Internet Engineering Task Force
INTERNET DRAFT
Document: draft-balakrishnan-cm-01.txt

Hari Balakrishnan
MIT LCS
Srinivasan Seshan
IBM T.J. Watson

October 22, 1999
Expires: April 22, 2000

The Congestion Manager

Status of this Memo

1. Abstract

This document describes the Congestion Manager (CM), an end-system module that (i) enables an ensemble of multiple concurrent flows from a sender destined to the same receiver and sharing the same congestion properties to perform proper congestion avoidance and control, and (ii) allows applications to easily adapt to network congestion. This framework integrates congestion management across all applications and transport protocols. The CM maintains congestion parameters (available aggregate and per-flow bandwidth, per-receiver round-trip times, etc.) and exports an API that enables applications to learn about network characteristics, pass information to the CM, share congestion information with each other, and schedule data transmissions. This document focuses on applications and transport protocols with their own independent per-byte or per-packet sequence number information, and does not require modifications to the receiver protocol stack. The receiving application must provide feedback to the sending application about received packets and losses, and the latter uses the CM API to update the CM state. This document does not address networks with reservations or service discrimination.

2. Conventions used in this document:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [Bradner97].

FLOW
   A stream of packets that all share the same source and
     destination IP address, IP type-of-service, transport
     protocol, and source and destination transport port numbers.

MACROFLOW
     A group of flows that uses the same congestion management and
     scheduling algorithms, and shares congestion state
     information. Flows destined to different receivers MUST belong
     to different macroflows. Flows destined to the same receiver
     MAY belong to different macroflows. Flows that experience
     identical congestion behavior in the Internet and use the
     same congestion control algorithm SHOULD belong to the same
     macroflow.

APPLICATION Any software module that uses the CM. This includes
     user-level applications such as Web servers or audio/video
     servers, as well as in-kernel protocols such as TCP [Postel81]
     that use the CM for congestion control.

WELL-BEHAVED APPLICATION
     An application that only transmits when allowed by the CM and
     accurately accounts for all data that it has sent to the
     receiver by informing the CM using the CM API.

STREAM
     A logical sequence of packets generated by an application that
     directly corresponds (one-to-one) with a network-layer FLOW.

PATH MAXIMUM TRANSMISSION UNIT (PMTU)
     The size of the largest packet that the sender can transmit
     without it being fragmented en route to the receiver. It
     includes the sizes of all headers and data except the IP
     header.

CONGESTION WINDOW (cwnd)
     A CM state variable that modulates the amount of outstanding
     data between sender and receiver.

OUTSTANDING WINDOW (ownd)
     The number of bytes that has been transmitted by the source,
     but not known to have been either received by the destination
     or lost in the network.

INITIAL WINDOW (IW)
     The size of the sender's congestion window at the beginning of
     a macroflow.

DATA TYPE SYNTAX
     We use "u64" for unsigned 64-bit, "u32" for unsigned 32- bit,
   "u16" for unsigned 16-bit, "u8" for unsigned 8-bit, "i32" for
   signed 32-bit, "i16" for signed 16-bit quantities, "float" for
   IEEE floating point values. The type "void" is used to
   indicate that no return value is expected from a
   call. Pointers are referred to using "*" syntax, following C
   language convention.

3. Introduction

The CM is an end-system module that enables an ensemble of multiple concurrent flows to perform proper congestion avoidance and control, and allows applications to easily adapt their transmissions to prevailing network conditions. It integrates congestion management across all applications and transport protocols. It maintains congestion parameters (available aggregate and per-flow bandwidth, per-receiver round-trip times, etc.) and exports an API that enables applications to learn about network characteristics, pass information to the CM, share congestion information with each other, and schedule data transmissions. All data transmissions MUST be done with the explicit consent of the CM via this API to ensure proper congestion behavior.

This document focuses on applications and networks where the following conditions hold:

1. Applications are well-behaved with their own independent per-byte or per-packet sequence number information, and use the CM API to update internal state in the CM.

2. Networks are best-effort without service discrimination or reservations. In particular, it does not address situations where different flows between the same pair of hosts traverse paths with differing characteristics.

The Congestion Manager framework can be extended to support applications that do not provide their own feedback and to differentially served networks. These extensions will be addressed in later documents.

The CM is motivated by two main goals:

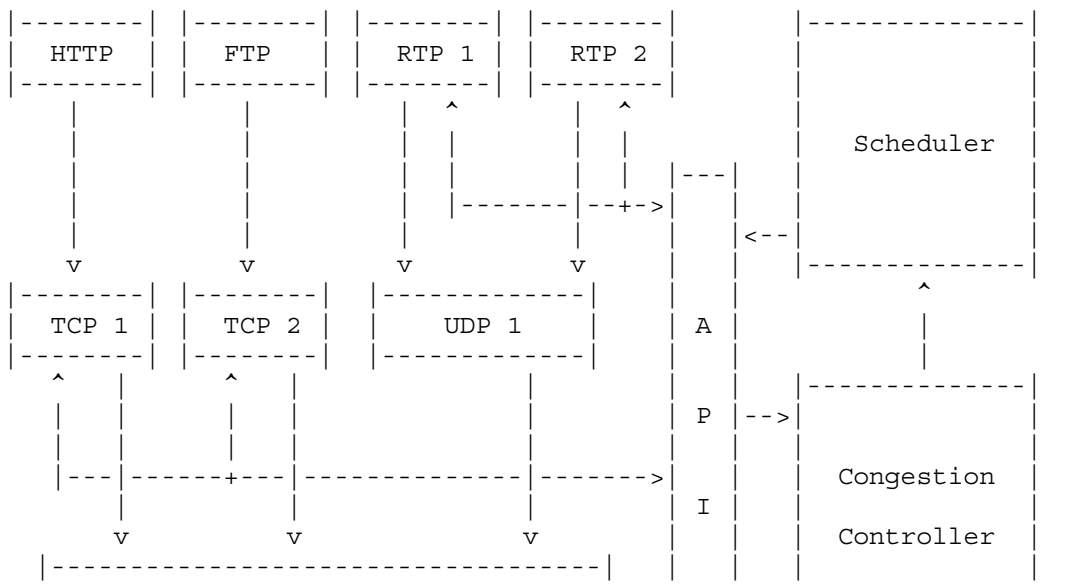
(i) Enable efficient multiplexing. Increasingly, the trend on the Internet is for unicast data senders (e.g., Web servers) to transmit heterogeneous types of data to receivers, ranging from unreliable real-time streaming content to reliable Web pages and applets. As a result, many logically different flows share the same path between sender and receiver. For the Internet to remain stable, each of these streams must incorporate control protocols that safely probe for spare bandwidth and react to congestion. Unfortunately, these concurrent flows typically compete with each other for network resources, rather than share them effectively. Furthermore, they do not learn from each other about the state of the network. Even if they each independently implement congestion control (e.g., a group of TCP connections each implementing the algorithms in [Jacobson88, Stevens97]), the ensemble of flows tends to be more aggressive in the face of congestion than a single TCP connection implementing standard TCP congestion control and avoidance [Balakrishnan98].

(ii) Enable application adaptation to congestion. Increasingly popular real-time streaming applications run over UDP using their own user-level transport protocols for good application performance, but in most cases today do not adapt or react properly

to network congestion. By implementing a stable control algorithm and exposing an adaptation API, the CM enables easy application adaptation to congestion. Applications adapt the data they transmit to the current network conditions.

The CM framework builds on recent work on TCP control block sharing [Touch97], integrated TCP congestion control (TCP-Int) [Balakrishnan98] and TCP sessions [Padmanabhan98]. [Touch97] advocates the sharing of some of the state in the TCP control block to improve transient transport performance and describes sharing across an ensemble of TCP connections. [Balakrishnan98] and [Padmanabhan98] describe several experiments that quantify the benefits of sharing congestion state, including improved stability in the face of congestion and better loss recovery. Integrating loss recovery across concurrent connections significantly improves performance because losses on one connection can be detected by noticing that later data sent on another connection has been received and acknowledged. The CM framework extends these ideas in two significant ways: (i) it extends congestion management to non-TCP streams, which are becoming increasingly common and often do not implement proper congestion management, and (ii) it provides an API for applications to adapt their transmissions to current network conditions. For an extended discussion of the motivation for the CM, its architecture, API, algorithms and performance, see [Balakrishnan99].

The resulting end-host protocol architecture at the sender is shown in Figure 1. The CM helps achieve network stability by implementing stable congestion avoidance and control algorithms that are "TCP-friendly" [Mahdavi98] based on algorithms described in [Stevens97]. However, it does not attempt to enforce proper congestion behavior for all applications (but it does not preclude a policer on the host that performs this task). Note that while the policer at the end-host can use CM, protecting the network against compromises to the CM and policer requires router machinery [Floyd99a]. We do not address this issue further in this document.



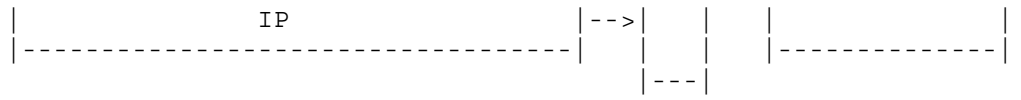


Figure 1

The key components of the CM framework are (i) the API, (ii) the congestion controller, (iii) the scheduler. The API is (in part) motivated by the ideas of application-level framing (ALF) [Clark90] and is described in Section 4. The CM internals (Section 5) include a congestion controller (Section 5.1) and a scheduler to orchestrate data transmissions between concurrent flows in a macroflow (Section 5.2). The congestion controller adjusts the aggregate transmission rate between sender and receiver based on its estimate of congestion in the network. It obtains feedback about its past transmissions from applications themselves via the API. The scheduler allocates available bandwidth amongst the different flows within each macroflow and notifies applications when they are permitted to send data. A future document will describe the sender-receiver protocol and header formats that will handle applications that do not incorporate their own feedback to the CM. (This document focuses on well-behaved applications.)

4. CM API

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

The function prototypes below follow standard C language convention.

4.1 State maintenance

1. Open: All applications MUST call cm_open(u32 dst) before using the CM API. dst is the 32-bit IPv4 address. This returns an i32 handle, cm_flowid, for the application to use for all further CM API invocations for that flow. If cm_flowid is -1, then the cm_open() failed and the flow cannot use the CM.

2. Close: When a flow terminates, the application SHOULD invoke cm_close(i32 cm_flowid) to inform the CM about the termination of the flow. This allows CM to clean up its internal state in a timely manner. The CM also cleans up its internal state for the flow (and if necessary the associated macroflow) when there has been no activity for CM_CLEANUP seconds (the current value is 75 seconds).

3. Packet size: cm_mtu(i32 cm_flowid) returns the estimated PMTU of the path between sender and receiver. Internally, this information may either be statically configured, or obtained via path MTU discovery [Mogul90].

4.2 Data transmission

The CM accommodates three types of senders, including streams that use ALF to dynamically adapt their content based on prevailing network conditions.

1. Buffered transmission. A sender stream can call cm_send(i32 cm_flowid, (u8*) data, u32 length) to transmit data via this API. Here, the CM is on the data path and buffers the data for eventual transmission, which in turn occurs at a time determined by its congestion controller and scheduler. The data buffer MUST contain a raw IP datagram (excluding the IP header) ready to be sent, and length MUST be the length of the entire IP payload (i.e, excluding the IP header). A disadvantage of this method is that ALF-based applications are not accommodated, because the sender does not get to revisit and change its prior transmission decisions once data is buffered in the CM.

2. Callback-based transmission. The callback-based transmission API puts the stream in firm control of deciding what to transmit at each point in time. To achieve this, the CM does not buffer any data; instead, it allows streams the opportunity to adapt to unexpected network changes at the last possible instant. This enables streams to "pull out" and repacketize data upon learning about any rate change, which is hard to do once the data has been buffered. A stream wishing to send data in this style can call cm_request(i32 cm_flowid). After some time, depending on the rate, the CM invokes a callback using cmapp_send(), which is a grant for the stream to send up to PMTU bytes. Note that cm_request() does not take the number of bytes or MTU-sized units as an argument; each call to cm_request() is an implicit request for sending up to PMTU bytes. Section 5.2 describes how these requests are scheduled and callbacks made. The callback-style API is the recommended choice for ALF-based streams.

3. Synchronous-style. The above callback-based API accommodates a class of ALF streams that are "asynchronous." Asynchronous stream transmissions are triggered by asynchronous events like file reads. However, many applications use "synchronous" streams, which transmit periodically based on their own internal timers (e.g., an audio sender that sends at a constant sampling rate). While CM callbacks could be configured to periodically interrupt such transmitters, the structure of such applications is less affected if they retain their original timer-based loop. Thus, the CM exports an API that allows such streams to be informed of changes in rates using the cmapp_update(u64 newrate, u32 srtt, u32 rttdev) callback function, where newrate is the new rate in bits per second for this flow, srtt is the current smoothed round trip time estimate in microseconds, and rttdev is the mean linear deviation of the round-trip time estimate. In response, the stream MUST adapt its packet size or change its timer interval to not exceed the allowed rate. Of course, it may choose not to use all of this rate.

To avoid unnecessary cmapp_update() callbacks that the application will only ignore, the stream can use the cm_thresh(float downthresh, float upthresh) function at any stage in its execution. In response, the CM will invoke the callback when the rate

decreases to less than (downthresh * lastrate) or increases to more than (upthresh * lastrate), where lastrate is the rate last notified to the stream. This information is used as a hint by the CM, in the sense the cmapp_update() can be called even if these conditions are not met. (At this point, the API does not include a callback when the round-trip time or variation changes significantly; this may be changed in the future.)

An application can query the current CM state by using cm_query(u32 cm_flowid, u64* rate, u32* srtt, u32* rttdev). This sets the rate variable to the current rate estimate in bits per second, the srtt variable to the current smoothed round-trip time estimate in microseconds, and rttdev to the mean linear deviation of the round-trip time in microseconds.

Note that a stream can use more than one of the above transmission APIs at the same time. In particular, the knowledge of sustainable rate is useful for asynchronous streams as well as synchronous ones; e.g., an asynchronous Web server disseminating images using TCP may use cmapp_send() to schedule its transmissions and cmapp_update() to decide whether to send a low- resolution or high-resolution image. A TCP implementation using the CM is described in Section 6.1.1, where the benefit of the cm_request() API for TCP will become apparent.

4.3 Application notification

When a stream receives feedback from receivers, it MUST use cm_update(i32 cm_flowid, u32 nsent, u32 nrecd, u8 lossmode, i32 rtt) to inform the CM about events such as congestion losses, successful receptions, type of loss (timeout event, Explicit Congestion Notification [Ramakrishnan97], etc.) and round-trip time samples. The nsent parameter indicates how many bytes were sent, the nrecd parameter identifies how many of those bytes were received. The rtt value indicates the round-trip time measured during the transmission of these bytes. The rtt value must be set to -1 if no valid round-trip sample was obtained by the stream. The lossmode parameter provides an indicator of how a loss was detected. A value of CM_PERSISTENT indicates that the application believes congestion to be severe, e.g., a TCP that has experienced a timeout. A value of CM_TRANSIENT indicates that the application believes that the congestion is not severe, e.g., a TCP loss detected using duplicate (selective) acknowledgements or other data-driven techniques. A value of CM_ECN indicates that the receiver echoed an explicit congestion notification [Ramakrishnan98] message. Finally, a value of CM_NOLOSS indicates that no congestion-related loss has occurred.

cm_notify(i32 cm_flowid, u32 nsent) MUST be called when data is transmitted from the host (e.g., by the IP output routine) to inform the CM that nsent bytes were just transmitted on the specified flow. This allows the CM to update its estimate of ownd for the macroflow and for the flow. If a stream does not transmit any data upon a cmapp_send() callback invocation, it SHOULD call cm_notify(flowid, 0) to allow the CM to permit other flows in the macroflow to transmit data.

4.4 Sharing granularity

The CM needs to decide which flows belong to a single macroflow and share congestion information. The API provides two functions that allow applications to group their streams into the same macroflows.

cm_getmacroflow(i32 cm_flowid) returns a unique i32 macroflow identifier. cm_setmacroflow(i32 cm_macroflowid, i32 cm_flowid) sets the macroflow of the flow cm_flowid to cm_macroflowid. If the cm_macroflowid that is passed to cm_setmacroflow() is -1, then a new macroflow is constructed and this is returned to the caller. Each call to cm_setmacroflow() overrides the previous macroflow association for the flow, should one exist.

The default aggregation method (i.e., within which macroflow should CM place a flow if the application does not use cm_setmacroflow) is as yet unresolved.

5. CM internals

This section describes the internal components of the CM. It includes a congestion controller and a scheduler, with well-defined interfaces exported by them.

5.1 Congestion controller

Associated with each macroflow is a congestion control algorithm; the collection of all these algorithms comprises the congestion controller of the CM. The control algorithm decides when and how much data can be transmitted by a macroflow. It uses application notifications (Section 4.3) from concurrent streams on the same macroflow to build up information about the congestion state of the different network paths.

The congestion controller MUST implement "TCP-friendly" [Mahdavi98] congestion control algorithms. Several macroflows MAY (and indeed, often will) use the same congestion control algorithm but each macroflow maintains separate state about the network used by its flows.

The congestion control module MUST implement the following interfaces (these are not directly visible to applications; they are within the context of a macroflow):

- void query(u64 *rate, u32 *srtt, u32 *rttdev): This function returns the estimated rate (in bits per second), smoothed round trip time (in microseconds) and mean linear deviation of the round-trip time estimate (in microseconds) for the macroflow.

- void notify(u32 nsent): This function MUST be used to notify the congestion control module whenever data is sent by a flow. The nsent parameter indicates the number of bytes just sent.

- void update(u32 nsent, u32 nrecd, u32 rtt, u32 lossmode): This function is called whenever any of the CM flows associated with a macroflow identifies that data has reached the receiver or has been lost en route. The nrecd parameter indicates the number of

bytes that have just arrived at the receiver. The nsent parameter is the sum of the number of bytes just received and the number of bytes identified as lost en route. The rtt parameter is the estimated round trip time in microseconds during the transfer. The lossmode parameter provides an indicator of how a loss was detected (section 4.3).

The congestion control module MUST also call the associated scheduler's schedule() function (section 5.2) when it believes that the current congestion state allows an MTU-sized packet to be sent.

5.2 Scheduler

While it is the responsibility of a macroflow's congestion controller to determine when and how much data can be transmitted, it is the responsibility of a its scheduler to determine which of its flows gets the opportunity to transmit data at any time.

The Scheduler MUST implement the following interfaces:

- void schedule(u32 num_bytes): When the congestion controller determines that data can be sent, the schedule() routine MUST be called with the number of bytes that can be sent. In turn, the scheduler MAY call a stream's cmapp_send() function.

- float query_share(u32 cm_flowid): This call returns the described flow's share of the total bandwidth available to the macroflow. This call combined with the query call of the congestion control provides the information to satisfy a stream's cm_query() request.

- void notify(u32 cm_flowid, u32 nsent): This interface is used to notify the scheduler whenever data is sent by a CM flow. The nsent parameter indicates the number of bytes just sent by the application.

- void delete_flow(u32 cm_flowid): This call notifies the scheduler that a particular flow has been closed.

6. Examples

6.1 Example applications

This section describes the use of the CM API by an asynchronous application (an implementation of a TCP sender) and a synchronous application (an audio server).

6.1.1 TCP/CM

TCP/CM removes the congestion management functionality from TCP and uses the CM to control the flow of outgoing data. The CM eliminates the need for tracking and reacting to congestion in TCP, because the CM and its transmission API ensure proper congestion behavior. TCP/CM continues to perform all functions related to loss recovery and flow control.

This section describes the modifications necessary to make TCP

Reno-like transmitter use the CM. Other loss recovery techniques such as NewReno [Floyd99b] and integrated loss recovery across an ensemble of concurrent connections [Balakrishnan98] can be accommodated as easily.

TCP using the CM MUST use the callback-based API (i.e. it must implement a cmapp_send() callback function). TCP/CM identifies what data it should send only upon the arrival of an acknowledgement (ACK) or expiration of a timer. As a result, it requires tight control of when and if new data or retransmissions are sent -- using the buffered or synchronous API is therefore undesirable.

When TCP/CM either connects to or accepts a connection from another TCP (which can be any TCP that conforms to RFC793 [Postel81]), it calls cm_open() to associate the TCP/CM connection with a cm_flowid. TCP/CM is also modified to have its own outstanding window (tcp_ownd) estimate. Whenever, data is sent from its cmapp_send() callback, TCP/CM updates its tcp_ownd value. The value of tcp_ownd is also updated after each cm_update() call. In addition, TCP/CM maintains a count of the number of outstanding segments (seg_cnt). At any time, TCP/CM can calculate the average segment size (avg_seg_size) as tcp_ownd/seg_cnt.

TCP/CM makes the following key modifications to a conformant TCP implementation's output routines:

1. All congestion window (cwnd) checks and computations are removed.

2. When application data is available, TCP/CM output routines performs all non-congestion checks (related to flow control, such as the Nagle algorithm, receiver advertised window check, etc). If these checks pass, the output routine queues the data and calls cm_request() for the flow.

3. If incoming duplicate or selective ACKs or a timeout result in a loss being detected, the retransmission is also placed in a queue and cm_request() is called for the flow.

4. The cmapp_send() callback for TCP/CM is set to a simple output routine. If any retransmission is enqueued, the routine outputs the lost segment. Otherwise, the routine outputs as much new data as the TCP/CM connection state allows. However, cmapp_send() never sends more than a single segment per call.

The IP output routine on the host calls cm_notify() when the data is actually sent out.

The TCP/CM input routines are modified as follows:

1. RTT and RTO estimation is done as usual using either timestamps [Jacobson92] or Karn's algorithm. Any RTT estimate that is generated is passed to CM via the cm_update call.

2. All cwnd and slow start threshold (ssthresh) updates and computations are removed.

3. Upon the arrival of an ACK (with value ack_seq) for new data,

TCP/CM computes the value of in_flight (the amount of data in flight) as (snd_max - ack_seq) (i.e. maximum sequence sent - current ACK). TCP/CM then calls cm_update(flowid, tcp_ownd - in_flight, 0, CM_NOLOSS, rtt). However, if the ACK contains an ECN notification, TCP/CM calls cm_update(flowid, tcp_ownd - in_flight, 0, CM_ECN, rtt) instead.

4. Upon the arrival of a duplicate ACK, TCP/CM must check its dupack count (dup_acks) to determine its action. If dup_acks < 3, the TCP/CM does nothing. If dup_acks is 3, TCP/CM assumes that a segment was lost and that at least 3 segments reached to generate these duplicate ACKs. Therefore, it calls cm_update(flowid, 4 * avg_seg_size, 3 * avg_seg_size, CM_TRANSIENT, rtt). TCP/CM also enqueues a retransmission of the lost segment and calls cm_request(). If dup_acks > 3, TCP/CM infers that a segment has reached the other end and caused a duplicate ACK to be sent. As a result, it calls cm_update(flowid, avg_seg_size, avg_seg_size, CM_NOLOSS, rtt). This allows the CM to accurately track ownd.

When the TCP/CM retransmission timer expires, the sender identifies that a segment has been lost and calls cm_update(flowid, avg_seg_size, 0, CM_PERSISTENT, 0) to signify the occurrence of persistent congestion to the CM. TCP/CM also enqueues a retransmission of the lost segment and calls cm_request().

Finally, when the connection is closed, TCP/CM performs a cm_close() to allow CM to reclaim the associated state.

6.1.2 Audio server

A typical audio source often has access to an audio sample in a multitude of data rates and qualities. The objective of this application is then to deliver the highest possible quality of audio (typically the highest data rate) to its client. The selection of which version of audio to transmit should be based on the current congestion state of the network. In addition, the source will want audio delivered to its users at a consistent sampling rate. As a result, it must send data a regular rate, without delaying transmissions thereby reducing buffering before playback at the client. To meet these requirements, this application can use the synchronous sender API (Section 4.2).

When the source first starts, it uses cm_query() to get an initial estimate of network bandwidth and delay. It then chooses an encoding that does not exceed these estimates and begins transmitting data. The source also implements the cmapp_update() callback. When the CM determines that network characteristics have changed, it calls the application's cmapp_update() function and passes it a new rate and round-trip time estimate. The source MUST change its choice of audio encoding if its current transmission rate exceeds the new estimates.

To use the CM, the application must incorporate feedback from the receiver. In this example, it must periodically (at least once or twice per round trip time) determine how many of its packets arrived at the receiver. When the source gets this feedback, it MUST use cm_update() to inform the CM of this new information.

This allows in the CM to update ownd and may result in the CM changing its estimates and calling cmapp_update() of the streams of the macroflow.

6.2 Example congestion controller

To illustrate the responsibilities of a congestion controller, this section describes some of the actions of a simple TCP-like congestion controller that implements Additive Increase Multiplicative Decrease congestion control (AIMD_CC):

- query(): AIMD_CC returns the current congestion window (cwnd) divided by the smoothed rtt (srtt) as its bandwidth estimate for the macroflow. It also returns the smoothed rtt estimate as srtt and the mean linear deviation as rttdev. These are calculated according to the TCP algorithm [Jac88].

- notify(): AIMD_CC adds the number of bytes sent to its outstanding data window (ownd).

- update(): AIMD_CC subtracts nsent from ownd. If the value of rtt is non-zero, AIMD_CC updates srtt using the TCP srtt calculation. If the update indicates that data has been lost, AIMD_CC sets cwnd to 1 MTU if the loss_mode is CM_PERSISTENT and to ownd/2 (with a minimum of 1 MTU) if the loss_mode is CM_TRANSIENT or CM_ECN. AIMD_CC also sets its internal ssthresh variable to ownd/2. If no loss had occurred, AIMD_CC mimics TCP slow start and linear growth modes [Stevens97]. It increments cwnd by nsent when cwnd < ssthresh (bounded by a maximum of ssthresh-cwnd) and by nsent * MTU/cwnd when cwnd > ssthresh.

- When cwnd or ownd are updated and indicate that at least one MTU may be transmitted, AIMD_CC calls the CM to schedule a transmission.

6.3 Example Scheduler Module

To clarify the responsibilities of a scheduler module, this section describes some of the actions of a simple round robin scheduler module (RR_sched):

- schedule(): RR_sched schedules as many flows as possible in round robin fashion.

- query_share(): RR_sched returns 1/(number of flows in macroflow).

- notify(): RR_sched does nothing. Round robin scheduling is not affected by the amount of data sent.

- delete_flow(): RR_sched adjusts its round robin schedule to omit the deleted flow.

7. Security considerations

The CM provides many of the same services that the congestion control in TCP provides. As such, it is vulnerable to many of the same security problems. For example, incorrect reports of losses

and transmissions will give the CM an inaccurate picture of the network's congestion state. By giving CM a high estimate of congestion, an attacker can reduce the performance observed by applications. The more dangerous form of attack is giving CM a low estimate. This would cause CM to be overly aggressive and allow data to be sent much more quickly than sound congestion control policies would allow.

8. References

[Balakrishnan98] Balakrishnan, H., Padmanabhan, V., Seshan, S., Stemm, M., and Katz, R., "TCP Behavior of a Busy Web Server: Analysis and Improvements," Proc. IEEE INFOCOM, San Francisco, CA, March 1998.

[Balakrishnan99] Balakrishnan, H., Rahul, H., and Seshan, S., "An Integrated Congestion Management Architecture for Internet Hosts," Proc. ACM SIGCOMM, Cambridge, MA, September 1999.

[Bradner96] Bradner, S., "The Internet Standards Process --- Revision 3", BCP 9, RFC-2026, October 1996.

[Bradner97] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC-2119, March 1997.

[Clark82] Clark, D., "Window and Acknowledgement Strategy in TCP," RFC-813, July 1982.

[Clark90] Clark, D. and Tennenhouse, D., "Architectural Consideration for a New Generation of Protocols", Proc. ACM SIGCOMM, Philadelphia, PA, September 1990.

[Floyd99a] Floyd, S. and Fall, K.," Promoting the Use of End-to-End Congestion Control in the Internet," IEEE/ACM Trans. on Networking, 7(4), August 1999, pp. 458-472.

[Floyd99b] Floyd, S. and Henderson, T., "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC-2582, April 1999. (Experimental.)

[Jacobson88] Jacobson, V., "Congestion Avoidance and Control," Proc. ACM SIGCOMM, Stanford, CA, August 1988.

[Jacobson92] Jacobson, V., Braden, R. and Borman, D., "TCP Extensions for High Performance," RFC1323, May 1992.

[Mahdavi98] Mahdavi, J. and Floyd, S., "The TCP Friendly Website," http://www.psc.edu/networking/tcp_friendly.html

[Mogul90] Mogul, J. and Deering, S., "Path MTU Discovery," RFC-1191, November 1990.

[Padmanabhan98] Padmanabhan, V., "Addressing the Challenges of Web Data Transport," PhD thesis, Univ. of California, Berkeley, December 1998.

[Postel81] Postel, J. (ed.), "Transmission Control Protocol",

RFC-793, September 1981.

[Ramakrishnan98] Ramakrishnan, K. and Floyd, S., "A Proposal to Add Explicit Congestion Notification (ECN) to IP," RFC-2481, January 1999. (Experimental.)

[Stevens97] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC-2001, January 1997.

[Touch97] Touch, J., "TCP Control Block Interdependence," RFC-2140, April 1997. (Informational.)

9. Acknowledgments

We thank Sally Floyd, Mark Handley, Steve McCanne, and Vern Paxson for useful feedback and suggestions on the CM architecture. We also thank David Andersen, Deepak Bansal, Dorothy Curtis, and Hariharan Rahul for their work on the CM design and implementation.

10. Authors' addresses

Hari Balakrishnan
Laboratory for Computer Science
545 Technology Square
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: hari@lcs.mit.edu
Web: http://wind.lcs.mit.edu/~hari/

Srinivasan Seshan
IBM T.J. Watson Research Center
30 Saw Mill River Rd.
Hawthorne, NY 10532
Email: sseshan@us.ibm.com
Web: http://www.research.ibm.com/people/s/srini/

Full Copyright Statement