Implementation and Evaluation of a Congestion Manager

David G. Andersen, Deepak Bansal, Dorothy Curtis
Srinivasan Seshan, Hari Balakrishnan

MIT Laboratory for Computer Science
`http://nms.lcs.mit.edu/projects/cm/`

October 2000

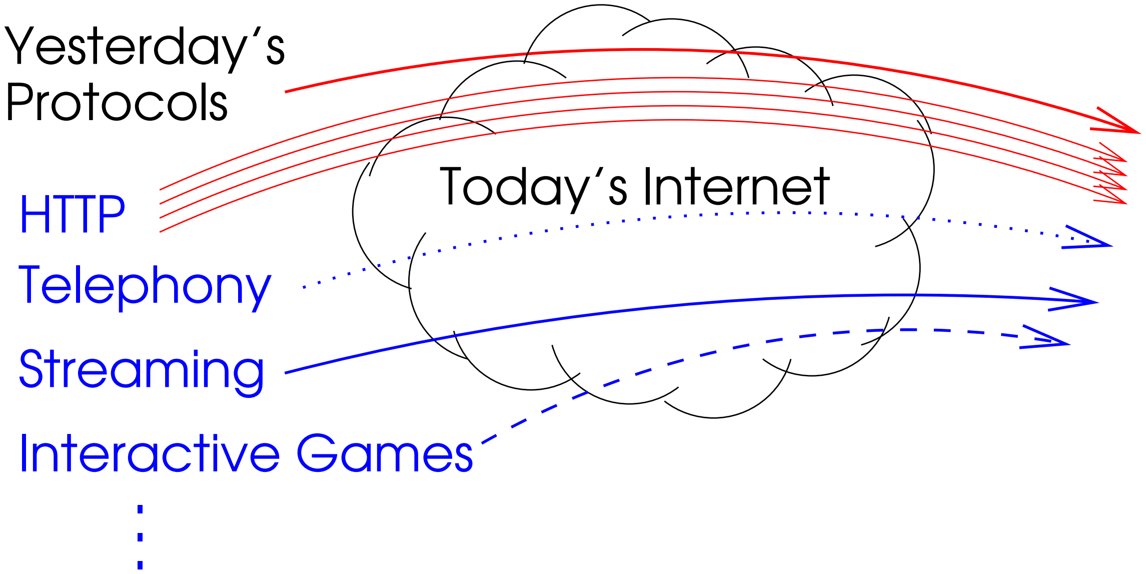
End-to-End Congestion Control

- ✔ Prevents congestion collapse
- ✔ Adapts to network conditions, other streams

Today's solution is TCP

- "AIMD" window-based congestion control
- Provides reliable, in-order bytestream

Problems with current solution



When everything used TCP, it was sufficient

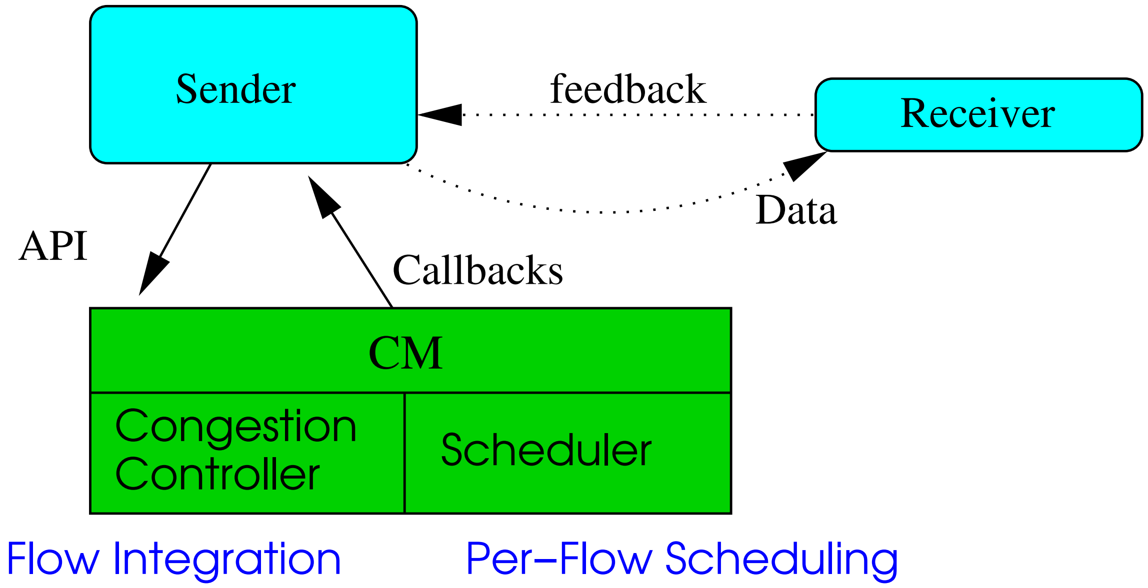
Today's traffic has moved beyond this:

- ✘ Not everything wants TCP
- ✘ Multiple TCP streams are less social

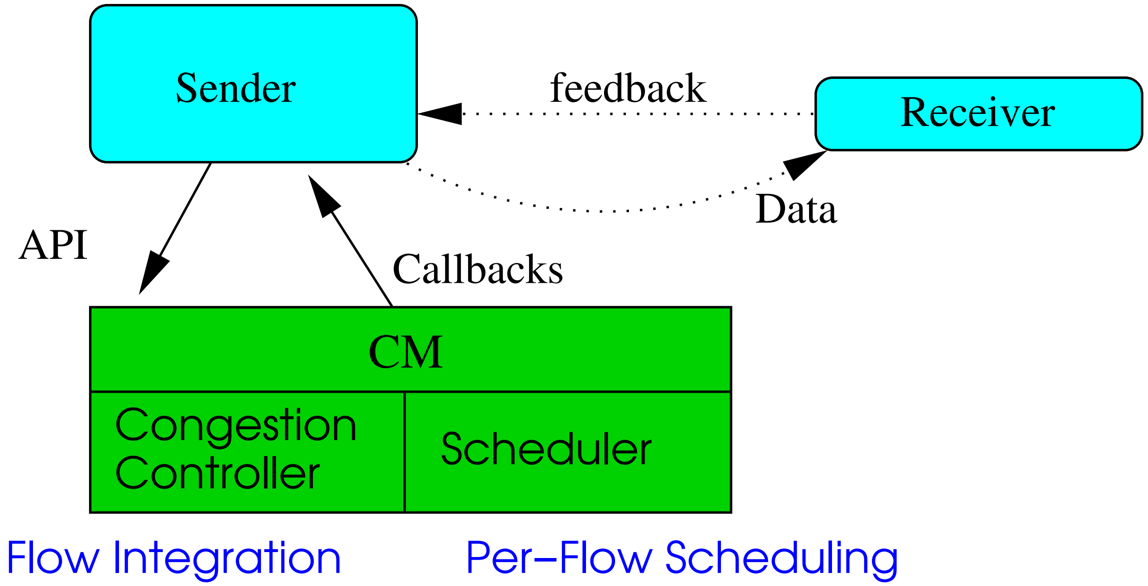
The solution: The Congestion Manager

Sigcomm 99 introduced CM concept, simulation

- Separate congestion management, protocols
- Let multiple protocols share a single CM
- Separate congestion control and scheduling



The Congestion Manager



- Unmodified receiver network stack
- Single API for congestion management
- Transmissions are orchestrated by the CM

A simple API

API overview:

- `open` new connections
- `request` permission to send
- `notify` of transmission
- `update` with successes and losses

Plus callbacks:

- `send` a packet
- `rate` has changed

Application-controlled transmissions

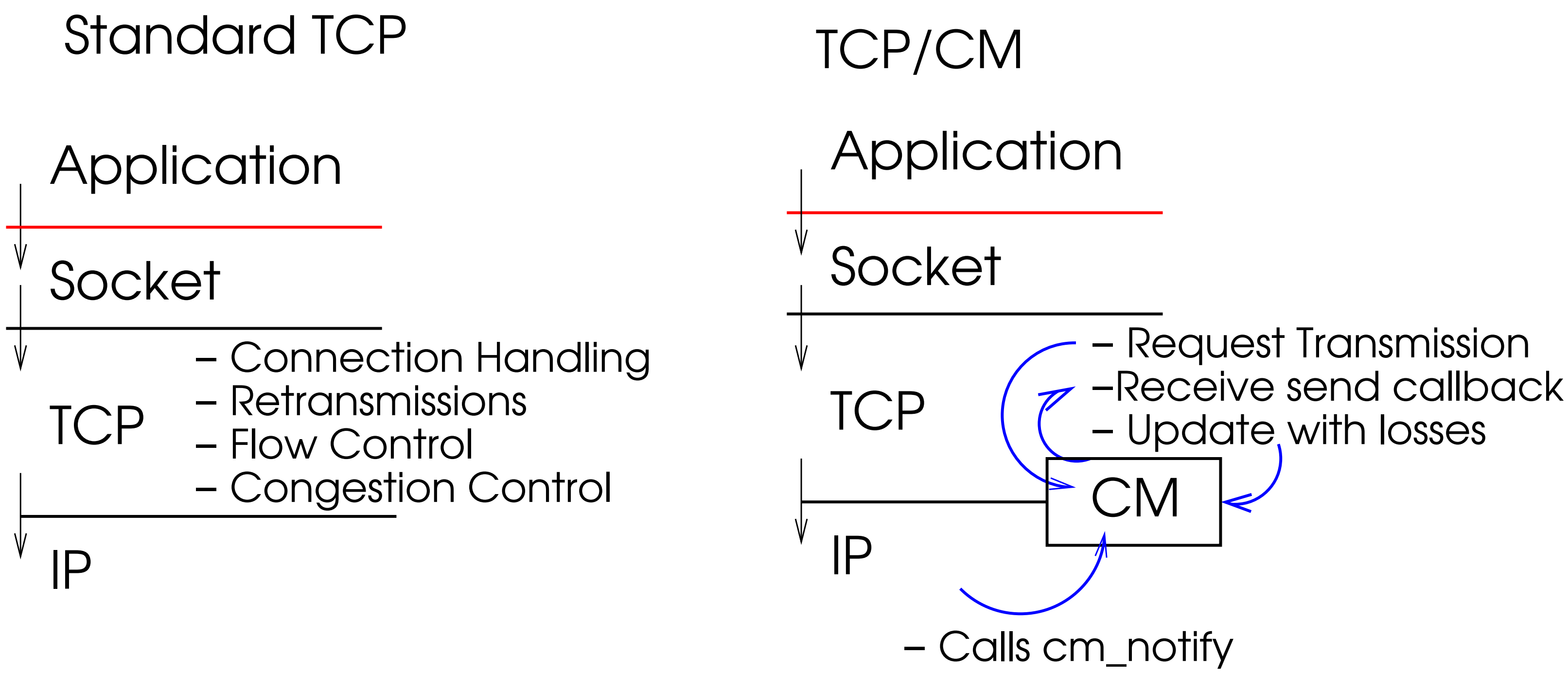
✘ Buffering would reduce application control

- "Last-minute adaptation:"
  - TCP (losses)
  - Streaming media (quality)
- Allows lazy evaluation of work
- Perfect for in-kernel clients like TCP

➔ Used for several very different approaches

- Request/Callback, Rate callbacks, buffered socket

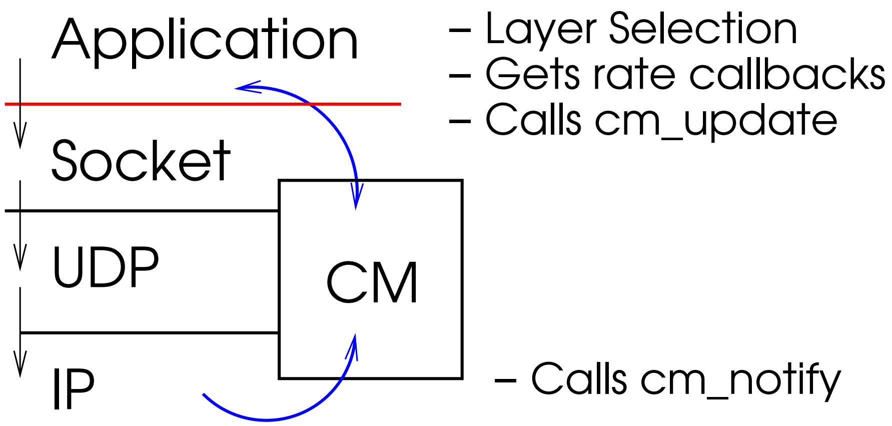
TCP: Request/Callback API



- Gives TCP control over *what* to send

➔ Behavior nearly identical to TCP

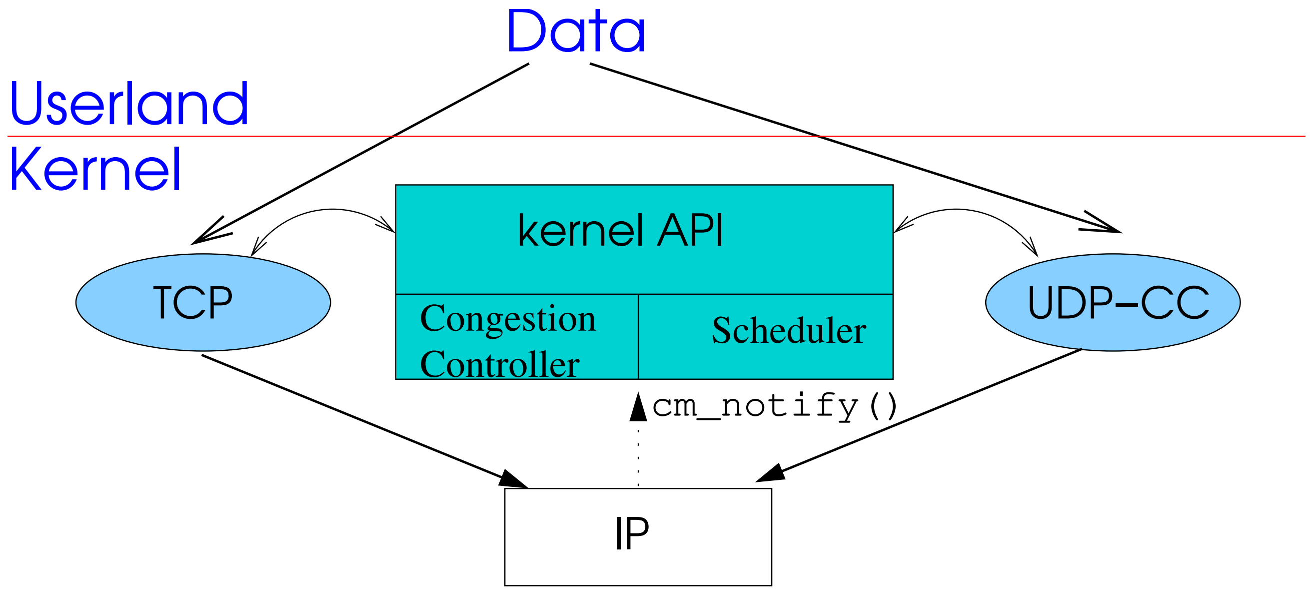
Multi-Rate Encoding Server: Rate Callbacks



- Select encoding by bandwidth
- Fixed number of encodings more/less bandwidth
- Send at specified rate until notified via callback

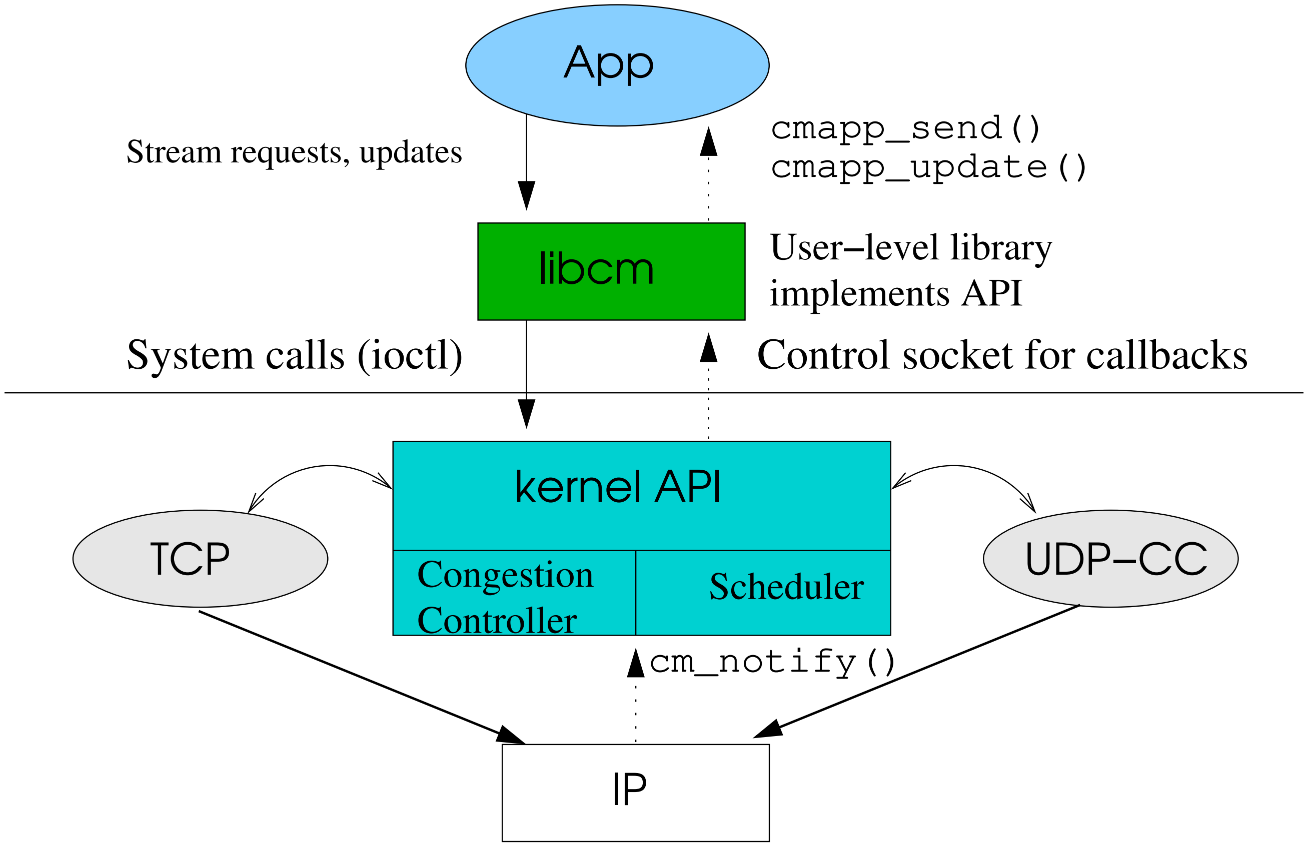
➔ Reduces extraneous callbacks

The CM Implementation



- Function callbacks great in kernel
- `ip_output` informs CM of packet transmission using `cm_notify`

The CM Implementation



Evaluation questions

Impact:

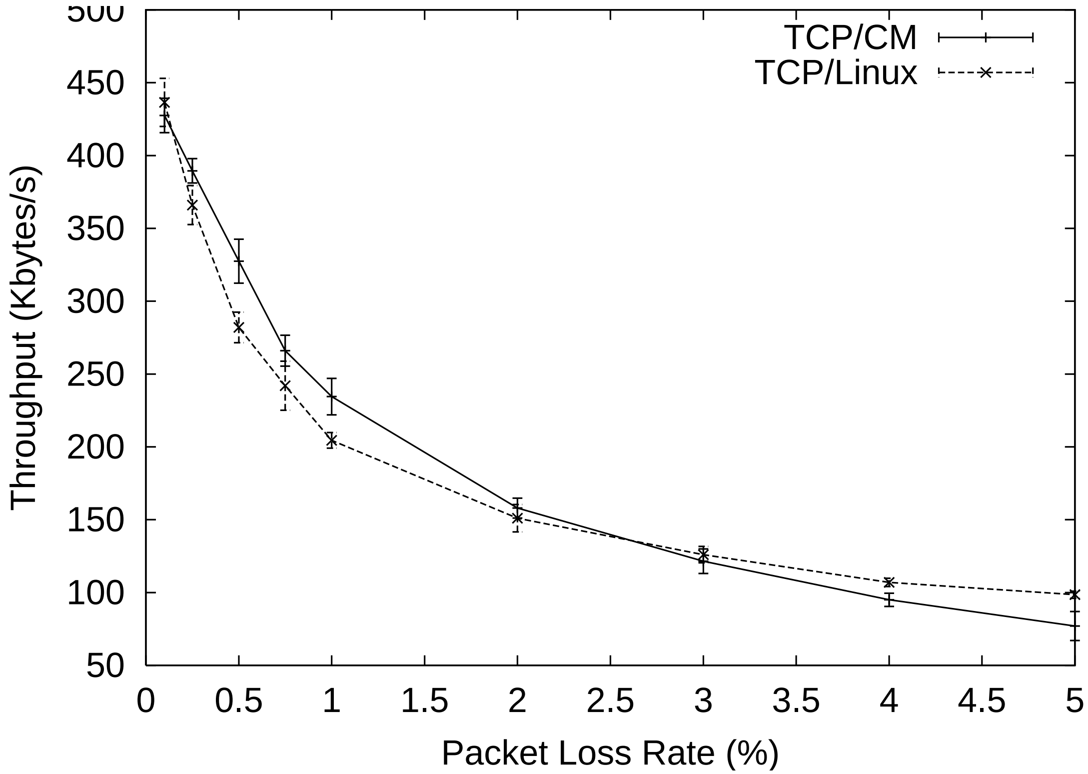
- How does it impact the network?
- How does it impact *my* connections?

Design & Implementation:

- Can the CM be implemented efficiently?
- Can we write new apps using it?
  - How convenient is the adaptation API?

The CM is as friendly as TCP

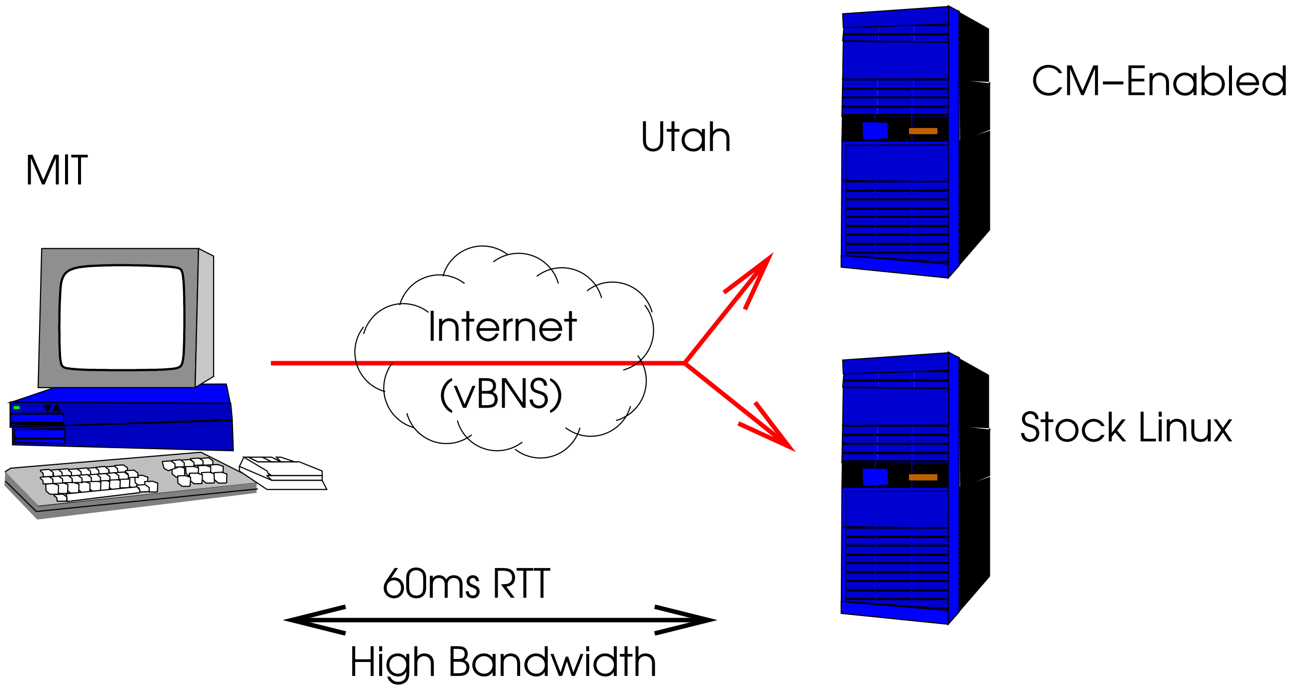
Measured throughput vs. loss rate $(tput \propto \frac{1}{\sqrt{loss}})$



- Dummynet random losses
- 10Mbps 60ms RTT
- 10Mbyte file

→ Global network benefits from the CM

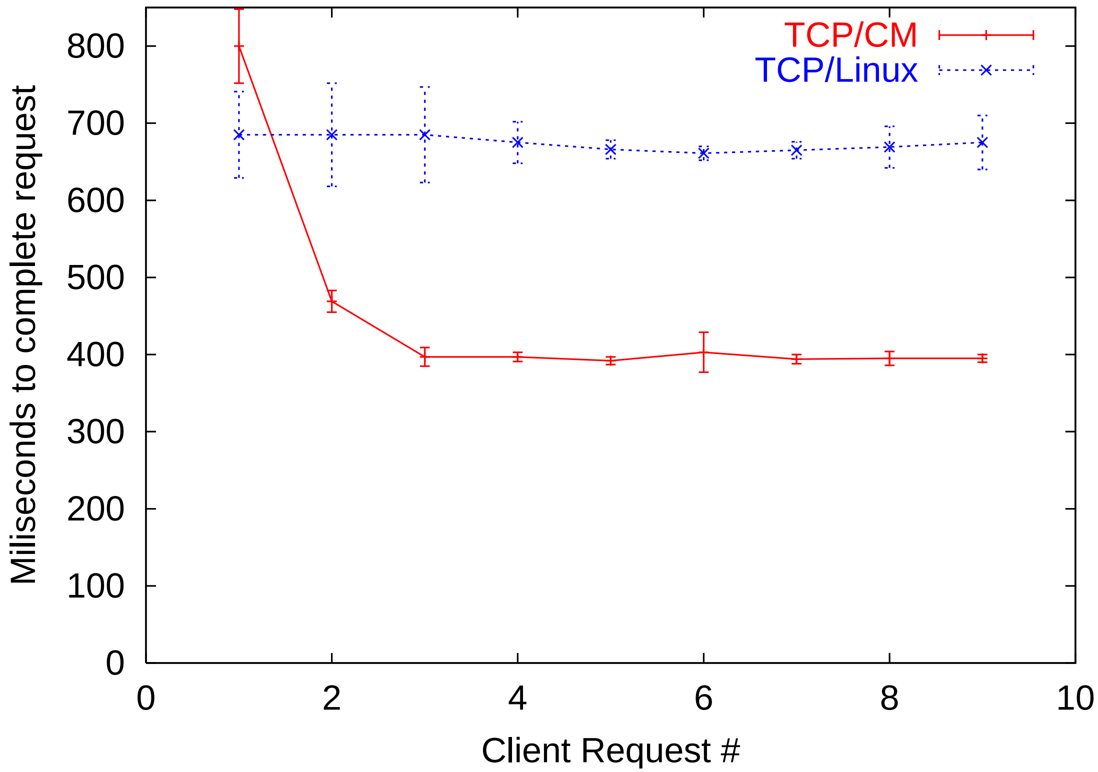
Testing the effect of flow integration



- Web-like: Issue a request every 500ms
  Regardless of completion of earlier requests
- Measure completion times

Integrating congestion control helps

Throughput can benefit from sharing

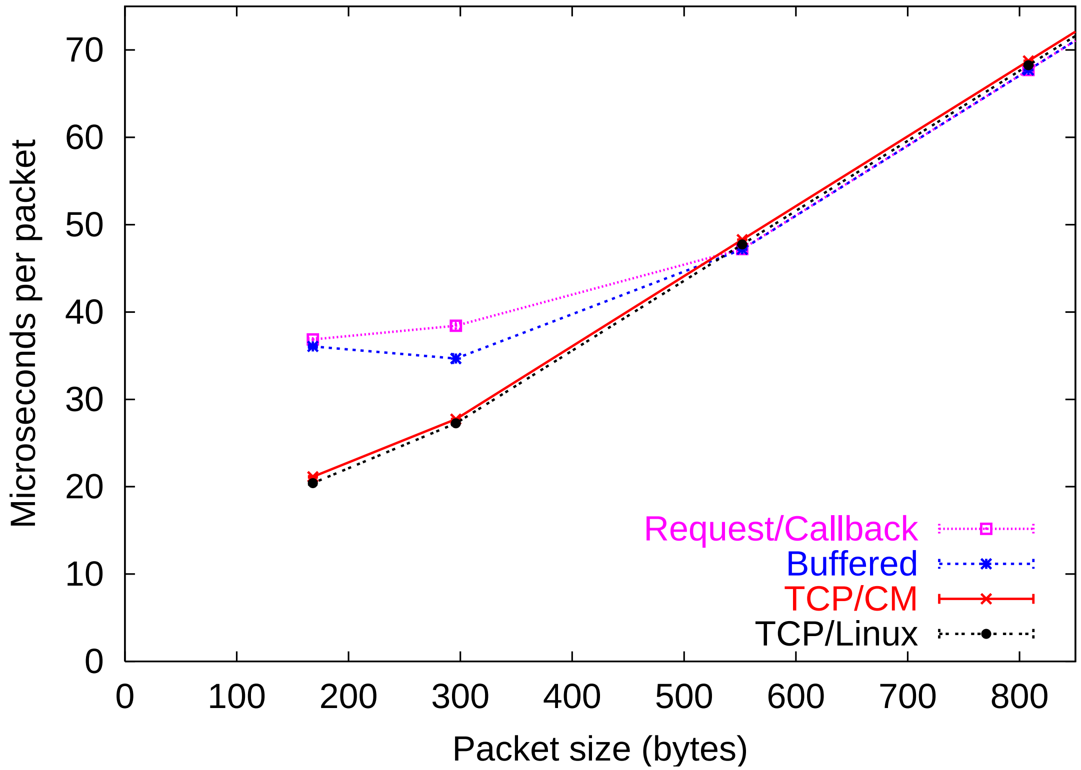


- Web-like workload
- Internet path MIT → Utah
- 500ms request spacing
- 128k file

→ Applications benefit locally from the CM

The CM is efficient

Examined wall clock time / number of packets



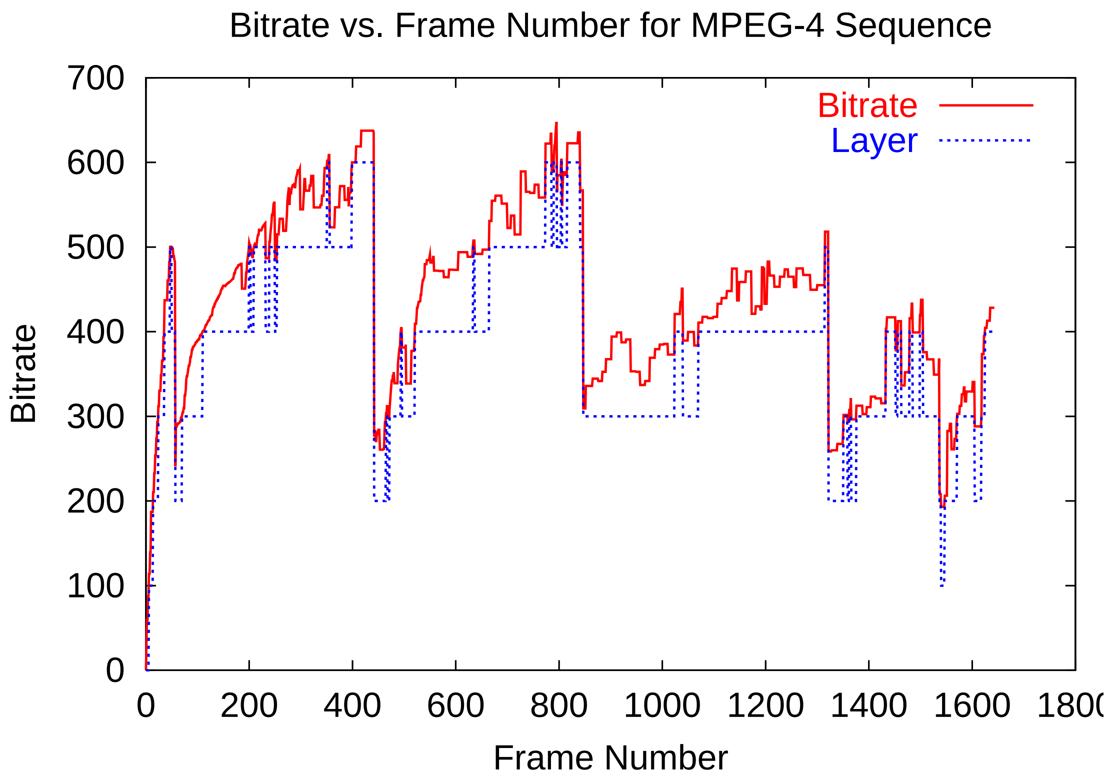
- 100M Ethernet
- 600 Mhz PIIIs
- Quiet network

- < 1-2% difference for in-kernel TCP

- Same throughput with packets > 512 bytes

The CM enables application to adapt

Layered MPEG-4 video sender (unrelated project)



- Layer n at $100{*}n$ Kbps
- Cross trafic from SURGE web workload

→ CM API facilitates adaptive applications

More applications

- Implemented an adaptive `vat`
  - Audioconferencing tool
  - Avoids wasted packets
  - Achieves same audio quality

- Several adaptive test apps in software release

➔ Good platform for research
  - New congestion control algorithms
  - New scheduling algorithms

Some Future Work

- Multiuser machine security
- When to aggregate flows
  - More: Between machines
  - Less: QoS flow isolation, proxies
- Feedback frequency and mechanisms

Conclusions

Separating congestion control is *good*

- Potential for global performance gains
- Potential for local performance gains

Our API makes using the CM *easy*

- Add congestion control to non-TCP apps

Our implementation is *efficient*

- Flexibility without loss of efficiency

Software and more information

`nms.lcs.mit.edu/projects/cm/`

- Freely available (GPL)
  implementation in Linux 2.2
- Internet-Draft of CM spec (ECM WG)
- Lots more!

Persistent HTTP

- Protocol-specific solution
- Browsers still open 2-4 streams
- Coupling of fate between unrelated items
- Contributes to protocol complexity

The CM API

- Register a flow
  - ➔ `flow = cm_open(src, dst)`
- Request permission to send
  - ➔ `cm_request(flow)`
- Wait for a callback
  - ➔ `cmapp_send(flow)`
- Transmit up to 1 MTU
- Tell the CM how it went
  - ➔ `cm_update(flow, sent, recd, rtt)`

The CM API: Requests

```
fid = cm_open(struct sockaddr_in *src,
              struct sockaddr_in *dst)
```

`mtu = cm_mtu(fid)`	Obtain connection's MTU
`cm_request(fid)`	Ask to send up to 1 MTU
`cm_register_send(fn)`	Set send callback
`cm_register_update(fn)`	Set update callback
`cm_thresh(down, up)`	Set update threshholds
`cm_update(fid, nsent, nrecd, mode, rtt)`	
`cm_notify(fid, bytes)`	Notify # bytes sent

The CM API: Callbacks

`cmapp_send(fid)`

- Application may send 1 MTU on this flow

`cmapp_notify(fid, flow_parameters)`

- The network conditions for this flow changed