Internet Engineering Task Force
INTERNET DRAFT
Document: draft-ietf-cm-cm-00.txt

Hari Balakrishnan
MIT
Srinivasan Seshan
IBM

June 23, 1999
Expires: December 23, 1999

# 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 sharing the same receiver and congestion behavior to display proper congestion behavior, 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, obtain information from and pass information to the CM, share congestion information, and schedule data transmissions. This document focuses on applications and transport protocols with their own independent per-byte or per-packet sequence number information. It 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 [2].

FLOW

A "flow" is a stream of packets that all share the same source and destination IP address, transport protocol, and 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 desire the same congestion control algorithm SHOULD belong to the same macroflow.

APPLICATION

Any software module that uses the CM is called an "application." This includes user-level applications such as Web servers or audio/video servers, as well as in-kernel protocols such as TCP [3] 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 sends, by informing the CM using the CM API.

STREAM

A "stream" is 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 PMTU is the size of the largest packet that the sender can transmit without it being fragmented. 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, is called OUTSTANDING. OUTSTANDING MUST not exceed the CONGESTION WINDOW.

INITIAL WINDOW (IW)

The initial window is the size of the source'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 display proper congestion behavior and allows applications to adapt to network congestion. It integrates congestion management across all applications and transport protocols. The CM maintains congestion parameters (available aggregate and per-flow bandwidths, per-receiver round-trip times, etc.) and exports an API to enable applications to learn about network characteristics, obtain information from and pass information to the CM, share congestion information, 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. Well-behaved applications with their own independent per-byte or per-packet sequence number information.
2. Best-effort networks 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 can be extended to support applications that do not provide their own feedback. 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 ("servers") to transmit a wide variety of data to receivers ("clients"), 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), the ensemble of flows tends to be more aggressive in the face of congestion than a single TCP connection implementing congestion control and avoidance.

(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 a simple API, the CM enables easy application adaptation to congestion.

The resulting end-host protocol architecture at the source is shown in Figure 1. The CM helps achieve network stability by implementing stable congestion avoidance and control algorithms that are "TCP-friendly" [4]. 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, the network has to be protected against compromises to the CM at the end hosts, a task that requires router machinery. We do not address this issue further in this document.

```
   |--------| |--------| |--------| |--------|        |--------------|
   |  HTTP  | |  FTP   | |  RTP 1 | |  RTP 2 |        |              |
   |--------| |--------| |--------| |--------|        |              |
       |          |          |  ^       |  ^          |              |
       |          |          |  |       |  |          |   Scheduler  |
       |          |          |  |       |  |  |---|   |              |
       |          |          |  |-------|--+->|   |   |              |
       |          |          |          |     |   |<--|              |
       v          v          v          v     |   |   |--------------|
   |--------| |--------|   |-------------|    |   |         ^
   |  TCP 1 | |  TCP 2 |   |    UDP 1    |    | A |         |
   |--------| |--------|   |-------------|    |   |         |
      ^   |      ^   |               |        |   |   |--------------|
      |   |      |   |               |        | P |-->|              |
      |   |      |   |               |        |   |   |              |
      |---|------+---|---------------|------->|   |   |  Congestion  |
          |          |               |        | I |   |              |
          v          v               v        |   |   |  Controller  |
     |------------------------------------|   |   |   |              |
     |               IP                   |-->|   |   |              |
     |------------------------------------|   |   |   |--------------|
                                              |---|
```

Figure 1

The key components of the CM framework are (i) the CM API, (ii) the Congestion Controller, (iii) the Scheduler. The API is motivated by the ideas of application-level framing [5] and is described in Section 4. The CM internals (Section 5) consist of a Congestion Controller (Section 5.1), 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. The Scheduler apportions 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 the class of "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 path bandwidth 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 invoke cm_open(u32 dst) before using the CM API. dst is the 32-bit IPv4 address. This returns a i32 handle, cm_flowid, which the application MUST use for all further CM API invocations for that flow. If cm_flowid is -1, then the cm_open() failed and that 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.

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 discovery [6].

### 4.2 Data transmission

The CM accommodates a variety of sources, including ALF-based streams. There are three styles of data transmission using the CM.

1. Callback-style. The callback-style 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. Thus, this enables streams to "pull out" and repacketize data upon learning about any rate change. A stream wishing to send data in this style MUST 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. The callback-style API is the recommended choice for ALF-based streams.

2. Buffered-style. Streams that do not want to use the callback-style API can use cm_send(i32 cm_flowid, (u8*) data, u32 length). The CM buffers the data for eventual transmission. 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).

3. Synchronous-style. The above callback-style API (#1) accommodates a class of transmitters that are ASYNCHRONOUS. Asynchronous transmitters do not transmit based on a periodic clock, but do so triggered by asynchronous events like file reads or captured frames. On the other hand, SYNCHRONOUS transmitters transmit periodically based on their own internal timers. While CM callbacks could be configured to interrupt such transmitters periodically, the transmit loop of such applications is less affected if they retain their original timer-based loop. Thus, such applications will benefit from a CM callback informing them of changes in rates, for which the CM provides the cmapp_update(u64 newrate, u32 srtt) callback function, where newrate is the new rate in bits per second for this flow and srtt is the current smoothed round trip time estimate in microseconds. In response, the stream MUST adapt its packet size or change its timer interval to conform to the allowed rate.

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

Note that a given stream can use more than one of the above transmission APIs for different reasons. For example, 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 could use cmapp_send() to schedule its transmissions and cmapp_update() to decide whether to send a low-resolution or high-resolution image.

### 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 [7], 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. 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 message. Finally, a value of CM_NOLOSS indicates that no congestion-related loss has occurred.

cm_notify(u32 dst, u32 nsent) MUST be called in the IP output routine to inform the CM that nsent bytes were just transmitted on a given flow. This allows the CM to update its estimate of the number of outstanding bytes for the macroflow as well as for the flow. If a stream does not transmitany data upon a cmapp_send() callback invocation, it SHOULD call cm_notify(dst, 0) to allow the CM to permit other flows in the macroflow to transmit data.

### 4.4 Querying

If applications wish to learn about per-stream available bandwidth and round-trip time, they SHOULD use the CM's cm_query(u32 flowid, u64* rate, u32* srtt) call, which fills in the desired quantities.

## 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 flow. 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 a "TCP-friendly" [4] congestion control algorithm. Several macroflows MAY (and indeed, often will) use the same congestion control algorithm but each macroflow maintains 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): This function returns the estimated rate (in bits per second) and smoothed round trip time (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 an application. The nsent parameter indicates the number of bytes just sent by the application.

- 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 the congestion control module to determine when and how much data can be transmitted, it is the responsibility of a macroflow's scheduler module to determine which of the flows should get the opportunity to transmit data.

The Scheduler MUST implement the following interfaces:

- void schedule(u32 num_bytes): When the congestion control module 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 the cmapp_send() function that CM applications must provide.

- 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 an application's cm_query() request.

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

# 6. Examples

## 6.1 Example Applications

The following describes the possible 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

A TCP MUST use the cmapp_send() callback API. TCP only identifies which data it should send upon the arrival of an acknowledgement or expiration of a timer. As a result, it requires tight control over when and if new data or retransmissions are sent.

When the TCP sender desires to send a packet, it requests CM to schedule the transmission using cm_request(). When the CM decides to service a TCP send request, it performs a callback using cmapp_send() to the TCP send routine. The TCP send routine then transmits the minimum of the flow control window and one Maximum Segment Size (MSS) according to the TCP specification.  The MSS should be determined using cm_mtu() (Section 4.1).  The IP output routine MUST call cm_notify() to inform it how many bytes were actually transmitted, which could in general be smaller than MSS (e.g., when the TCP/CM sender performs silly window syndrome avoidance [8], or when the receiver's flow control window constrains the number of bytes.)

The CM eliminates the need for tracking and reacting to congestion in TCP, because the CM and its transmission API ensure proper congestion behavior.  Loss recovery is still performed by TCP based on fast retransmissions and recovery as well as timeouts. The TCP sender calls cm_update() on the arrival of every acknowledgement and when timeouts occur.

### 6.1.2 Audio Server

A typical audio application often has access to the sample in a multitude of data rates and qualities. The objective of the application is then to deliver the highest possible quality of audio (typically the highest data rate) its clients. 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, minimizing delaying transmissions and reducing buffering before playback. To meet these requirements, this application can use the synchronous sender API (Section 4.2).

When the source first starts, it uses the cm_query() call 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 application 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 application MUST change its choice of audio encoding to ensure that it does not exceed these new estimates.

To use the CM, the application must incorporate feedback from the receiver. In this example, it must periodically (typically 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 results in the CM updating ownd and may result in CM changing its estimates and calling cmapp_update() of the streams of the macroflow.

6.3 Example Congestion Control Module

To illustrate the responsibilities of a congestion control module, the following describes some of the actions of a simple TCP-like congestion control module 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. It returns the smoothed rtt estimate as srtt.

- 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 cwnd/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 cwnd/2. If no loss had occurred, AIMD_CC mimics TCP slow start and linear growth modes. 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.

8.4 Example Scheduler Module

To clarify the responsibilities of a scheduler module, the following 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.

## 7. Security Considerations

The 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 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


1 Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.
2 Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
3 Postel, J., "Transmission Control Protocol", RFC793, April 1997.
4 Mahdavi, J. and Floyd, S., "The TCP Friendly Website", http://www.psc.edu/networking/tcp_friendly.html
5 Clark, D. and Tennenhouse, D., "Architectural Consideration for a New Generation of Protocols", Proc. ACM SIGCOMM, September 1990.
6 Mogul, J. and Deering, S., "Path MTU Discovery", RFC 1191, November 1990.
7 Ramakrishnan, K. and Floyd, S., "A Proposal to Add Explicit Congestion Notification to IPv6 and TCP", Internet Draft draft-kksjf-ecn-00.txt.
8 Clark, D., "Window and Acknowledgement Strategy in TCP", RFC 813, July 1982.


## 9. Acknowledgments


Sally Floyd, Mark Handley, Steve McCanne and Vern Paxson provided useful feedback and suggestions that contributed to and improved the the CM architecture. We are grateful to them. We also thank Hariharan Rahul for his contributions to the design of the CM.


## 10. Authors' Addresses


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

Srinivasan Seshan
30 Saw Mill River Rd.
Hawthorne, NY 10532

srini@watson.ibm.com
http://www.research.ibm.com/people/s/srini/

## Full Copyright Statement