Data Communications & Networks

Session 10 – Main Theme
Network Congestion
Causes, Effects, Controls, and TCP Applications

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Adapted from course textbook resources
Computer Networking: A Top-Down Approach, 5/E
Copyright 1996-2013
J.F. Kurose and K.W. Ross, All Rights Reserved
Agenda

1. Session Overview
2. Network Congestion Principles
3. Internet Transport Protocols Review
4. TCP Congestion Control
5. Summary and Conclusion
What is the class about?

- Course description and syllabus:
  - [http://www.nyu.edu/classes/jcf/csci-ga.2262-001/](http://www.nyu.edu/classes/jcf/csci-ga.2262-001/)
  - [http://cs.nyu.edu/courses/Fall13/CSCI-GA.2262-001/index.html](http://cs.nyu.edu/courses/Fall13/CSCI-GA.2262-001/index.html)

- Textbooks:
    - James F. Kurose, Keith W. Ross
    - Addison Wesley
Course Overview

- Computer Networks and the Internet
- Application Layer
- Fundamental Data Structures: queues, ring buffers, finite state machines
- Data Encoding and Transmission
- Local Area Networks and Data Link Control
- Wireless Communications
- Packet Switching
- OSI and Internet Protocol Architecture
- Congestion Control and Flow Control Methods
- Internet Protocols (IP, ARP, UDP, TCP)
- Network (packet) Routing Algorithms (OSPF, Distance Vector)
- IP Multicast
- Sockets
Course Approach

- Introduction to Basic Networking Concepts (Network Stack)
- Origins of Naming, Addressing, and Routing (TCP, IP, DNS)
- Physical Communication Layer
- MAC Layer (Ethernet, Bridging)
- Routing Protocols (Link State, Distance Vector)
- Internet Routing (BGP, OSPF, Programmable Routers)
- TCP Basics (Reliable/Unreliable)
- Congestion Control
- QoS, Fair Queuing, and Queuing Theory
- Network Services – Multicast and Unicast
- Extensions to Internet Architecture (NATs, IPv6, Proxies)
- Network Hardware and Software (How to Build Networks, Routers)
- Overlay Networks and Services (How to Implement Network Services)
- Network Firewalls, Network Security, and Enterprise Networks
Network Congestion in Brief

- Session Overview
- Network Congestion Principles
- Internet Transport Protocols Review
- TCP Congestion Control
- Summary & Conclusion
Icons / Metaphors

- Information
- Common Realization
- Knowledge/Competency Pattern
- Governance
- Alignment
- Solution Approach
Subtopics

- What is Congestion?
- Effects of Congestion
- Causes/Costs of Congestion
- Approaches Towards Congestion Control
What is Congestion?

- Congestion occurs when the number of packets being transmitted through the network approaches the packet handling capacity of the network.
- Congestion control aims to keep the number of packets below the level at which performance falls off dramatically.
- Data network is a network of queues (e.g., router buffers).
- Generally 80% utilization is critical.
- Finite queues mean data may be lost (e.g., as router buffers become congested).
- A top-10 problem!
Queues at a Node

Diagram showing a node with arrows labeled 'to other node' and 'to subscriber' pointing out of and into the node, respectively. The node contains sections labeled 'Input buffer' and 'Output buffer.'
Effects of Congestion

- Packets arriving are stored at input buffers
- Routing decision made
- Packet moves to output buffer
- Packets queued for output transmitted as fast as possible
  - Statistical time division multiplexing
- If packets arrive too fast to be routed, or to be output, buffers will fill
- Can discard packets
- Can use flow control
  - Can propagate congestion through network
Interaction of Queues
Causes/Costs of Congestion: Scenario 1

- Two senders, two receivers
- One router, infinite buffers
- No retransmission
- No flow control
- No congestion control

**Host A per connection throughput**

(# of bytes/sec at receiver) as a function of the connection sending rate

**Congestion cost:**

Average delay increases when operating near link capacity

Host A:

\[
\lambda_{in} \quad \lambda_{out}
\]

\[
\frac{C}{2}
\]

\[
\lambda_{in} \quad \frac{C}{2}
\]

C: Router outgoing link capacity

- Large delays when congested
- Maximum achievable throughput
Causes/Costs of Congestion: Scenario 2 (1/2)

- one router, \textit{finite} buffers
- sender retransmits lost packet (i.e. reliable connection assumed)

Performance depends on how retransmission is performed:
(a) Host A only sends a packet when a buffer is free -> no loss (offered load = sending rate)
(b) Sender only retransmits when a packet is known to be lost (timeout large enough…)
   ->\ congestion cost: sender must retransmit to compensate for loss due to buffer overflow
(c) Sender retransmits prematurely a delayed packet that is not lost
   ->\ congestion cost: unneeded retransmissions in the face of large delays
Causes/Costs of Congestion: Scenario 2 (2/2)

- always: $\lambda_{in} = \lambda_{out}$ ($\lambda^{'in} = \lambda_{in}$)
- “perfect” retransmission only when loss: $\lambda^{'in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes $\lambda^{'in}$ larger (than perfect case) for same $\lambda_{out}$

“costs” of congestion:
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
Causes/Costs of Congestion: Scenario 3 (1/2)

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{\text{in}}$ and $\lambda'_{\text{in}}$ increase?

Small increase in sending rate results in a throughput increase
As offered load gets larger and larger, throughput eventually goes to zero
Congestion cost: waste of upstream transmission capacity to packet drop point is wasted
Another “cost” of congestion:

- when packet dropped, any “upstream transmission capacity used for that packet becomes wasted!
Two broad approaches towards congestion control:

**End-end congestion control:**
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP (via indication of timeout of triple duplicate ack)

**Network-assisted congestion control:**
- routers provide feedback to end systems
- single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM ABR)
- explicit rate sender should send at
- indications: choke packets, packet field update
Case Study: ATM ABR Congestion Control

- **ABR: available bit rate:**
  - “elastic service”
  - if sender’s path “underloaded”:
    - sender should use available bandwidth
  - if sender’s path congested:
    - sender throttled to minimum guaranteed rate

- **RM (resource management) cells:**
  - sent by sender, interspersed with data cells (one per 32)
  - bits in RM cell set by switches (“network-assisted”)
    - **NI bit:** no increase in rate (mild congestion)
    - **CI bit:** congestion indication
  - RM cells returned to sender by receiver, with bits intact
- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender’s send rate thus minimum supportable rate on path

- EFCI (Explicit Forward Congestion Indication) bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, destination sets CI bit in returned RM cell
Two Transport Protocols Available

- Transmission Control Protocol (TCP)
  - connection oriented
  - most applications use TCP
  - RFC 793

- User Datagram Protocol (UDP)
  - Connectionless
  - RFC 768
Transport Layer Multiplexing and Demultiplexing

- Extend IP’s delivery svc between two end systems to a delivery svc between two processes running on the end systems

Segment Integrity Checking

TCP Only:

- Reliable data transfer (flow control, seq #s, acknowledgements, and timers)
- Congestion control
Communications endpoint addressed by:

- IP address (32 bit) in IP Header
- Port numbers (16 bit) in TP Header\(^1\)
- Transport protocol (TCP or UDP) in IP Datagram Header

\(^1\) TP => Transport Protocol (UDP or TCP)
<table>
<thead>
<tr>
<th>service</th>
<th>tcp</th>
<th>udp</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>daytime</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>netstat</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>ftp-data</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>ftp</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>telnet</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>smtp</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>time</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>domain</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>finger</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>http</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>pop-2</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>pop</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>sunrpc</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>uucp-path</td>
<td>117</td>
<td>117</td>
</tr>
<tr>
<td>nntp</td>
<td>119</td>
<td>119</td>
</tr>
<tr>
<td>talk</td>
<td>517</td>
<td>517</td>
</tr>
</tbody>
</table>
TCP: Overview
RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
  - one sender, one receiver (no multicasting possible)
- **reliable, in-order byte steam:**
  - no “message boundaries”
- **pipelined:**
  - TCP congestion and flow control set window size
- **send & receive buffers**

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size (app layer data size)
- **connection-oriented:**
  - handshaking (exchange of control msgs) init’s sender, receiver state before data exchange
- **flow controlled:**
  - sender will not overwhelm receiver
TCP Header / Segment Structure

- Data offset: specifies length of TCP header in 32-bit words
- Options field: used when a sender and receiver negotiate the MSS or as a window scaling factor for use in high-speed networks or for timestamping (RFC 854/1323)
TCP Segment Structure

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>head len</td>
<td>not used</td>
</tr>
<tr>
<td>checksum</td>
<td></td>
</tr>
<tr>
<td>ptr urgent data</td>
<td></td>
</tr>
</tbody>
</table>

Options (variable length)

application
data
(variable length)

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept
IP offers best-effort (unreliable) delivery
TCP uses IP
TCP provides completely reliable transfer
How is this possible? How can TCP realize:
- Reliable connection startup?
- Reliable data transmission?
- Graceful connection shutdown?
Reliable Data Transmission

- Positive acknowledgment
  - Receiver returns short message when data arrives
  - Called *acknowledgment*

- Retransmission
  - Sender starts timer whenever message is transmitted
  - If timer expires before acknowledgment arrives, sender *retransmits* message
  - THIS IS NOT A TRIVIAL PROBLEM! – more on this later
**TCP Flow Control**

- **Receiver**
  - Advertises available buffer space
  - Called *window*
  - This is known as a *CREDIT* policy

- **Sender**
  - Can send up to entire window before ACK arrives
  - Each acknowledgment carries new window information
    - Called *window advertisement*
    - Can be zero (called *closed window*)
  - Interpretation: I have received up through $X$, and can take $Y$ more octets
Credit Scheme

- Decouples flow control from ACK
  - May ACK without granting credit and vice versa
- Each octet has sequence number
- Each transport segment has seq number, ack number and window size in header
Use of Header Fields

- When sending, seq number is that of first octet in segment
- ACK includes AN=i, W=j
- All octets through SN=i-1 acknowledged
  - Next expected octet is i
- Permission to send additional window of W=j octets
  - i.e. octets through i+j-1
Credit Allocation

Transport Entity A

...1000 1001 2400 2401...

A may send 1400 octets

...1000 1001 1601 2401...

A shrinks its transmit window with each transmission

...1000 1001 2001 2401...

...1600 1601 2001 2601...

A adjusts its window with each credit

...1600 1601 2600 2601...

A exhausts its credit

...2600 2601 4000 4001...

A receives new credit

Transport Entity B

...1000 1001 2400 2401...

SN = 1001

SN = 1201

SN = 1401

AN = 1601, W = 1000

SN = 1601

SN = 1801

SN = 2001

SN = 2201

SN = 2401

AN = 2601, W = 1400

...1600 1601 2601...

B is prepared to receive 1400 octets, beginning with 1001

...1600 1601 2001 2601...

B acknowledges 3 segments (600 octets), but is only prepared to receive 200 additional octets beyond the original budget (i.e., B will accept octets 1601 through 2600)

...1600 1601 2001 2601...

...2600 2601 4000 4001...

B acknowledges 5 segments (1000 octets) and restores the original amount of credit
TCP Flow Control

**flow control**

sender won’t overrun receiver’s buffers by transmitting too much, too fast

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- RcvWindow field in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received RcvWindow

\[ \text{RcvBuffer} = \text{size of TCP Receive Buffer} \]

\[ \text{RcvWindow} = \text{amount of spare room in Buffer} \]
Seq. #'s:
- byte stream “number” of first byte in segment’s data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor
<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps, everything else already ACKed</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps, one delayed ACK pending</td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>out-of-order segment arrival higher-than-expect seq. # gap detected</td>
<td>send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>arrival of segment that partially or completely fills gap</td>
<td>immediate ACK if segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
**TCP: Retransmission Scenarios**

- **Host A**
  - Seq=92, 8 bytes data
  - ACK=100
  - timeout
  - X
  - loss
  - lost ACK scenario

- **Host B**
  - Seq=100 timeout

- **Host A**
  - Seq=92, 8 bytes data
  - ACK=100
  - timeout

- **Host B**
  - Seq=100 timeout
  - Seq=92, 8 bytes data
  - ACK=120
  - premature timeout, cumulative ACKs
Why Startup / Shutdown Difficult?

- Segments can be
  - Lost
  - Duplicated
  - Delayed
  - Delivered out of order
  - Either side can crash
  - Either side can reboot

- Need to avoid duplicate “shutdown” message from affecting later connection
Recall: TCP sender, receiver establish “connection” before exchanging data segments

initialize TCP variables:
» seq. #s
» buffers, flow control info (e.g. RcvWindow)

client: connection initiator
Socket clientSocket = new Socket("hostname","port number");

server: contacted by client
Socket connectionSocket = welcomeSocket.accept();

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
– specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
– ACKs received SYN
– allocates buffers
– specifies server-> receiver initial seq. #
TCP Connection Management (OPEN)

- Client
- Server
- Opening
- SYN
- SYNACK
- ACK
- Established
- Closed
Closing a connection:

client closes socket:
    `clientSocket.close();`

**Step 1:** client end system sends TCP FIN control segment to server

**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.
**Step 3:** client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.
TCP Connection Management (cont.)

TCP client lifecycle

TCP server lifecycle
The delay required for data to reach a destination and an acknowledgment to return depends on traffic in the internet as well as the distance to the destination. Because it allows multiple application programs to communicate with multiple destinations concurrently, TCP must handle a variety of delays that can change rapidly.

How does TCP handle this.....
Solving Timing Problem

- Keep estimate of round trip time on each connection
- Use current estimate to set retransmission timer
- Known as *adaptive retransmission*
- Key to TCP’s success
Q: how to set TCP timeout value?
- longer than RTT
  - note: RTT will vary
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?
- \textit{SampleRTT}: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- \textit{SampleRTT} will vary, want estimated RTT “smoother”
  - use several recent measurements, not just current \textit{SampleRTT}
TCP Round Trip Time & Timeout

**EstimatedRTT** = \((1-x)\times\text{EstimatedRTT} + x\times\text{SampleRTT}\)

- Exponential weighted moving average (EWMA)
- Influence of given sample decreases exponentially fast
- Typical value of \(x\): 1/8 (RFC 6298)

**Setting the timeout**

- **EstimatedRTT** plus “safety margin”
- Large variation in **EstimatedRTT** -> larger safety margin (\(y\) typically 0.25)

\[
\text{Timeout} = \text{EstimatedRTT} + 4\times\text{Deviation}
\]

\[
\text{Deviation} = (1-y)\times\text{Deviation} + y\times|\text{SampleRTT}-\text{EstimatedRTT}|
\]
Implementation Policy Options

- Send
- Deliver
- Accept
- Retransmit
- Acknowledge
- If no push or close TCP entity transmits at its own convenience (IFF send window allows!)
- Data buffered at transmit buffer
- May construct segment per data batch
- May wait for certain amount of data
Deliver (to application)

- In absence of push, deliver data at own convenience
- May deliver as each in-order segment received
- May buffer data from more than one segment
Accept

- Segments may arrive out of order
- In order
  - Only accept segments in order
  - Discard out of order segments
- In windows
  - Accept all segments within receive window
- TCP maintains queue of segments transmitted but not acknowledged
- TCP will retransmit if not ACKed in given time
  - First only
  - Batch
  - Individual
**Immediate**
- as soon as segment arrives.
- will introduce extra network traffic
- Keeps sender’s pipe open

**Cumulative**
- Wait a bit before sending ACK (called “delayed ACK”)
- Must use timer to insure ACK is sent
- Less network traffic
- May let sender’s pipe fill if not timely!
UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless*:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

**Why is there a UDP?**
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header (8 vs. 20 bytes)
- no congestion control: UDP can blast away as fast as desired
- No retransmission
- Good for real-time apps
  - Require min sending rate and reduced delays and tolerate loss
UDP: more

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive

- Other UDP uses
  - DNS
  - SNMP

- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recover!

UDP segment format:

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Length, in bytes of UDP segment, including header.
UDP Uses

- Inward data collection
- Outward data dissemination
- Request-Response
- Real time application
- Examples:
  - DNS
  - RIP
  - SNMP
1. Session Overview
2. Network Congestion Principles
3. Internet Transport Protocols Review
4. TCP Congestion Control
5. Summary and Conclusion
- TCP Congestion Control
- TCP Fairness
TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission: \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \)
- Roughly,

\[
\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}
\]

- \text{CongWin} is dynamic, function of perceived network congestion

How does sender perceive congestion?
- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (\text{CongWin}) after loss event

three mechanisms:
- AIMD
- slow start
- conservative after timeout events
TCP AIMD

**Multiplicative decrease:** cut CongWin in half after loss event

**Additive increase:** increase CongWin by 1 MSS every RTT in the absence of loss events: *probing*

**Diagram:**
- Congestion window
- 8 Kbytes
- 16 Kbytes
- 24 Kbytes
- Time

**Long-lived TCP connection**
TCP Slow Start

• When connection begins, \textbf{CongWin} = 1 MSS
  • Example: MSS = 500 bytes & RTT = 200 msec
  • initial rate = 20 kbps
• available bandwidth may be $>>$ MSS/RTT
  • desirable to quickly ramp up to respectable rate

• When connection begins, increase rate exponentially fast until first loss event
TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double $\text{CongWin}$ every RTT
  - done by incrementing $\text{CongWin}$ for every ACK received

- **Summary**: initial rate is slow but ramps up exponentially fast
Refinement

- After 3 dup ACKs:
  - *CongWin* is cut in half
  - window then grows linearly

- But after timeout event:
  - *CongWin* instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is “more alarming”
- **Q:** When should the exponential increase switch to linear?
- **A:** When CongWin gets to 1/2 of its value before timeout.

**Implementation:**

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event
Summary: TCP Congestion Control

- When CongWin is below Threshold, sender is in **slow-start** phase, window grows exponentially.
- When CongWin is above Threshold, sender is in **congestion-avoidance** phase, window grows linearly.
- When a **triple duplicate ACK** occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When **timeout** occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.
**Fairness goal:** if $K$ TCP sessions share same bottleneck link of bandwidth $R$, each should have average rate of $R/K$
Why is TCP Fair?

Two competing sessions:

» Additive increase gives slope of 1, as throughout increases
» Multiplicative decrease decreases throughput proportionally

equal bandwidth share

loss: decrease window by factor of 2
congestion avoidance: additive increase

loss: decrease window by factor of 2
congestion avoidance: additive increase
Fairness and UDP
- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections
- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate $R$ supporting 9 connections;
  - new app asks for 1 TCP, gets rate $R/10$
  - new app asks for 11 TCPs, gets $R/2$!
1. Session Overview
2. Network Congestion Principles
3. Internet Transport Protocols Review
4. TCP Congestion Control
5. Summary and Conclusion
Summary

- Session Overview
- Network Congestion Principles
- Internet Transport Protocols Review
- TCP Congestion Control
- Summary & Conclusion
Assignments & Readings

- Readings
  - Chapter 3 – Sections 3.3, 3.5, 3.6, and 3.7
  - RFC 793 – Introduction, Sections 1 and 2
  - RFC 2581
- Assignment #8 previously assigned is due on 12/13/13
Next Session: IP Multicast – Network Security