Congestion Manager

Nick Feamster
M.I.T. Laboratory for Computer Science

6.829 Computer Networks
October 24, 2001
Outline

- Motivation (problem CM solves?)
  - Sharing info on concurrent flows
  - Enable application adaptation
- CM Architecture
- The CM API (and tradeoffs)
- More about the CM Framework...
- Implementation
- Limitations
Motivation

- End-systems supply much functionality
  - Reliability
  - In-order delivery
  - Demultiplexing
  - Message boundaries
  - Connection abstraction
  - Congestion control

Of these, congestion control is the only functionality required by all communications applications!
"Multimedia Transmissions Drive Net Toward Gridlock"
◊ Sara Robinson, NYT, 8/23/99
Today’s End-System Architecture

TCP’s AIMD solves the problem, right?

- Doesn’t enable application adaptation
  - Streaming (e.g., audio, video, etc.)

- Doesn’t handle concurrent flows
  - e.g., WWW
  - any application that would benefit from shared information
Little information is transferred across layers to applications

- Increasing number of non-TCP applications

CM exports a simple adaptation API
Concurrent Flows: Web of Troubles

- Web browsers perform concurrent downloads
  - Simultaneous downloading for embedded images
  - Proxies can multiplex requests
  - Aggressive downloading $\Rightarrow$ Higher Throughput

- But...
  - Why slow start each connection?
  - Loss information is not shared between flows
  - More connections = More bandwidth! (fair?)
  - Concurrent streams are competing, should be cooperating!

*What can we do to ensure fair behavior and yet gain some of the benefits of concurrent downloads?*

CM abstracts all congestion-related information into one place.
CM performs all congestion related tasks for macroflow.
Applications adapt using the CM-exported API.
Frees transport/app protocols from reimplementing CC.
Questions

○ What to send?
  ▶ API

○ When to send?
  ▶ Congestion controller

○ Who should send?
  ▶ Scheduler

○ What’s the network state?
  ▶ Application feedback/CM Probing
  ▶ OR...can avoid modifying receiver stack if applications provide feedback.
CM Architecture

- Application
  - Congestion Controller
  - Scheduler
  - Prober

- Application
  - Hints Dispatch
  - Responder
  - Congestion Detector

SENDER

RECEIVER
CM Architecture

- Separate congestion management from transport.
- Multiple applications and protocols can share congestion info.
- Separate congestion control and scheduling.
Macroflows

- All streams on a "macroflow" share congestion state

- What is a "macroflow"? Streams grouped by:
  - Destination address?
  - Address and port?
  - End host application?

- Let the application group flows into macroflows that share state

- Quickly detecting good macroflows...
The CM API

○ **State Management**
  - `cm_open()` -- returns stream ID
  - `cm_close()` -- closes session
  - `cm_mtu()` -- get path MTU for flow

○ **Data transmission options**
  - Buffered send
  - Request/callback
  - Rate callback

○ **Application Notification**
  - `cm_update()` -- get new rate
  - `cm_notify()` -- tell CM about any losses

○ **Queries: cm_query()**
Different applications, different needs

- Buffered Send
  - Data-driven applications (send a single file and exit)

- Request/Callback
  - TCP (retransmission decision)
  - Asynchronous apps, last minute adaptation...
  - Video streaming apps, last minute decisions, etc.

- Rate Callback
  - Synchronous event-driven apps (rate-clocked)

Buffering reduces application control, limits the application to do "last minute adaptation"...
Application achieves TCP-like behavior, but has control over *what to send*. 
Asynchronous Transmission for Everyone?

- Request API works for asynchronous sources --
  ```
  asyncronous() {
    wait for (event) {
      get_data();
      send();
    }
  }
  ```

- What about synchronous sources? (e.g., audio)
  ```
  do_every_10_ms () {
    get_data();
    send();
  }
  ```
Synchronous Transmission

- Asynchronous callbacks are not appropriate for applications that must transmit at a constant rate (e.g., audio servers).
  - A more appropriate API:
    - Register info on RTT, rate thresholds
    - `cmapp_update(newrate, new_rtt, new_rttdev)`
  - Application adjusts sending interval, packet size, etc.
Congestion Controller

- Obtains feedback about past transmissions
- Adjusts the aggregate transmission rate between sender and receiver
- Decides when a macroflow should send
- Modular: Congestion control algorithms on per-macroflow basis
Example: Layered Video

- Track loss rates and RTT using RTP/RTCP, report to CM
- Callbacks from CM control sending rates
Goal: Smooth transmission rate => constant quality video
Scheduler

- Decides which flow on a macroflow should send
- Hints from application/receiver to prioritize flows
- Plug in other scheduling algorithms...
Feedback

- Required for stable end-to-end congestion control

- Probing Protocol
  - optional, can use application feedback instead

- Application
  - `cm_update()`
  - no changes to receiver stack

- Frequency?
Probing Protocol

- Sender periodically sends out probes

- Receiver responds with
  - Last received sequence number (i.e., this one)
  - SN of last probe received
  - Bytes in between

- Reordering...?
  - Reverse window choices later.

- Lost probes...?
  - Exponential aging
  - Minimum RTT for half-life (why stable?)
<table>
<thead>
<tr>
<th>V</th>
<th>P</th>
<th>RC</th>
<th>Payload Type=RR</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>SSRC of packet sender (Receiver ID)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SSRC_1 (SSRC of first source)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fraction lost</td>
<td>cumulative number of lost packets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>extended highest sequence number received</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Interarrival Jitter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Timestamp Echo (LSR)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Processing Time (DLSR)</td>
<td></td>
</tr>
<tr>
<td>Window Size (kB)</td>
<td>Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADU Sequence Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADU Fragment Length (bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADU Offset (bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CM Implementation

- **App**
  - Stream requests, updates
  - cmapp_send()
  - cmapp_update()

- **libcm**
  - User-level library implements API
  - Control socket for callbacks

- **Kernel API**
  - TCP
  - Congestion Controller
  - Scheduler

- **UDP-CC**

- **IP**
  - cm_notify()

- IP notifies CM about data transfer on output
Related Work: HTTP/TCP Interactions

○ Connection Establishment
  ▶ 3-Way Handshake, Timeouts (what’s the RTO?)
  ▶ "Stop and Reload" ...manual SYN retransmit

○ Persistent Connections
  ▶ Good: Can avoid slow-start, 3-way handshake, etc.
  ▶ Bad: What’s the congestion window?
  ▶ Solutions: pacing, slowly decrease window, etc.

○ Nagle’s Algorithm: limits number of small packets sent
  ▶ Good: Interactive apps (e.g., ssh) send fewer small packets.
  ▶ Bad: HTTP response delayed if not aligned on packet boundaries.
Limitations?

- Buggy/malicious applications
  - Incorrect loss, RTT reports
  - Application "hogging" bandwidth of macroflow

- Aging of congestion information
  - Detriment of low feedback frequency

- Macroflow granularity
  - Current research is addressing this...

- Multicast applications
Conclusion

- The Congestion Manager Architecture:
  - Separates transport protocol from congestion control algorithms
  - Gives application control over what data to send

- Callback-based architecture allows last minute adaptation by adaptation.

- Applications can benefit from sharing information about congestion.

  Buffering reduces application control, limits the application to do "last minute adaptation"...
Congestion Controller

- May want to use something besides TCP’s AIMD
- What applications may be harmed by high oscillations?
- CM allows separation of congestion control algorithms from transport!
Why Sockets?

What about other kernel to user communication:

- **Signals**
  - Conflict with other applications
  - Receiving a signal is expensive

- **System Calls**
  - Requires threading support...

- **Semaphores**
  - Most network apps use sockets instead (??)
Applications can employ congestion control algorithms that are more amenable to the task. ...and can experiment with different types of algorithms with relative ease.