a small value causing slow start [12] to occur. A detailed performance study of the rate-control scheme is presented in [2].

4.2. Receiver feedback

Receiver feedback is essential for stable end-to-end congestion control. The CM obtains *implicit feedback* when the receiver application or transport protocol (for example, TCP over CM) provides feedback to the sender application. The sender application can now notify the CM about the number of transmitted and received bytes generating no extra traffic. Since not all applications provide frequent feedback, we incorporate an *explicit feedback* protocol in the CM architecture with modifications to the receiver to respond to periodic probe messages from the sender and report loss or Explicit Congestion Notification (ECN) information. The protocol generates minimal network traffic and is resilient to losses. The CM reacts to losses of probe messages and responses by exponentially aging the rate to maintain stable network behavior. Details about the appropriate probe frequency and pseudocode for the probing protocol are available in [2].

4.3. Flow Scheduling

The CM internally schedules all requests using a Hierarchical Round Robin (HRR) Scheduler [13]. The scheduler apportions bandwidth among flows in proportion to pre-configured weights as well as receiver hints. The scheduler is invoked whenever any application makes a call to the CM. If the scheduler can immediately satisfy the next pending request based on the current bandwidth estimate of the CM, it performs an application callback informing it about the appropriate number of bytes allowed. Otherwise, it will notify the application at a future point in time based on the minimum number of bytes requested by the application and the sending rate.

5. Using the API

We have implemented TCP as well as an adaptive layered audio server on top of the CM. In this paper, we describe the implementation of a webserver over TCP. The architecture of the audio server is described in [2].

5.1. Webserver over TCP

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

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

Multiple representations of different sizes can exist for several of these objects. In such a case, the sender uses the `cm_query()` call and the `change_rate()` handler to adapt to changing available bandwidths (tracked by the CM) and picks the representation that maximizes receiver quality without incurring high latency. We are currently extending the HTTP content negotiation protocol [11] to incorporate these ideas.

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

Normally TCP's congestion management keeps track of a congestion window on a per-connection basis. When ACKs arrive, TCP updates the congestion window and transmits data if the congestion window allows it, and when it detects losses, the window is reduced by at least a factor of two. To use the CM, we modify TCP to call `cm_open()` when it establishes a connection. When `nsend` bytes of data arrive from the application (e.g., Web server), TCP/CM calls `cm_request(nsend)` to schedule the transmission of `nsend` bytes of data. When an ACK arrives from the network acknowledging `nrecd` bytes of data, TCP/CM calls `cm_update(nrecd)` as a useful hint to update the congestion state in the CM. It then calls `cm_request()` if the receiver-advertised flow control window has opened up and there is more data queued for transmission.

When the CM decides to service TCP/CM's request, it performs a callback using `app_notify()` to the TCP/CM send routine that accepts a parameter indicating the maximum amount of data it is allowed to transmit. The TCP send routine then transmits the minimum of the flow control window and the amount allowed by the CM in the callback. Immediately after transmitting this data, TCP/CM calls `cm_notify()` to update CM with the actual amount of data sent. This could be smaller than the amount permitted, and may even be zero at some points in time, for example, when the TCP/CM sender performs silly window syndrome avoidance [25].

Notice that we have eliminated the need for tracking and reacting to congestion in TCP/CM because proper congestion behavior is ensured by the CM and its callback-based transmission API. Notice also that duplicate ACK and timeout based loss recovery as well as end-to-end flow control based on advertised windows remain unchanged. Performance studies comparing TCP/CM with vanilla TCP are available in [2].

6. Conclusions and Future Directions

In this paper we have proposed the addition of a unified congestion management architecture to improve the interface that operating systems provide for Internet communications. This framework adds the following important functionality:

- The Congestion Manager (CM) ensures that all traffic adheres to basic Internet congestion control principles. This improves the stability of the Internet in general as well as the overall performance observed by communicating hosts.
- The CM allows multiple applications and transports to share information about the congestion state of the network. This eliminates redundant probing of the network and therefore improves the efficiency of network communication between hosts.
- The CM provides a simple API to make applications more network-aware. Using this, applications can adapt to the state of the network. Applications are free from the burden of discovering traffic congestion and can find domain-specific ways to adapt to changing network state.

We have implemented the CM framework in the VINT ns-2 [16] simulator. We have also implemented TCP and a layered audio application over this framework. Details and performance evaluation of the architecture can be found in [2]. We plan on moving the framework to a UNIX-based kernel. This UNIX-based implementation should provide an excellent testbed to experiment with various congestion control algorithms, adaptive applications and traffic prioritization techniques. We also intend to study the effect of aggregating flow information by domains to allow a greater set of flows to share congestion and bandwidth information and react to it.

Acknowledgments

The authors would like to thank William Adjie-Winoto, Jean Bolot, Chandra Boyapati, John Guttag, Raj Iyer, Suchitra Raman, and Elliot Schwartz for their useful comments and suggestions on this work.

References

[1] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. TCP Behavior of a Busy Web Server: Analysis and Improvements. In *Proc. IEEE INFOCOM*, Mar. 1998.

[2] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. Technical Report MIT-LCS-TR-771, Massachusetts Institute of Technology, Feb. 1999.

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

[4] 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.

[5] 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.

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

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

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

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

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

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

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

[13] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate Controlled Servers for Very High-Speed Networks. In *Proceedings of the IEEE Conference on Global Communications*, Dec 1990.

[14] J. Li, D. Dwyer, and V. Bharghavan. A Transport Protocol for Heterogeneous Packet Flows. In *Proc. IEEE INFOCOM*, Mar. 1999.

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

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

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

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

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

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

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

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

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

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

[25] G. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, Reading, MA, Jan 1995.

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