# User Datagram Protocol (UDP)
# Transmission Control Protocol (TCP)

Matti Siekkinen

28.09.2010

Some material from "Computer Networking: A Top Down Approach" by Jim Kurose, Keith Ross.

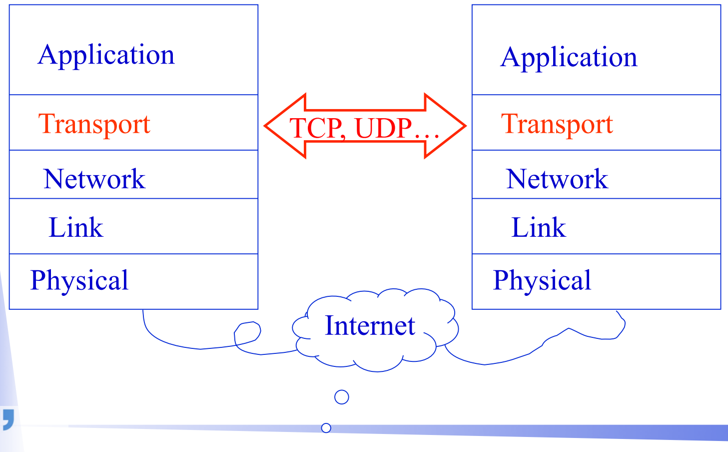
---

## Outline

- ❑ Background
- ❑ UDP
  - Role and Functioning
- ❑ TCP
  - Basics
  - Error control
  - Flow control
  - Congestion control

---

## Transport layer



---

## Transport layer (cont.)

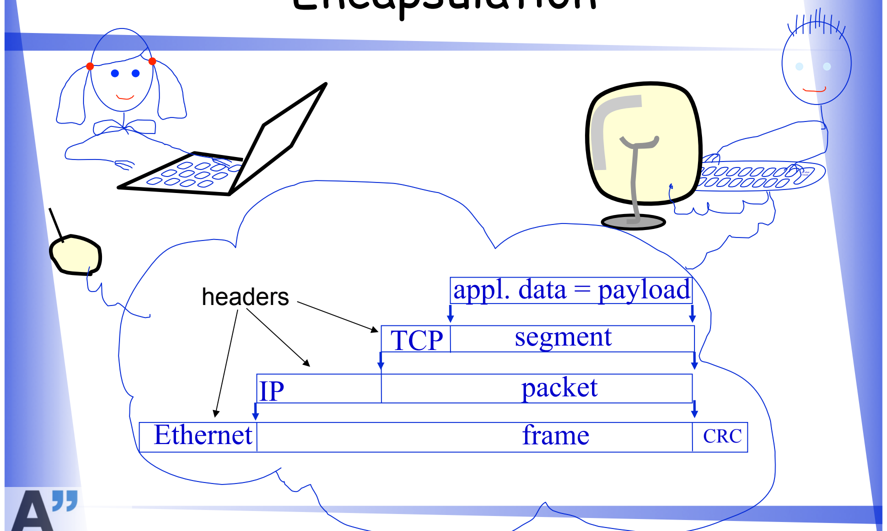
- ❑ Offers end-to-end transport of data for applications
- ❑ Different characteristics
  - Reliable vs. unreliable
  - Forward error correction (FEC) vs. Automatic Repeat-reQuest (ARQ)
  - TCP friendly or not
  - Structured vs. unstructured stream
  - …

# Reliable vs. best effort

- ❑ TCP – reliable transport
  - Guarantees ordered delivery of packets
  - Important for e.g.
    - Signaling messages
    - File transfer
- ❑ UDP – best effort transport
  - No guarantees of packet delivery
  - Non-critical data delivery, e.g. VoIP

---

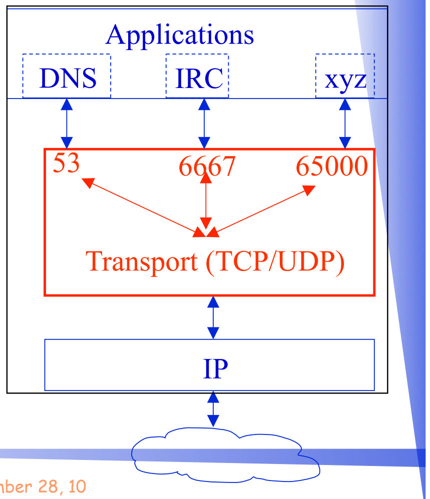
# Encapsulation



---

# Role of ports

- ❑ Well-known port numbers
  - RFC 2780 (&4443)
  - 0-1023
- ❑ Registered port numbers
  - 1024-49151
- ❑ Other port numbers
  - 49152-65535



---

# Checksums

- ❑ For detecting damaged packets
  - Compute at sender, check at receiver
- ❑ Computed from pseudo-header and transport segment
  - Pseudo header includes
    - source and destination IP addresses
    - protocol number
    - TCP/UDP length
    - Slightly different method for IPv4 (RFC 768/793) and IPv6 (RFC 2460)
    - Included for protection against misrouted segments
  - Divide into 16-bit words and compute one's complement of the one's complement sum of all the words

# Part 2: UDP - User Datagram Protocol

---

# User Datagram Protocol (UDP)

- Lightweight protocol
  - Just add port numbering and integrity checking (checksums) to IP
  - No segmentation
- Unreliable connectionless transport service
  - No acknowledgments and no retransmissions
  - Checksum optional in IPv4 and mandatory in IPv6

---

# UDP datagram

| 0 | 16 31 |
|---|---|
| UDP SOURCE PORT | UDP DESTINATION PORT |
| UDP MSG LENGTH | UDP CHECKSUM |
| DATA ... | |

- Source port and checksum are optional
  - Checksum mandatory with IPv6
- Length: header and data in bytes
- Ports provide application multiplexing within a host (single IP)

---

# Part 3: TCP – Transmission Control Protocol

# Outline

- ❑ TCP general overview
- ❑ TCP-header
- ❑ Connection management
- ❑ Error control
- ❑ Flow control
- ❑ Congestion control

---

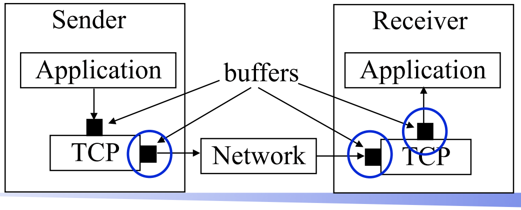
# TCP properties

- ❑ End-to-end
- ❑ Connection oriented
  - State maintained at both ends
  - Identified by a four-tuple
    - Formed by the two end point's IP address and TCP port number
- ❑ Reliable
  - Try to guarantee in order delivery of each packet
  - Buffered transfer
- ❑ Full Duplex
  - Data transfer simultaneously in both directions

---

# TCP properties

- ❑ Three main functionalities for active connection
  1. Error control
     - Deal with the best effort unreliable network
  2. Flow control
     - Do not overload the receiving application
  3. Congestion control
     - Do not overload the network itself



---

# TCP-header (RFC 793)

```
 0                   10                  20                    31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Source port                |       Destination port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Acknowledgment number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| hdr   |           |U|A|P|R|S|F|                               |
| length| Varattu   |R|C|S|S|Y|I| Advertized receiver window    |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Checksum                  |         Urgent-pointer      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

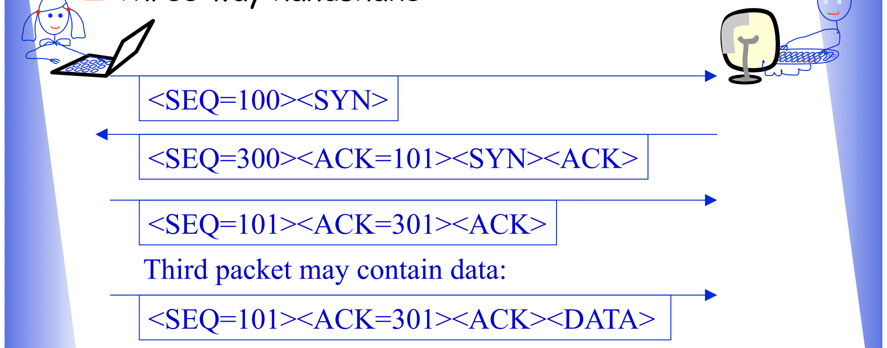
# TCP options

- ❑ 3 = window scaling
- ❑ 8,10 = Timestamp and echo of previous timestamp
  - Improve accuracy of RTT computation
  - Protect against wrapped sequence numbers
- ❑ 2 = Maximum Segment Size (MSS)
  - Negotiated while establishing connection
  - Try to avoid fragmentation
- ❑ 1 = No-operation
  - Sometimes between options, align option fields
- ❑ 0 = End of options

---

# Connection establishment

- ❑ Three-way handshake



<SEQ=100><SYN>

<SEQ=300><ACK=101><SYN><ACK>

<SEQ=101><ACK=301><ACK>

Third packet may contain data:

<SEQ=101><ACK=301><ACK><DATA>

---

# Terminating connection

- ❑ Modified three-way handshake
- ❑ If other end has no more data to send, can be terminated one way:
  - Send a packet with FIN flag set
  - Recipient acks the FIN packet
- ❑ After done with the data transfer to the other direction
  - FIN packet and ack to the inverse direction

---

# Outline

- ❑ TCP general overview
- ❑ TCP-header
- ❑ Connection management
- → ❑ Error control
- ❑ Flow control
- ❑ Congestion control

# Error control

- Mechanisms to detect and recover from lost packets
- Sequence numbers
  - Used in acknowledgments
  - Identify the packets that are acknowledged
- Positive acknowledgments (ARQ)
- Error detection and correction
  - Timers
  - Checksums
- Retransmissions

---

# Cumulative Acknowledgments

- Acknowledge only the next expected packet in sequence
  - E.g. received 1,2,3,4,6 -> ACK 5
- Advantages
  - Single ACK for multiple packets
    - Delayed ACKs scheme = one ACK for 2*MSS data
  - Lost ACK does not necessarily trigger retransmission
- Drawback
  - Cannot tell if lost only first or all of a train of packets
  - => Selective ACK

---

# Selective Acknowledgments (SACK)

- RFC 2018
- Helps recovery when multiple packets are lost
- Receiver reports which segments were lost using TCP SACK (Selective Acknowledgment) options
- Sender can retransmit several packets per RTT

---

# Retransmission timeout (RTO)

- RTO associated to each transmitted packet
- Retransmit packet if no ACK is received before RTO has elapsed
- Adjusting RTO (original algorithm):
  - $RTT = (\alpha*oldRTT)+((1-\alpha)*newRTTsample)$ (recommeded $\alpha=0,9$)
  - RTO: $\beta*RTT$, $\beta>1$ (recommended $\beta=2$)
- Problem?
  - Does not take into account large variation in RTT

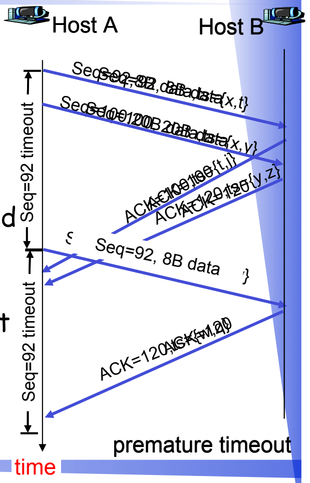
# Modified algorithm

- Initialize: RTO = 3
- Two variables: SRTT (smoothed round-trip time) and RTTVAR (round-trip time variation)
  - First measurement R:
    - SRTT = R
    - RTTVAR = R/2
  - For subsequent measurement R:
    - RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - R|
    - SRTT = (1 - alpha) * SRTT + alpha * R
    - Use alpha=1/8, beta=1/4
- RTO = SRTT + 4*RTTVAR
- If computed RTO < 1s -> round it up to 1s

---

# Karn's algorithm

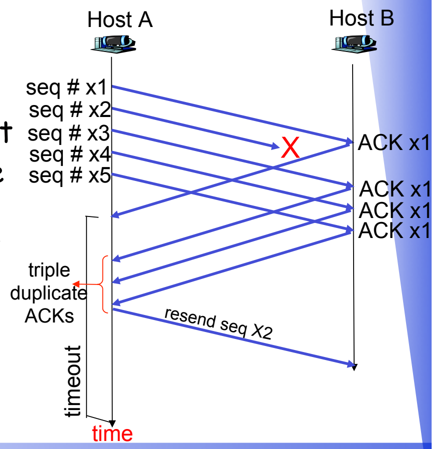
- Receiving ACK for retransmitted packet
  - Is the ACK for original packet or retransmission??
  - No way to know…
  - ➔Do not update RTO for retransmitted packets
- Timer backoff also needed
  - At timeout: new_timeout = 2*timeout (exponential backoff)
- TCP timestamps can also help disambiguate ACKs



---

# Fast Retransmit

- Introduced by Van Jacobson 1988
- TCP ACKs the next expected missing packet
- Duplicate ACKs indicate lost packet(s)
- Do not wait for timeout but retransmit after 3 duplicate ACKs
  - Wait for reordered packets, don't do go-back-n



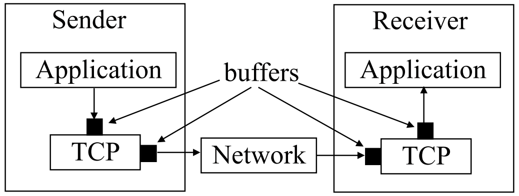
---

# Outline

- TCP general overview
- TCP-header
- Connection management
- Error control
- → Flow control
- Congestion control

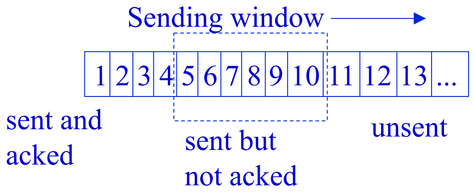
# Flow control

- ❑ Goal: do not overflow the receiving application
- ❑ Window based mechanism to limit transmission rate
- ❑ Receiver advertised window



---

# Sliding Window



- ❑ Multiple packets simultaneously "in flight", i.e. outstanding
  - Improve efficiency
- ❑ Buffer sent unacked packets

---

# Receiver advertised window

- ❑ Receiver advertises the maximum window size the sender is allowed to use
- ❑ Enables receiver TCP to signal sending TCP to backoff
  - Receiving application not consuming received data fast enough
- ❑ Value is included in each ACK
  - Can change dynamically

---

# Silly Window Syndrome

- ❑ Problem in worst case:
  - Receiver buffer between TCP and application fills up
  - Receiving application read a single byte -> TCP advertises a receiver window of size one
  - Sender transmits a single byte
- ❑ Lot of overhead due to packet headers

# Avoiding Silly Window Syndrome

- ❑ Window update only with significant size
  - At least MSS worth of data or
  - Half of its buffer
- ❑ Analogy at sender side
  - Application gives small chunks of data to TCP -> send small packets
  - Nagle's algorithm: Delay sending data until have MSS worth of it
    - Does not work for all applications, e.g. delay sensitive apps
    - Need also mechanism to tell TCP to transmit immediately -> Push flag

---

# Large Receiver Windows

- ❑ Receiver window hdr field size is 16 bits
  - => max size is about 65KBytes
- ❑ Example: 10Mbit/s path from Europe to US west coast
  - 0.15s * 10^7/8 ≈ 190KBytes

  bandwidth

  delay=RTT
  - 16 bits not enough!
- ❑ Use Window Scaling option
  - Both ends set a factor during handshake (SYN segments)
  - Multiply window field value with this factor

---

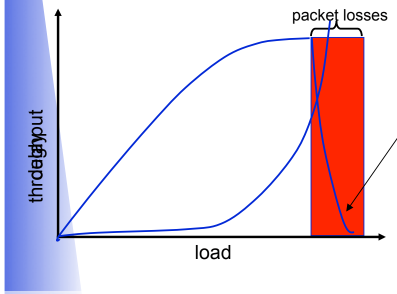
# Outline

- ❑ TCP general overview
- ❑ TCP-header
- ❑ Connection management
- ❑ Error control
- ❑ Flow control
- → ❑ Congestion control
  - Background and motivation
  - Basic TCP congestion control
  - Fairness
  - Other TCP versions and recent developments
- ❑ Conclusions

---

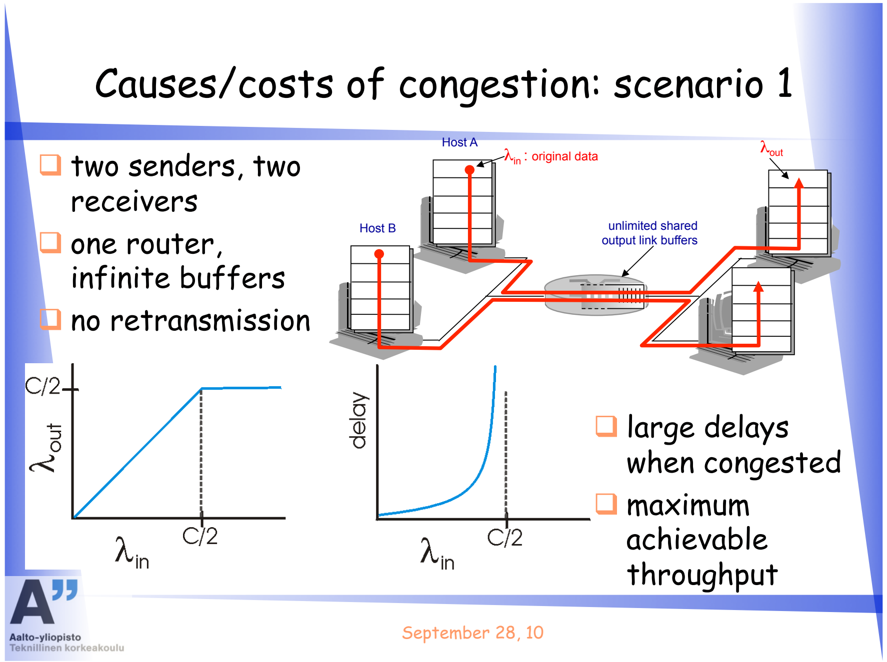
# Why we need congestion control

- ❑ Flow control in TCP prevents overwhelming the receiving application
- ❑ Problem: Multiple TCP senders sharing a link can still overwhelm it



Congestion collapse due to:

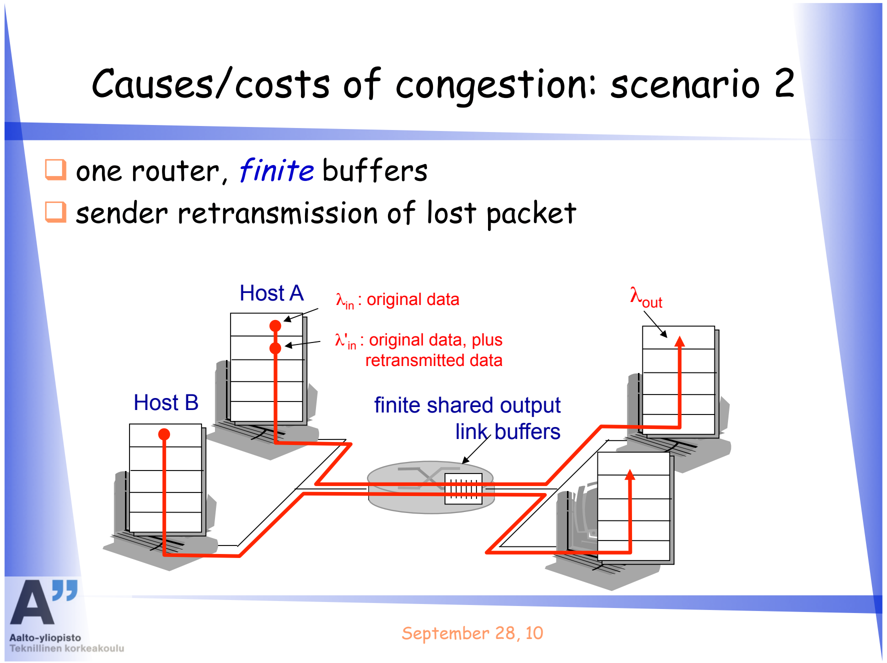
- ❑ Retransmitting lost packets
  - Further increases the load
- ❑ Spurious retransmissions of packets still in flight
  - Unnecessary retransmissions lead to even more load!
  - Like pouring gasoline on a fire



## Causes/costs of congestion: scenario 1

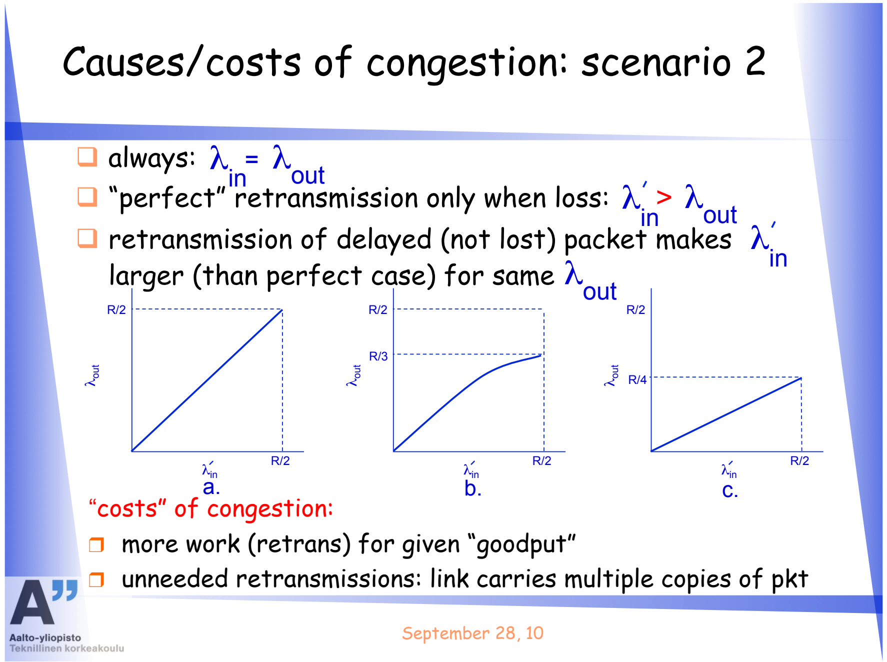
- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput



## Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet

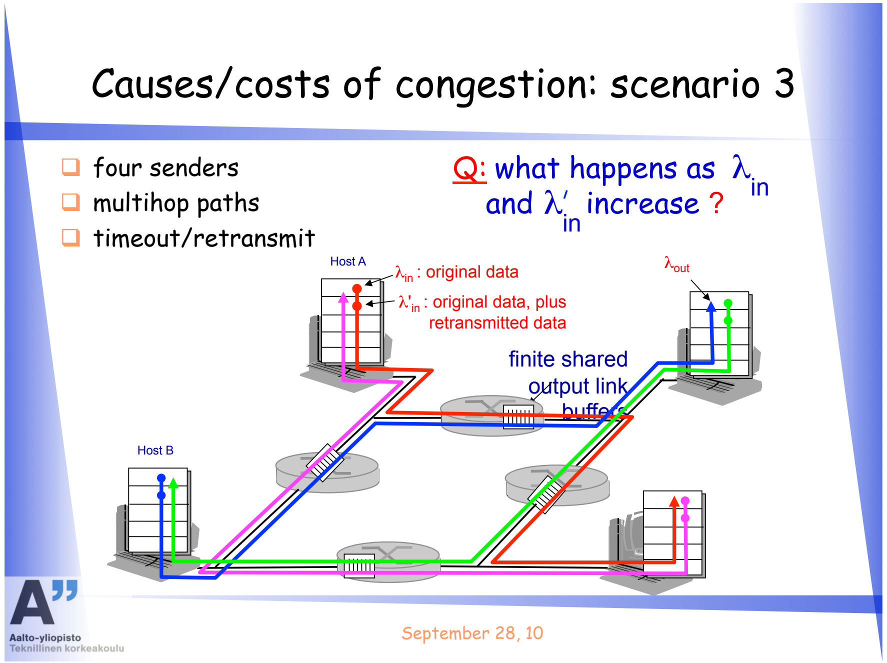


## Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$
- "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

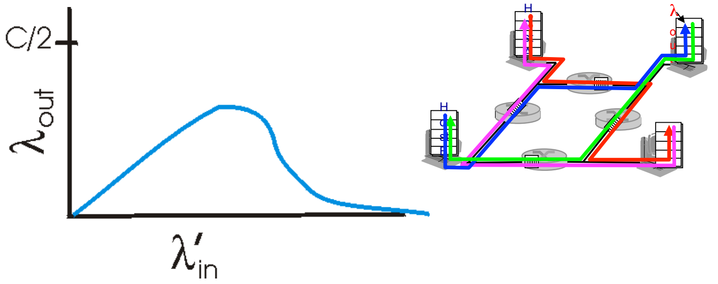


## Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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

# Causes/costs of congestion: scenario 3



another "cost" of congestion:

- when packet dropped, any upstream transmission capacity used for that packet was wasted!

---

# Approaches towards congestion control

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

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

---

# Explicit Congestion Notification (ECN)

- Routers flag packets upon congestion
  - Active queue management
- Sender consequently adjusts sending rate
- Supported by routers but not widely used
  - Fear of software bugs
  - Running with default configurations
- Most OSs (Win7, Ubuntu, Fedora) ship with ECN disabled
  - Tuning for bugs (e.g. popular Cisco PIX firewall)

---

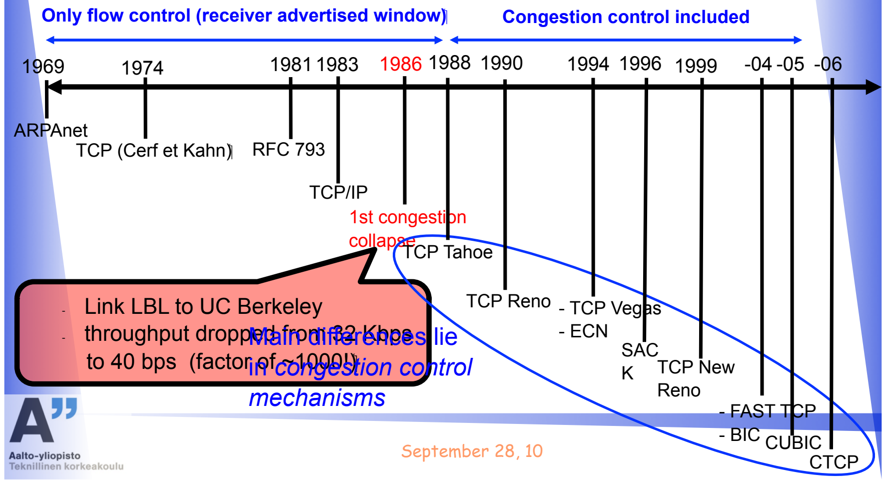
# TCP Congestion control

- Principle:
  - Continuously throttle TCP sender's transmission rate
  - Probe the network by increasing the rate when all is fine
  - Decrease rate when signs of congestion (e.g. packet loss)
- How?
  - Introduce *congestion window* (cwnd):

    *#outstanding bytes = min(cwnd, rwnd)*

    **flow control**
  - Adjust cwnd size to control the transmission rate
    - Adjustment strategy depends on TCP version

# Glimpse into the past



- Link LBL to UC Berkeley
- throughput dropped from 32 Kbps to 40 bps (factor of ~1000!)

*Main differences lie in congestion control mechanisms*

---

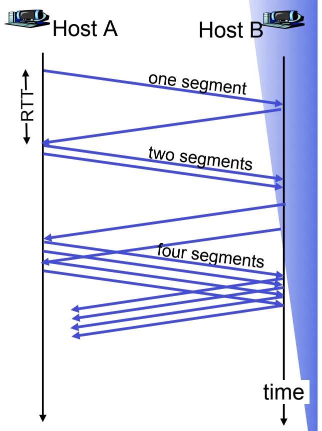
# TCP Tahoe

- ❑ 1988 Van Jacobson
- ❑ The basis for TCP congestion control
- ❑ Lost packets are sign of congestion
  - Detected with *timeouts:* no ACK received in time
- ❑ Two modes:
  - Slow Start
  - Congestion Avoidance
- ❑ New retransmission timeout (RTO) calculation
  - Incorporates variance in RTT samples
  - Timeout really means a lost packet (=congestion)
- ❑ Fast Retransmit

---

# Slow Start (SS)

- ❑ On each ACK for new data, increase cwnd by 1 packet
  - Exponential increase in the size of cwnd
  - Ramp up a new TCP connection fast (not slow!)
    - Kind of a misnomer…
- ❑ In two cases:
  - Beginning of connection
  - After a timeout



---

# Congestion Avoidance (CA)

- ❑ Approach the rate limit of the network more conservatively
- ❑ Easy to drive the net into saturation but hard for the net to recover
- ❑ Increase cwnd by 1 for cwnd worth of ACKs (i.e. per RTT)

# Combining SS and CA

- ❑ Introduce Slow start threshold (ssthresh)
- ❑ On timeout:
  - ▪ ssthresh = 0.5 x cwnd
  - ▪ cwnd = 1 packet
- ❑ On new ACK:
  - ▪ If cwnd < ssthresh: do Slow Start
  - ▪ Else: do Congestion Avoidance

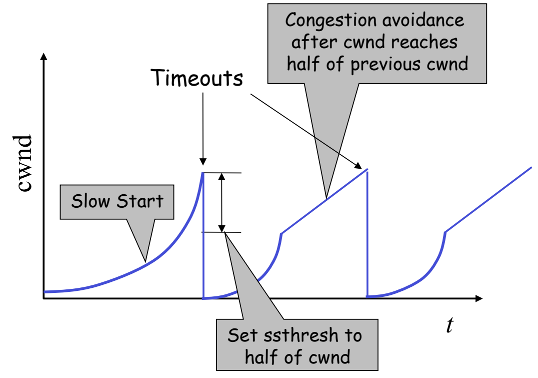
**AIMD**

- ❒ ACKs: increase **cwnd** by 1 MSS per RTT: additive increase
- ❒ loss: cut **cwnd** in half (non-timeout-detected loss ): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

---

# TCP Tahoe: adjusting cwnd



---

# TCP Reno

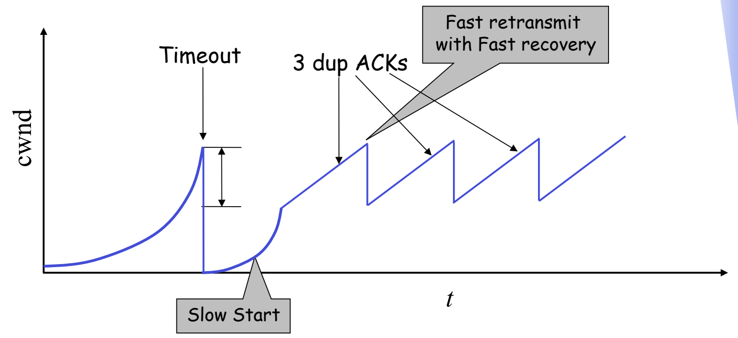
- ❑ Van Jacobson 1990
- ❑ Fast retransmit with Fast recovery
  - ▪ Duplicate ACKs tell sender that packets still go through
  - ▪ Do less aggressive back-off:
    - o ssthresh = 0.5 x cwnd
    - o cwnd = ssthresh + 3 packets
    - o Increment cwnd by one for each additional duplicate ACK
    - o When the next new ACK arrives: cwnd = ssthresh

**Nb of packets that were delivered**

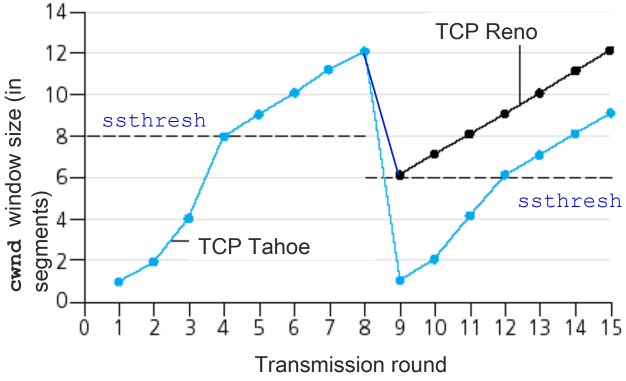
**Fast recovery**

---

# TCP Reno: adjusting cwnd

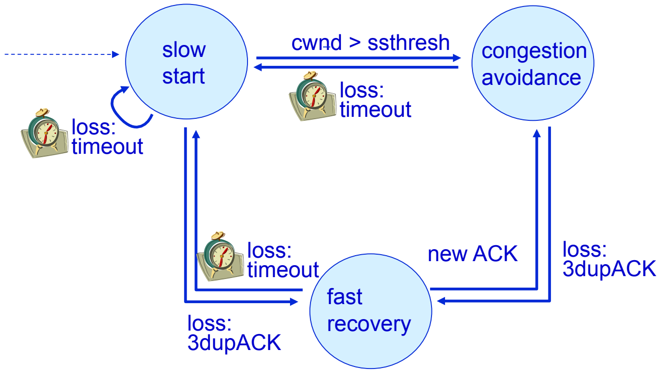


# Tahoe vs. Reno



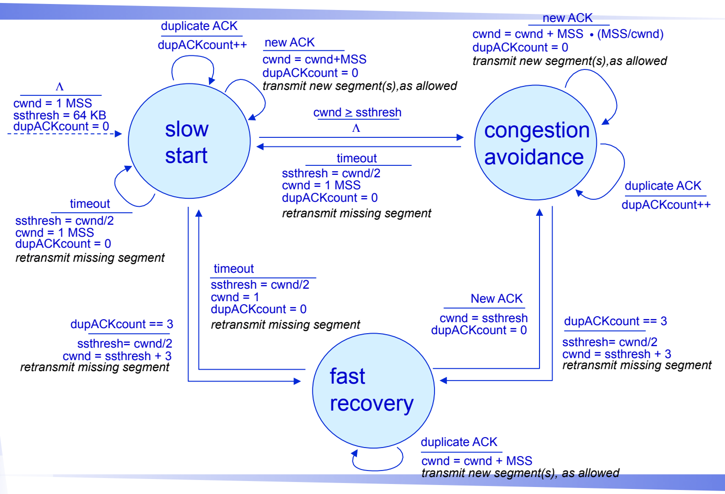
---

# Congestion control FSM



---

# Congestion control FSM: details



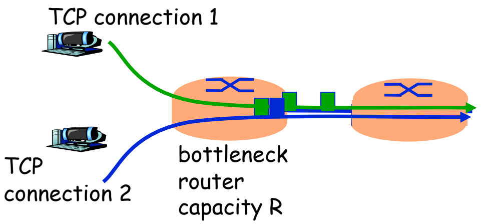
---

# TCP New Reno

- 1999 by Sally Floyd
- Modification to Reno's Fast Recovery phase
- Problem with Reno:
  - Multiple packets lost in a window
  - Sender out of Fast Recovery after retransmission of only one packet
    - ➔ cwnd closed up
    - ➔ no room in cwnd to generate duplicate ACKs for additional Fast Retransmits
    - ➔ eventual timeout
- New Reno continues Fast Recovery until all lost packets from that window are recovered

# TCP Fairness

fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

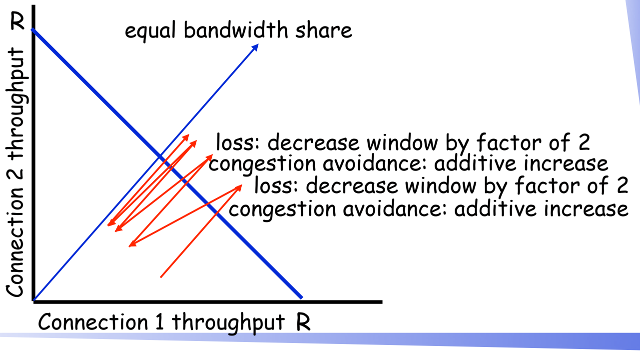


Is TCP fair?

# Why is TCP fair?

Two competing sessions:

- ❑ Additive increase gives slope of 1, as throughput increases
- ❑ multiplicative decrease decreases throughput proportionally



# TCP Fairness Issues (cont.)

**RTT Fairness**

- ❑ What if two connections have different RTTs?
  - "Faster" connection grabs larger share
- ❑ Reno's (AIMD) fairness is RTT biased

**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 !

# 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

# Other TCP versions

- ❑ Delay-based congestion control
  - TCP Vegas
- ❑ Wireless networks
  - Take into account random packet loss due to bit errors (not congestion!)
  - E.g. TCP Veno
- ❑ Paths with high *bandwidth*delay*
  - These *"long fat pipes"* require large cwnd to be saturated
  - SS and CA provide too slow response
  - TCP CUBIC
  - Compound TCP (CTCP)

---

# TCP Vegas

- ❑ 1994 by Brakmo et Peterson
- ❑ Issue: Tahoe and Reno RTO clock is very coarse grained
  - "ticks" each 500ms
- ❑ Increasing delay is a sign of congestion
  - Packets start to fill up queues
  - Expected throughput = cwnd / BaseRTT
  - Compare expected to actual throughput
  - Adjust rate accordingly before packets are lost
- ❑ Also some modifications to Slow start and Fast Retransmit
- ❑ Potentially up to 70% better throughput than Reno
- ❑ Fairness with Reno?
  - Reno grabs larger share due to late congestion detection

**minimum of all measured round trip times**

---

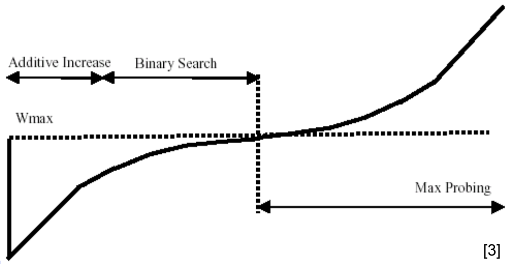
# BIC and CUBIC

- ❑ 2004, 2005 by Xu and Rhee
- ❑ Both for paths with high (*bandwidth* x *delay)*
  - These *"long fat pipes"* lead to large cwnd
  - SS and CA provide too slow response
  - Scale up to tens of Gb/s
- ❑ BIC TCP
  - No AIMD
  - Window growth function is combination of *binary search* and *linear increase*
  - Aim for TCP friendliness and RTT fairness

---

# BIC and CUBIC

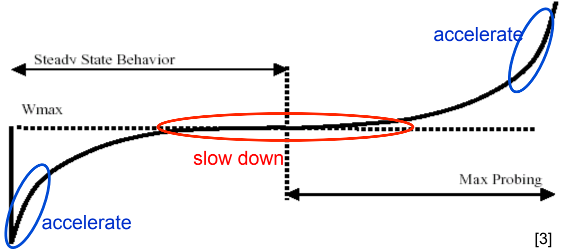
- ❑ BIC window growth function



[3]

# BIC and CUBIC (cont.)

- CUBIC TCP
  - Enhanced version of BIC
  - Simplifies BIC window control using a *cubic function*
  - Improves its TCP friendliness & RTT fairness



[3]

$$W_{cubic} = C(t-K)^3 + W_{\max} \qquad K = \sqrt[3]{W_{\max}\beta/C}$$

---

# Compound TCP (CTCP)

- From Microsoft research, 2006
- Tackles same problems as BIC and CUBIC
  - High speed and long distance networks
  - RTT fairness, TCP friendliness
- Loss-based vs. delay-based approaches
  - Loss-based (e.g. HSTCP, BIC...) too aggressive
  - Delay-based (e.g. Vegas) too timid
- Compound approach
  - Use delay metric to sense the network congestion
  - Adaptively adjust aggressiveness based on network congestion level
  - Loss-based component: *cwnd* (standard TCP Reno)
  - Scalable delay-based component: *dwnd*
  - TCP sending window is *Win = cwnd + dwnd*

---

# Deployment

- Windows
  - Server 2008 uses Compound TCP (CTCP) by default
  - Vista, XP support CTCP, New Reno by default
- Linux
  - TCP BIC default in kernels 2.6.8 through 2.6.18
  - TCP CUBIC since 2.6.19

---

# Conclusions

- Transport layer
  - End-to-end transport of data for applications
  - Application multiplexing through port numbers
  - Reliable (TCP) vs. unreliable (UDP)
- UDP
  - Unreliable, no state
  - Optionally integrity checking
- TCP
  - Connection management
  - Error control: deal with unreliable network path
  - Flow control: Prevent overwhelming receiving application
  - Congestion control: Prevent overwhelming the network
    - Loss-based and delay-based congestion detection
    - More and less aggressive rate control
    - Suitable for different network types
    - Fairness is important

# References

[1] IETF’s RFC page: http://www.ietf.org/rfc.html

[2] V. Jacobson: **Congestion Avoidance and Control**. *In proceedings of SIGCOMM '88.*

[3] L. Brakmo et al.: **TCP Vegas: New techniques for congestion detection and avoidance**. *In Proceedings of SIGCOMM '94.*

[4] **RFC2582/RFC3782 - The NewReno Modification to TCP's Fast Recovery Algorithm.**

[5] L. Hu et al.: **Binary Increase Congestion Control for Fast, Long Distance Networks**, *IEEE Infocom, 2004.*

[6] S. Ha et al.: **CUBIC: A New TCP-Friendly High-Speed TCP Variant**, *ACM SIGOPS, 2008.*

[7] K. Tan et al.: **Compound TCP: A Scalable and TCP-friendly Congestion Control for High-speed Networks**, *In IEEE Infocom, 2006.*

[8] W. John et al.: **Trends and Differences in Connection Behavior within Classes of Internet Backbone Traffic**, *In PAM 2008.*

[9] A. Medina et al.: **Measuring the evolution of transport protocols in the internet**, *SIGCOMM CCR, 2005.*