Computer Networks CS3001 (Section BDS-7A) Lecture 16
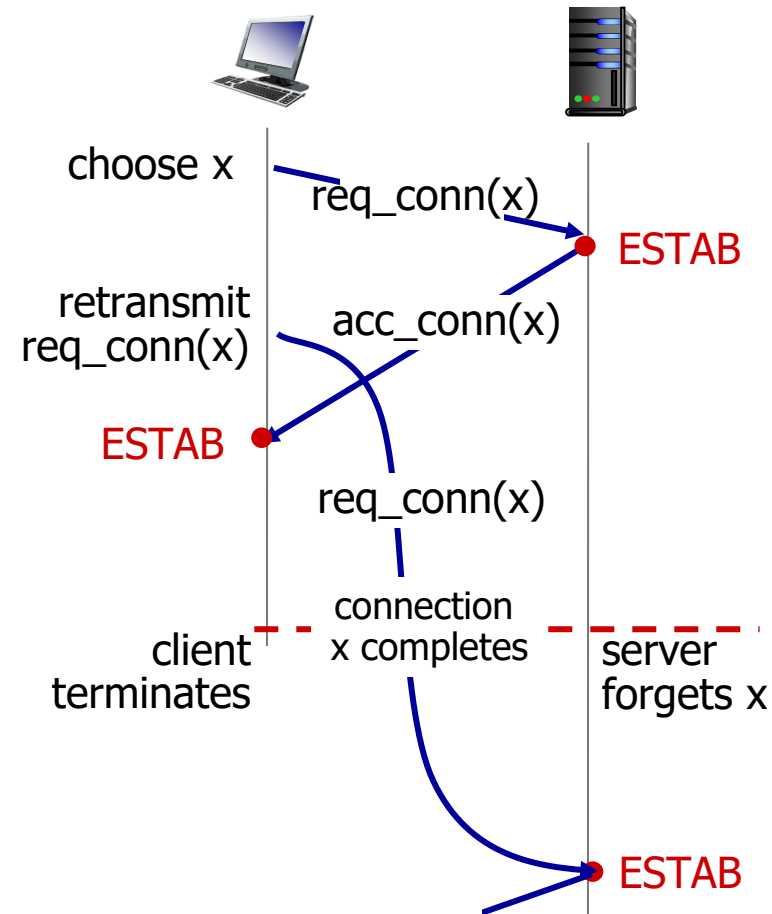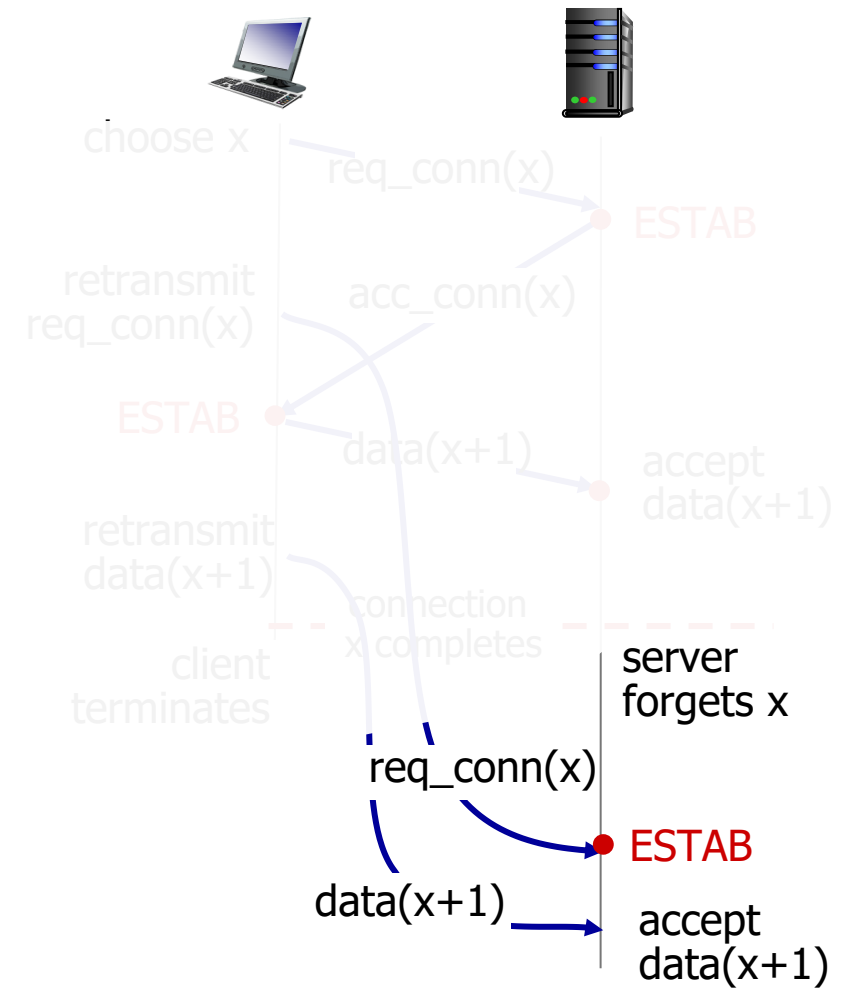
Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science

17 October, 2023

2-way handshake scenarios

choose x

req_conn(x)

ESTAB

retransmit
req_conn(x)

acc_conn(x)

ESTAB

req_conn(x)

connection
x completes

client
terminates

server
forgets x

ESTAB

❌ Problem: half open
connection! (no client)

2-way handshake scenarios

choose x

req_conn(x)

ESTAB

retransmit
req_conn(x)

acc_conn(x)

ESTAB

data(x+1)

accept
data(x+1)

retransmit
data(x+1)

connection
x completes

client
terminates

server
forgets x

req_conn(x)

ESTAB

data(x+1)

accept
data(x+1)

Problem: dup data accepted!

TCP 3-way handshake

Client state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```

LISTEN

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ESTAB

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTAB

A human 3-way handshake protocol

Closing a TCP connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1

- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN

- simultaneous FIN exchanges can be handled

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality

Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"

- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)

- different from flow control!

- a top-10 problem!

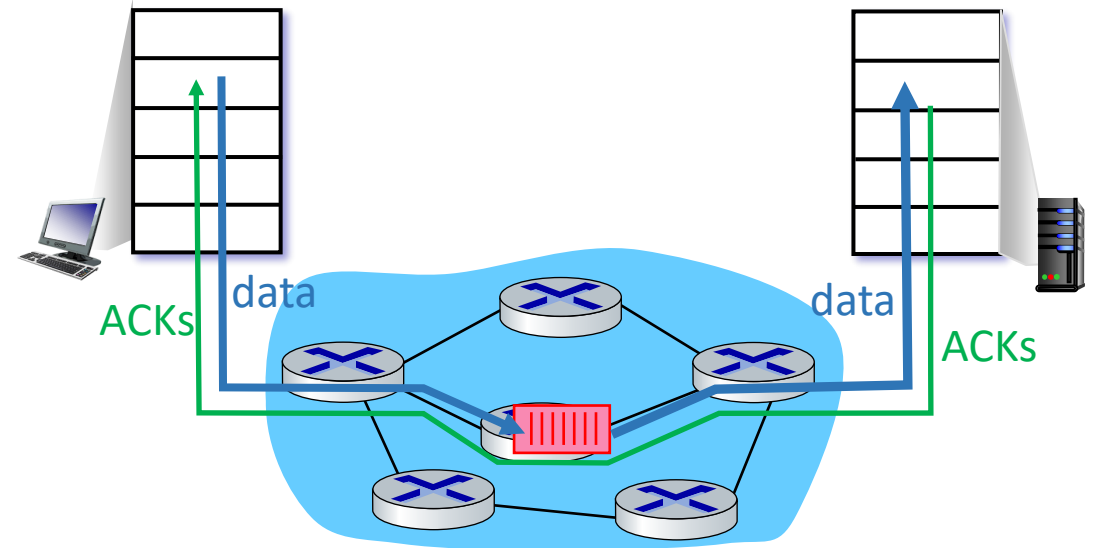**congestion control:** too many senders, sending too fast

flow control: one sender too fast for one receiver

# Approaches towards congestion control

End-end congestion control:

- no explicit feedback from network

- congestion *inferred* from observed loss, delay

  ▪ approach taken by TCP

Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router

- may indicate congestion level or explicitly set sending rate

• TCP ECN, ATM, DECbit protocols
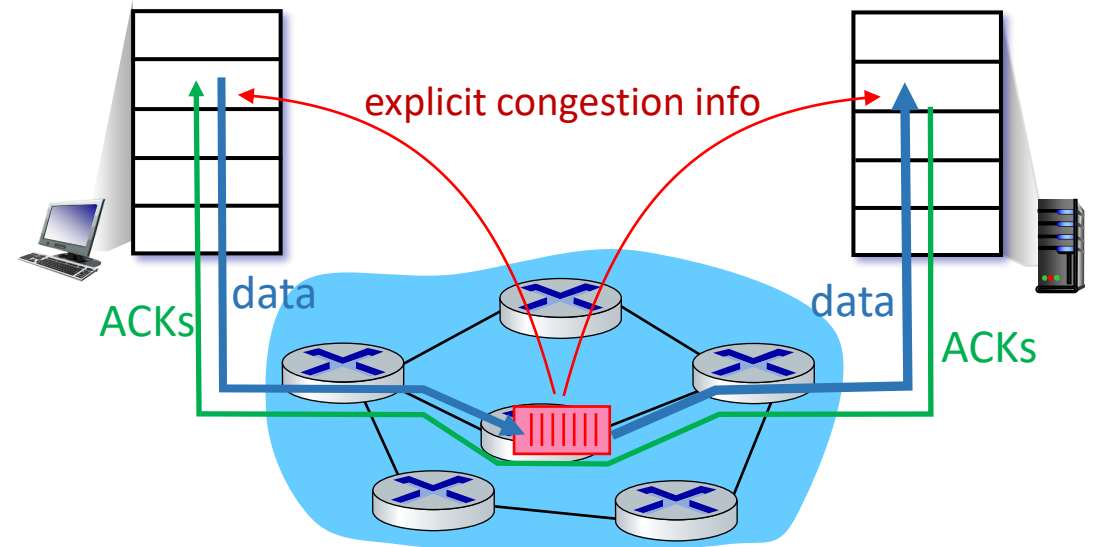
explicit congestion info
data
ACKs
data
ACKs

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- **TCP congestion control**
- Evolution of transport-layer functionality
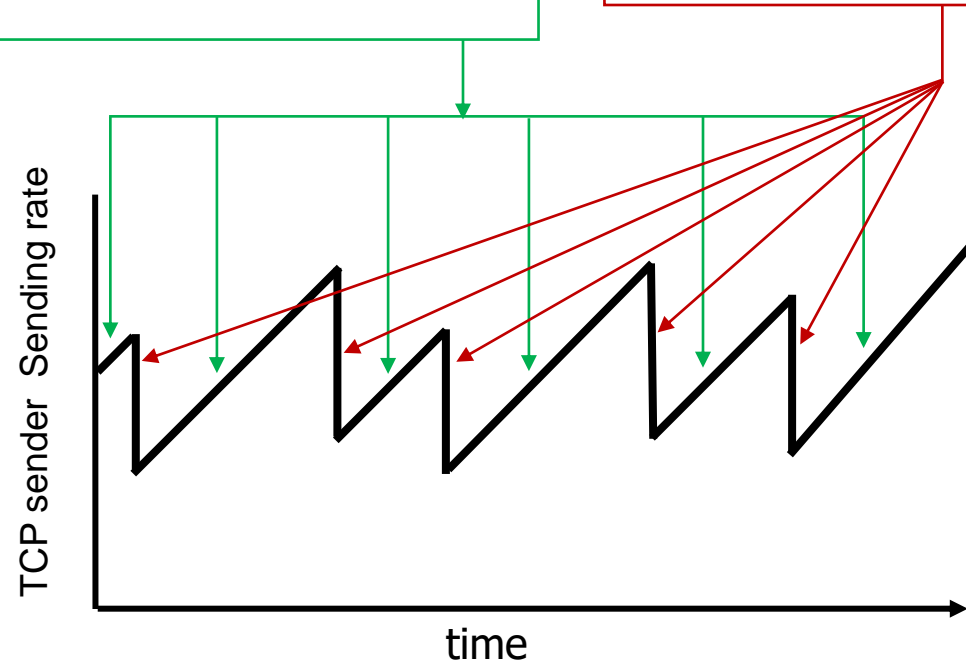
TCP congestion control: AIMD

- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event

AIMD sawtooth behavior: *probing* for bandwidth
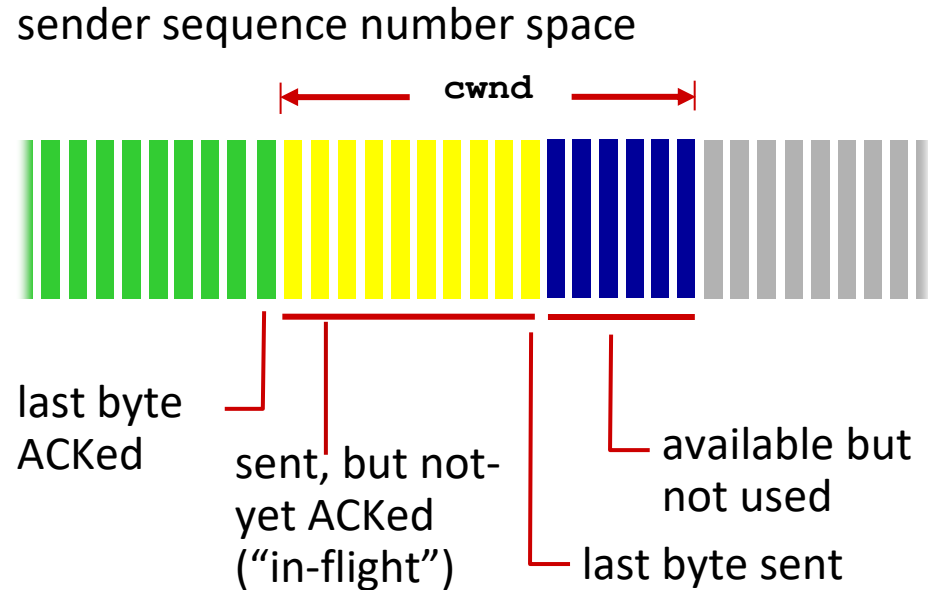
TCP sender Sending rate

time

TCP AIMD: more

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details

sender sequence number space
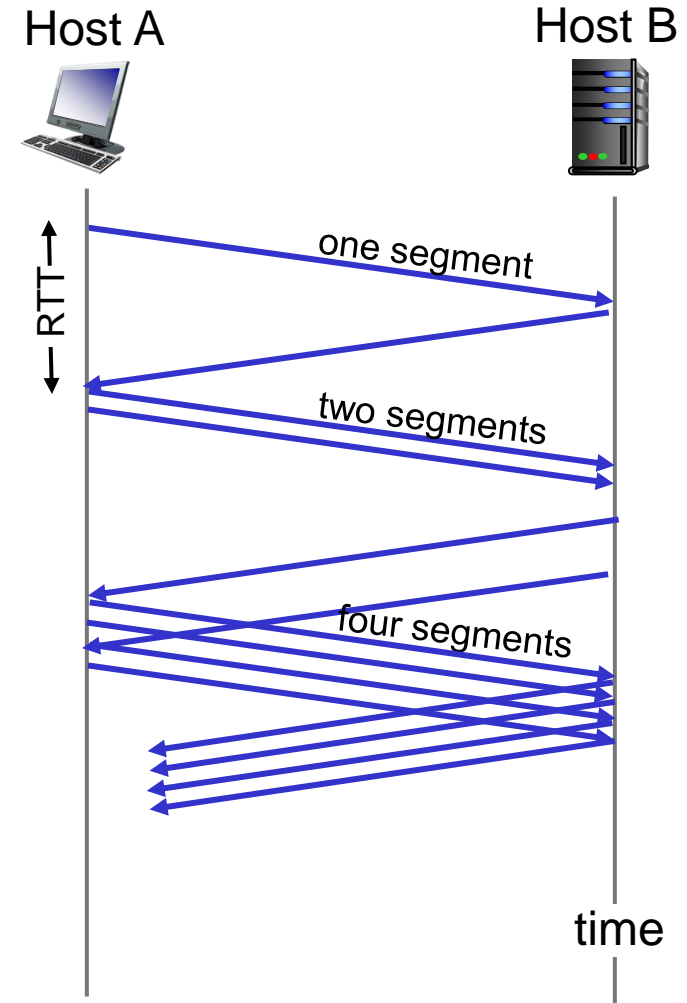
$$\overset{\text{cwnd}}{\longleftrightarrow}$$

- last byte ACKed
- sent, but not-yet ACKed ("in-flight")
- last byte sent
- available but not used

TCP sending behavior:

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

TCP rate $\approx \dfrac{\text{cwnd}}{\text{RTT}}$ bytes/sec

- TCP sender limits transmission: LastByteSent– LastByteAcked $\leq$ cwnd

- cwnd is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially `cwnd` = 1 MSS
  - double `cwnd` every RTT
  - done by incrementing `cwnd` for every ACK received

- *summary:* initial rate is slow, but ramps up exponentially fast

Host A   Host B

one segment

two segments

four segments

time
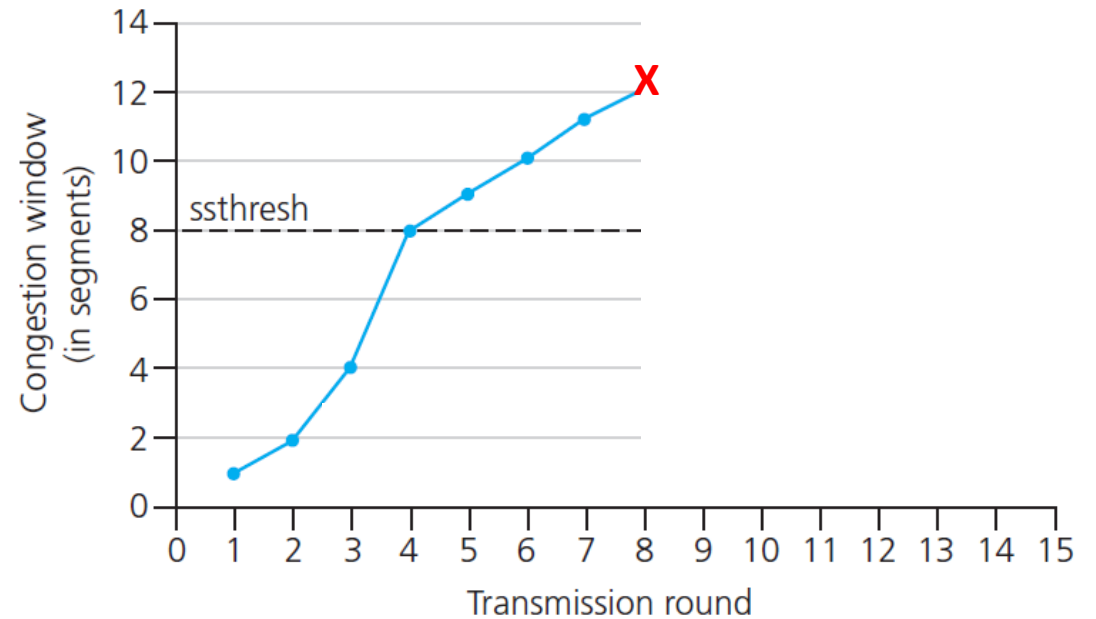
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when `cwnd` gets to 1/2 of its value before timeout.

Implementation:

- variable `ssthresh`

- on loss event, `ssthresh` is set to 1/2 of `cwnd` just before loss event

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/