Network Working Group
Request for Comments: 3124
Category: Standards Track

H. Balakrishnan
MIT LCS
S. Seshan
CMU

June 2001

The Congestion Manager

Status of this Memo

Copyright Notice

Abstract

This document describes the Congestion Manager (CM), an end-system module that:

(i) Enables an ensemble of multiple concurrent streams 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.

1. 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].

STREAM

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

MACROFLOW

A group of CM-enabled streams that all use the same congestion management and scheduling algorithms, and share congestion state information. Currently, streams destined to different receivers belong to different macroflows. Streams destined to the same receiver MAY belong to different macroflows. When the Congestion Manager is in use, streams that experience identical congestion behavior 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.

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.

We emphasize that all the API functions described in this document are "abstract" calls and that conformant CM implementations may differ in specific implementation details.

2. Introduction

The framework described in this document integrates congestion management across all applications and transport protocols. The CM maintains congestion parameters (available aggregate and per-stream 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. However, the receiving application must provide feedback to the sending application about received packets and losses, and the latter is expected to use the CM API to update CM state. This document does not address networks with reservations or service differentiation.

The CM is an end-system module that enables an ensemble of multiple concurrent streams to perform stable 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-stream 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. When the CM is used, all data transmissions subject to the CM must be done with the explicit consent of the CM via this API to ensure proper congestion behavior.

Systems MAY choose to use CM, and if so they MUST follow this specification.

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 streams 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 streams 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 streams 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, Allman99]), the ensemble of streams 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],

[Padmanabhan98], and [Eggert00] 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, and algorithms, see [Balakrishnan99]; for a description of an implementation and performance results, see [Andersen00].

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 [Allman99]. 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 and the policer at the end hosts, a task that requires router machinery [Floyd99a]. We do not address this issue further in this document.

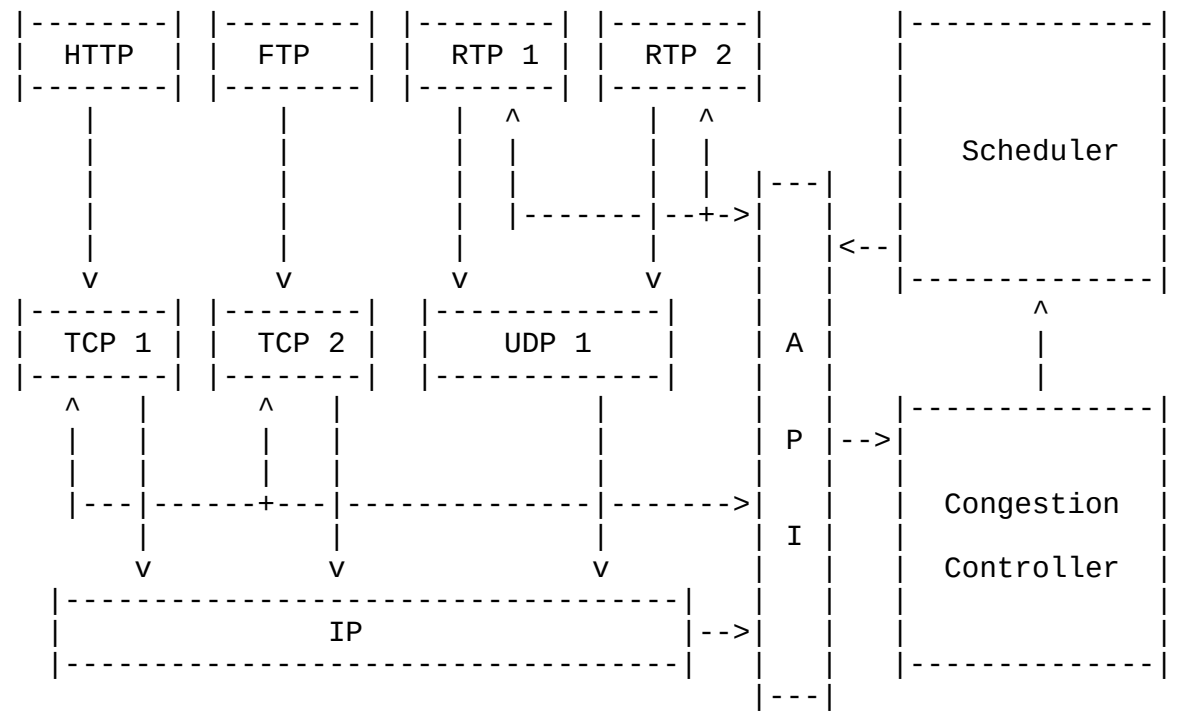

Figure 1

The key components of the CM framework are (i) the API, (ii) the congestion controller, and (iii) the scheduler. The API is (in part) motivated by the requirements 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 streams 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 apportions available bandwidth amongst the different streams within each macroflow and notifies applications when they are permitted to send data. This document focuses on well-behaved applications; a future one will describe the sender-receiver protocol and header formats that will handle applications that do not incorporate their own feedback to the CM.

3. CM API

By convention, the IETF does not treat Application Programming Interfaces as standards track. However, it is considered important to have the CM API and CM algorithm requirements in one coherent document. The following section on the CM API uses the terms MUST,

SHOULD, etc., but the terms are meant to apply within the context of an implementation of the CM API. The section does not apply to congestion control implementations in general, only to those implementations offering the CM API.

Using the CM API, streams 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. We emphasize that these API functions are abstract calls and conformant CM implementations may differ in specific details, as long as equivalent functionality is provided.

When a new stream is created by an application, it passes some information to the CM via the cm_open(stream_info) API call. Currently, stream_info consists of the following information: (i) the source IP address, (ii) the source port, (iii) the destination IP address, (iv) the destination port, and (v) the IP protocol number.

3.1 State maintenance

1. Open: All applications MUST call cm_open(stream_info) before using the CM API. This returns a handle, cm_streamid, for the application to use for all further CM API invocations for that stream. If the returned cm_streamid is -1, then the cm_open() failed and that stream cannot use the CM.

   All other calls to the CM for a stream use the cm_streamid returned from the cm_open() call.

2. Close: When a stream terminates, the application SHOULD invoke cm_close(cm_streamid) to inform the CM about the termination of the stream.

3. Packet size: cm_mtu(cm_streamid) returns the estimated PMTU of the path between sender and receiver. Internally, this information SHOULD be obtained via path MTU discovery [Mogul90]. It MAY be statically configured in the absence of such a mechanism.

3.2 Data transmission

The CM accommodates two types of adaptive senders, enabling applications to dynamically adapt their content based on prevailing network conditions, and supporting ALF-based applications.

1. 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. Thus, 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. The CM must implement a cm_request(i32 cm_streamid) call for streams wishing to send data in this style. After some time, depending on the rate, the CM MUST invoke 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. 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. The CM MAY provide an alternate interface, cm_request(int k). The cmapp_send callback for this request is granted the right to send up to k PMTU sized segments. Section 4.3 discusses the time duration for which the transmission grant is valid, while Section 5.2 describes how these requests are scheduled and callbacks made.

2. Synchronous-style. The above callback-based API accommodates a class of ALF streams 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, there are many streams that are "synchronous" transmitters, which transmit periodically based on their own internal timers (e.g., an audio senders that sends at a constant sampling rate). While CM callbacks could be configured to periodically interrupt such transmitters, the transmit loop of such applications is less affected if they retain their original timer-based loop. In addition, it complicates the CM API to have a stream express the periodicity and granularity of its callbacks. Thus, the CM MUST export 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 stream, srtt is the current smoothed round trip time estimate in microseconds, and rttdev is the smoothed linear deviation in the round-trip time estimate calculated using the same algorithm as in TCP [Paxson00]. The newrate value reports an instantaneous rate calculated, for example, by taking the ratio of cwnd and srtt, and dividing by the fraction of that ratio allocated to the stream.

In response, the stream MUST adapt its packet size or change its timer interval to conform to (i.e., not exceed) the allowed rate. Of course, it may choose not to use all of this rate. Note that the CM is not on the data path of the actual transmission.

To avoid unnecessary cmapp_update() callbacks that the application will only ignore, the CM MUST provide a cm_thresh(float rate_downthresh, float rate_upthresh, float rtt_downthresh, float rtt_upthresh) function that a stream can use at any stage in its execution. In response, the CM SHOULD invoke the callback only when the rate decreases to less than (rate_downthresh * lastrate) or increases to more than (rate_upthresh * lastrate), where lastrate is the rate last notified to the stream, or when the round-trip time changes correspondingly by the requisite thresholds. 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.

The CM MUST implement a cm_query(i32 cm_streamid, u64* rate, u32* srtt, u32* rttdev) to allow an application to query the current CM state. 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. If the CM does not have valid estimates for the macroflow, it fills in negative values for the rate, srtt, and rttdev.

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() callback API for TCP will become apparent.

The reader will notice that the basic CM API does not provide an interface for buffered congestion-controlled transmissions. This is intentional, since this transmission mode can be implemented using the callback-based primitive. Section 6.1.2 describes how congestion-controlled UDP sockets may be implemented using the CM API.

3.3 Application notification

When a stream receives feedback from receivers, it MUST use cm_update(i32 cm_streamid, u32 nrecd, u32 nlost, u8 lossmode, i32 rtt) to inform the CM about events such as congestion losses,

successful receptions, type of loss (timeout event, Explicit Congestion Notification [Ramakrishnan99], etc.) and round-trip time samples. The nrecd parameter indicates how many bytes were successfully received by the receiver since the last cm_update call, while the nrecd parameter identifies how many bytes were received were lost during the same time period. 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 application. The lossmode parameter provides an indicator of how a loss was detected. A value of CM_NO_FEEDBACK indicates that the application has received no feedback for all its outstanding data, and is reporting this to the CM. For example, a TCP that has experienced a timeout would use this parameter to inform the CM of this. A value of CM_LOSS_FEEDBACK indicates that the application has experienced some loss, which it believes to be due to congestion, but not all outstanding data has been lost. For example, a TCP segment loss detected using duplicate (selective) acknowledgments or other data-driven techniques fits this category. A value of CM_EXPLICIT_CONGESTION indicates that the receiver echoed an explicit congestion notification message. Finally, a value of CM_NO_CONGESTION indicates that no congestion-related loss has occurred. The lossmode parameter MUST be reported as a bit-vector where the bits correspond to CM_NO_FEEDBACK, CM_LOSS_FEEDBACK, CM_EXPLICIT_CONGESTION, and CM_NO_CONGESTION. Note that over links (paths) that experience losses for reasons other than congestion, an application SHOULD inform the CM of losses, with the CM_NO_CONGESTION field set.

cm_notify(i32 cm_streamid, u32 nsent) MUST be called when data is transmitted from the host (e.g., in the IP output routine) to inform the CM that nsent bytes were just transmitted on a given stream. This allows the CM to update its estimate of the number of outstanding bytes for the macroflow and for the stream.

A cmapp_send() grant from the CM to an application is valid only for an expiration time, equal to the larger of the round-trip time and an implementation-dependent threshold communicated as an argument to the cmapp_send() callback function. The application MUST NOT send data based on this callback after this time has expired. Furthermore, if the application decides not to send data after receiving this callback, it SHOULD call cm_notify(stream_info, 0) to allow the CM to permit other streams in the macroflow to transmit data. The CM congestion controller MUST be robust to applications forgetting to invoke cm_notify(stream_info, 0) correctly, or applications that crash or disappear after having made a cm_request() call.

3.4 Querying

If applications wish to learn about per-stream available bandwidth and round-trip time, they can use the CM's cm_query(i32 cm_streamid, i64* rate, i32* srtt, i32* rttdev) call, which fills in the desired quantities. If the CM does not have valid estimates for the macroflow, it fills in negative values for the rate, srtt, and rttdev.

3.5 Sharing granularity

One of the decisions the CM needs to make is the granularity at which a macroflow is constructed, by deciding which streams belong to the same macroflow and share congestion information. The API provides two functions that allow applications to decide which of their streams ought to belong to the same macroflow.

cm_getmacroflow(i32 cm_streamid) returns a unique i32 macroflow identifier. cm_setmacroflow(i32 cm_macroflowid, i32 cm_streamid) sets the macroflow of the stream cm_streamid 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 stream, should one exist.

The default suggested aggregation method is to aggregate by destination IP address; i.e., all streams to the same destination address are aggregated to a single macroflow by default. The cm_getmacroflow() and cm_setmacroflow() calls can then be used to change this as needed. We do note that there are some cases where this may not be optimal, even over best-effort networks. For example, when a group of receivers are behind a NAT device, the sender will see them all as one address. If the hosts behind the NAT are in fact connected over different bottleneck links, some of those hosts could see worse performance than before. It is possible to detect such hosts when using delay and loss estimates, although the specific mechanisms for doing so are beyond the scope of this document.

The objective of this interface is to set up sharing of groups not sharing policy of relative weights of streams in a macroflow. The latter requires the scheduler to provide an interface to set sharing policy. However, because we want to support many different schedulers (each of which may need different information to set policy), we do not specify a complete API to the scheduler (but see

Section 5.2). A later guideline document is expected to describe a few simple schedulers (e.g., weighted round-robin, hierarchical scheduling) and the API they export to provide relative prioritization.

4. CM internals

This section describes the internal components of the CM. It includes a Congestion Controller and a Scheduler, with well-defined, abstract interfaces exported by them.

4.1 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 network path used by the macroflow.

The congestion controller MUST implement a "TCP-friendly" [Mahdavi98] 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 streams.

The congestion control module MUST implement the following abstract interfaces. We emphasize that these are not directly visible to applications; they are within the context of a macroflow, and are different from the CM API functions of Section 4.

- void query(u64 *rate, u32 *srtt, u32 *rttdev): 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 streams 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).

Although these interfaces are not visible to applications, the congestion controller MUST implement these abstract interfaces to provide for modular inter-operability with different separately-developed schedulers.

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.

4.2 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 streams 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 no more than 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(i32 cm_streamid): This call returns the described stream's share of the total bandwidth available to the macroflow. This call combined with the query call of the congestion controller provides the information to satisfy an application's cm_query() request.

- void notify(i32 cm_streamid, 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.

  The Scheduler MAY implement many additional interfaces. As experience with CM schedulers increases, future documents may make additions and/or changes to some parts of the scheduler API.

5. Examples

5.1 Example applications

This section describes three possible uses of the CM API by applications. We describe two asynchronous applications---an implementation of a TCP sender and an implementation of congestion-controlled UDP sockets, and a synchronous application---a streaming audio server. More details of these applications and CM implementation optimizations for efficient operation are described in [Andersen00].

All applications that use the CM MUST incorporate feedback from the receiver. For 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 the CM changing its estimates and calling cmapp_update() of the streams of the macroflow.

The protocols in this section are examples and suggestions for implementation, rather than requirements for any conformant implementation.

5.1.1 TCP

A TCP implementation that uses CM should 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 TCP either connects to or accepts a connection from another host, it performs a cm_open() call to associate the TCP connection with a cm_streamid.

Once a connection is established, the CM is used to control the transmission 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. Loss recovery is still performed by TCP based on fast retransmissions and recovery as well as timeouts. In addition, TCP is also modified to have its own outstanding window (tcp_ownd) estimate. Whenever data segments are sent from its cmapp_send() callback, TCP updates its tcp_ownd value. The ownd variable is also updated after each cm_update() call. TCP also maintains a count of the number of outstanding segments (pkt_cnt). At any time, TCP can calculate the average packet size (avg_pkt_size) as tcp_ownd/pkt_cnt. The avg_pkt_size is used by TCP

to help estimate the amount of outstanding data. Note that this is not needed if the SACK option is used on the connection, since this information is explicitly available.

The TCP output routines are modified as follows:

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

2. When application data is available. The TCP output routines perform all non-congestion checks (Nagle algorithm, receiver-advertised window check, etc). If these checks pass, the output routine queues the data and calls cm_request() for the stream.

3. If incoming data or timers result in a loss being detected, the retransmission is also placed in a queue and cm_request() is called for the stream.

4. The cmapp_send() callback for TCP is set to an output routine. If any retransmission is enqueued, the routine outputs the retransmission. Otherwise, the routine outputs as much new data as the TCP connection state allows. However, the cmapp_send() never sends more than a single segment per call. This routine arranges for the other output computations to be done, such as header and options computations.

The IP output routine on the host calls cm_notify() when the packets are actually sent out. Because it does not know which cm_streamid is responsible for the packet, cm_notify() takes the stream_info as argument (see Section 4 for what the stream_info should contain). Because cm_notify() reports the IP payload size, TCP keeps track of the total header size and incorporates these updates.

The TCP input routines are modified as follows:

1. RTT estimation is done as normal using either timestamps 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 are removed.

3. Upon the arrival of an ack for new data, TCP computes the value of in_flight (the amount of data in flight) as snd_max-ack-1 (i.e., MAX Sequence Sent - Current Ack - 1). TCP then calls cm_update(streamid, tcp_ownd - in_flight, 0, CM_NO_CONGESTION, rtt).

4. Upon the arrival of a duplicate acknowledgement, TCP must check its dupack count (dup_acks) to determine its action. If dup_acks < 3, the TCP does nothing. If dup_acks == 3, TCP assumes that a packet was lost and that at least 3 packets arrived to generate these duplicate acks. Therefore, it calls cm_update(streamid, 4 * avg_pkt_size, 3 * avg_pkt_size, CM_LOSS_FEEDBACK, rtt). The average packet size is used since the acknowledgments do not indicate exactly how much data has reached the other end. Most TCP implementations interpret a duplicate ACK as an indication that a full MSS has reached its destination. Once a new ACK is received, these TCP sender implementations may resynchronize with TCP receiver. The CM API does not provide a mechanism for TCP to pass information from this resynchronization. Therefore, TCP can only infer the arrival of an avg_pkt_size amount of data from each duplicate ack. TCP also enqueues a retransmission of the lost segment and calls cm_request(). If dup_acks > 3, TCP assumes that a packet has reached the other end and caused this ack to be sent. As a result, it calls cm_update(streamid, avg_pkt_size, avg_pkt_size, CM_NO_CONGESTION, rtt).

5. Upon the arrival of a partial acknowledgment (one that does not exceed the highest segment transmitted at the time the loss occurred, as defined in [Floyd99b]), TCP assumes that a packet was lost and that the retransmitted packet has reached the recipient. Therefore, it calls cm_update(streamid, 2 * avg_pkt_size, avg_pkt_size, CM_NO_CONGESTION, rtt). CM_NO_CONGESTION is used since the loss period has already been reported. TCP also enqueues a retransmission of the lost segment and calls cm_request().

When the TCP retransmission timer expires, the sender identifies that a segment has been lost and calls cm_update(streamid, avg_pkt_size, 0, CM_NO_FEEDBACK, 0) to signify that no feedback has been received from the receiver and that one segment is sure to have "left the pipe." TCP also enqueues a retransmission of the lost segment and calls cm_request().

5.1.2 Congestion-controlled UDP

Congestion-controlled UDP is a useful CM application, which we describe in the context of Berkeley sockets [Stevens94]. They provide the same functionality as standard Berkeley UDP sockets, but instead of immediately sending the data from the kernel packet queue to lower layers for transmission, the buffered socket implementation makes calls to the API exported by the CM inside the kernel and gets callbacks from the CM. When a CM UDP socket is created, it is bound to a particular stream. Later, when data is added to the packet queue, cm_request() is called on the stream associated with the

socket. When the CM schedules this stream for transmission, it calls udp_ccappsend() in the UDP module. This function transmits one MTU from the packet queue, and schedules the transmission of any remaining packets. The in-kernel implementation of the CM UDP API should not require any additional data copies and should support all standard UDP options. Modifying existing applications to use congestion-controlled UDP requires the implementation of a new socket option on the socket. To work correctly, the sender must obtain feedback about congestion. This can be done in at least two ways: (i) the UDP receiver application can provide feedback to the sender application, which will inform the CM of network conditions using cm_update(); (ii) the UDP receiver implementation can provide feedback to the sending UDP. Note that this latter alternative requires changes to the receiver's network stack and the sender UDP cannot assume that all receivers support this option without explicit negotiation.

5.1.3 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. If some other streams on that macroflow have already been active, then it gets an initial estimate that is valid; otherwise, it gets negative values, which it ignores. It then chooses an encoding that does not exceed these estimates (or, in the case of an invalid estimate, uses application-specific initial values) 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.

5.2 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_NO_FEEDBACK and to cwnd/2 (with a minimum of 1 MTU) if the loss_mode is CM_LOSS_FEEDBACK or CM_EXPLICIT_CONGESTION. 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.

5.3 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 streams as possible in round robin fashion.

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

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

6. 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 degrade the performance observed by applications. For example, a stream on a host can arbitrarily slow down any other stream on the same macroflow, a form of denial of service.

The more dangerous form of attack occurs when an application gives the CM a low estimate of congestion. This would cause CM to be overly aggressive and allow data to be sent much more quickly than sound congestion control policies would allow.

[Touch97] describes a number of the security problems that arise with congestion information sharing. An additional vulnerability (not covered by [Touch97])) occurs because applications have access through the CM API to control shared state that will affect other applications on the same computer. For instance, a poorly designed, possibly a compromised, or intentionally malicious UDP application could misuse cm_update() to cause starvation and/or too-aggressive behavior of others in the macroflow.

7. References

[Allman99] Allman, M. and Paxson, V., "TCP Congestion Control", RFC 2581, April 1999.

[Andersen00] Balakrishnan, H., System Support for Bandwidth Management and Content Adaptation in Internet Applications, Proc. 4th Symp. on Operating Systems Design and Implementation, San Diego, CA, October 2000. Available from http://nms.lcs.mit.edu/papers/cm-osdi2000.html

[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.

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

[Eggert00] Eggert, L., Heidemann, J., and Touch, J., "Effects of Ensemble TCP," ACM Computer Comm. Review, January 2000.

[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 T. Henderson,"The New Reno Modification to TCP's Fast Recovery Algorithm," RFC 2582, April 1999.

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

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

[Mogul90] Mogul, J. and S. Deering, "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.

[Paxson00] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

[Postel81] Postel, J., Editor, "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[Ramakrishnan99] Ramakrishnan, K. and Floyd, S., "A Proposal to Add Explicit Congestion Notification (ECN) to IP," RFC 2481, January 1999.

[Stevens94] Stevens, W., TCP/IP Illustrated, Volume 1. Addison-Wesley, Reading, MA, 1994.

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

8. Acknowledgments

We thank David Andersen, Deepak Bansal, and Dorothy Curtis for their work on the CM design and implementation. We thank Vern Paxson for his detailed comments, feedback, and patience, and Sally Floyd, Mark Handley, and Steven McCanne for useful feedback on the CM architecture. Allison Mankin and Joe Touch provided several useful comments on previous drafts of this document.

9. Authors' Addresses

Hari Balakrishnan
Laboratory for Computer Science
200 Technology Square
Massachusetts Institute of Technology
Cambridge, MA 02139

EMail: hari@lcs.mit.edu
Web: http://nms.lcs.mit.edu/~hari/

Srinivasan Seshan
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

EMail: srini@cmu.edu
Web: http://www.cs.cmu.edu/~srini/

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.