Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

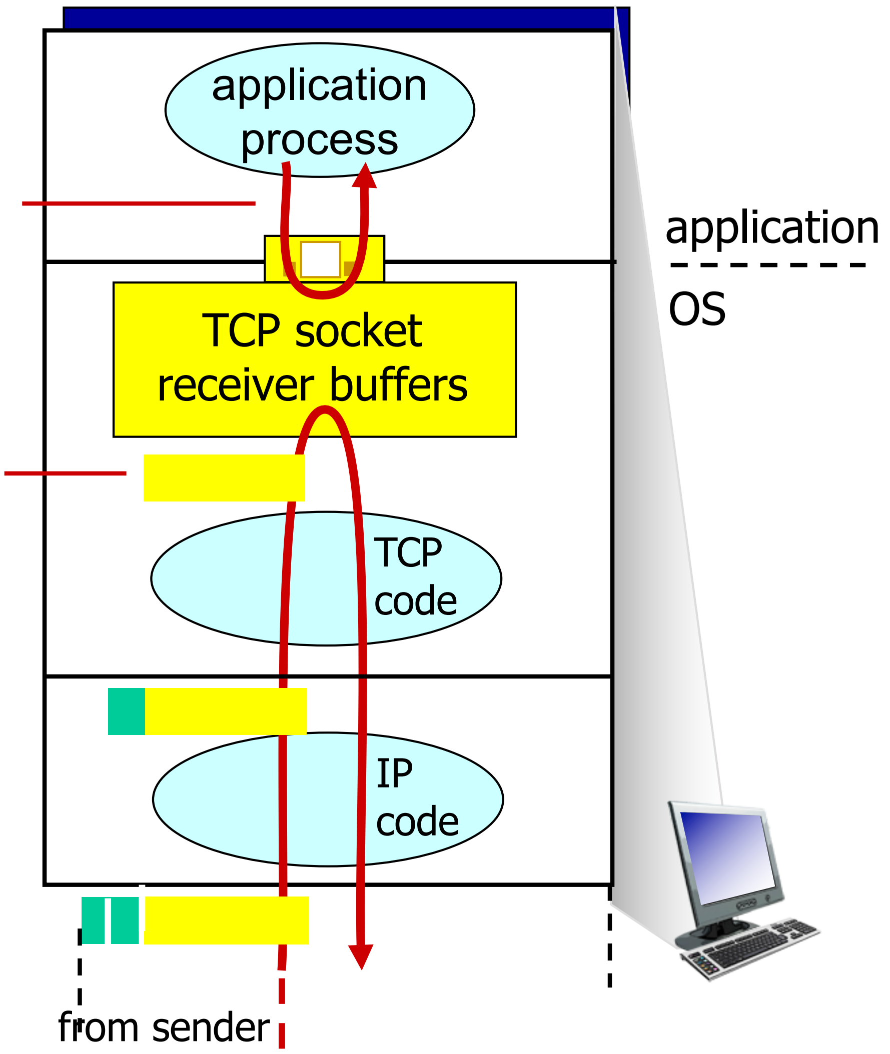
3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



application process

TCP socket receiver buffers

TCP code

IP code

application

OS

from sender

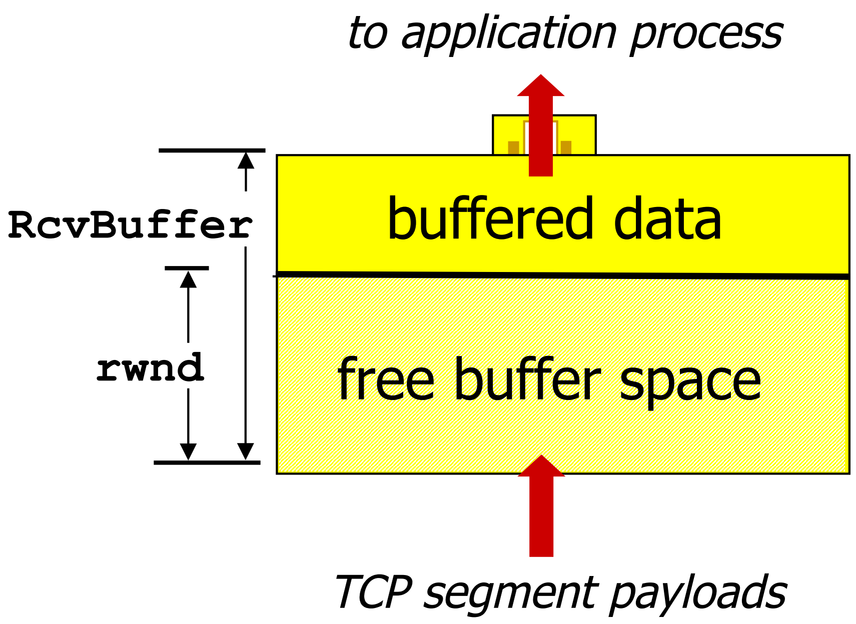
receiver protocol stack

flow control

receiver controls sender, so that sender won't overflow receiver's buffer by transmitting too much, too fast

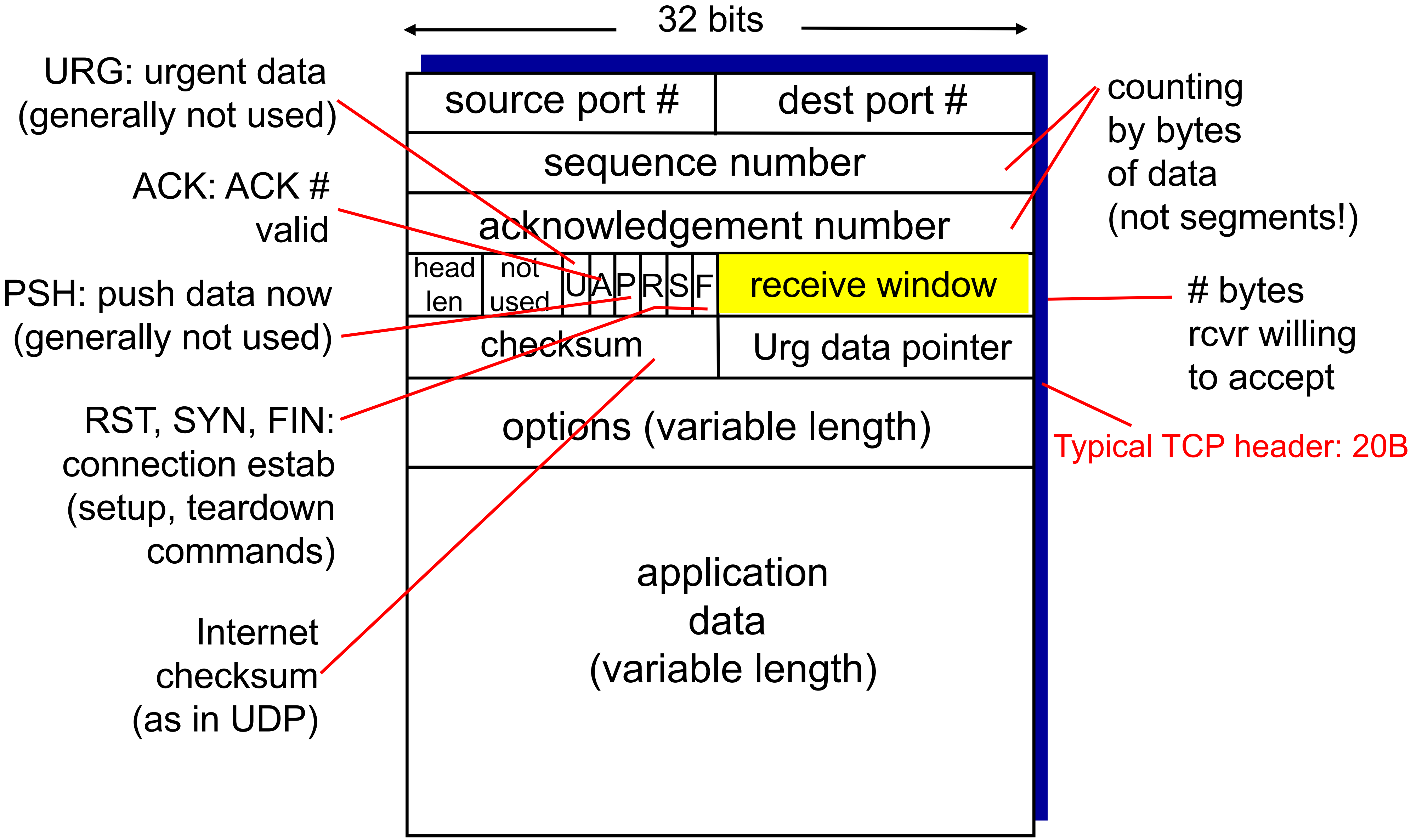
TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



receiver-side buffering

TCP segment structure


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
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
Typical TCP header: 20B

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

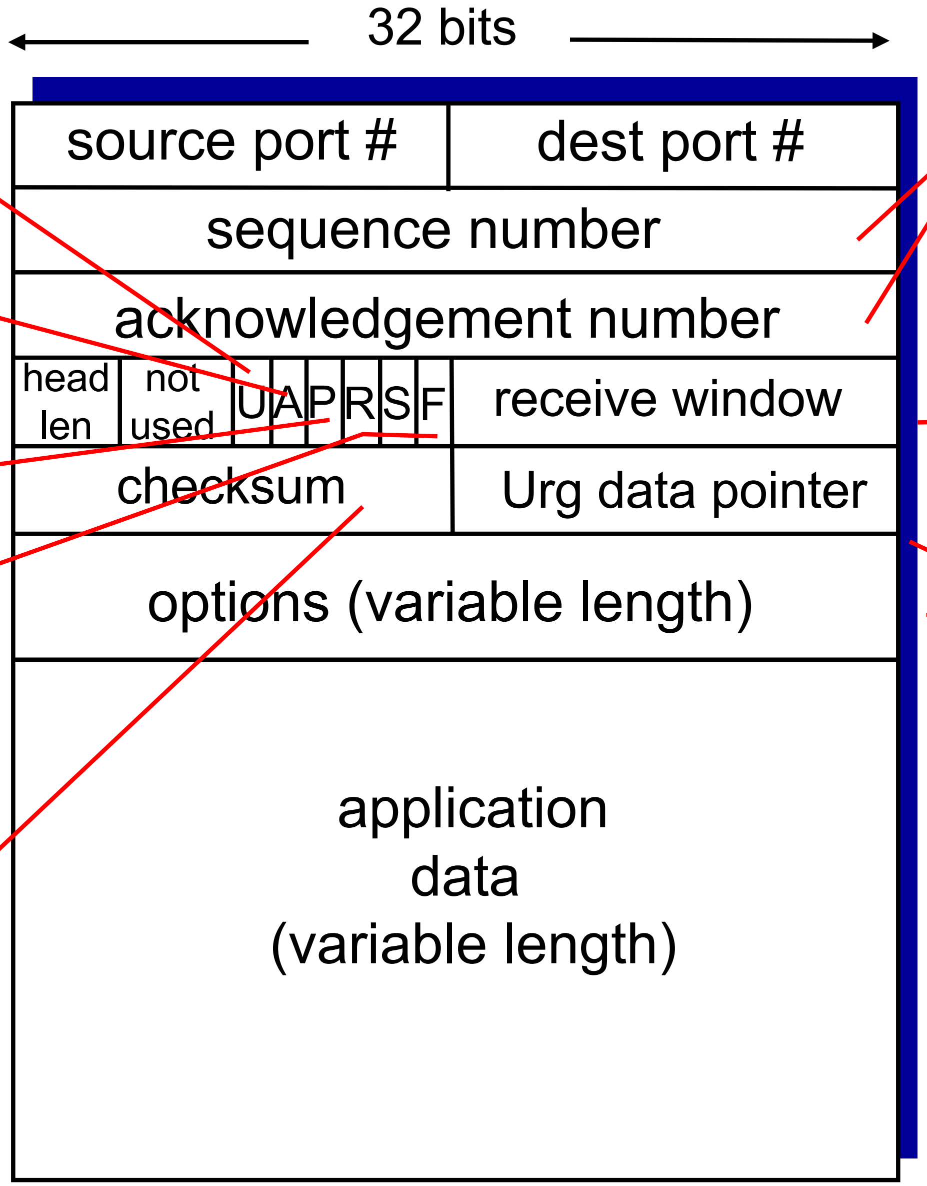
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP segment structure



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

Typical TCP header: 20B

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

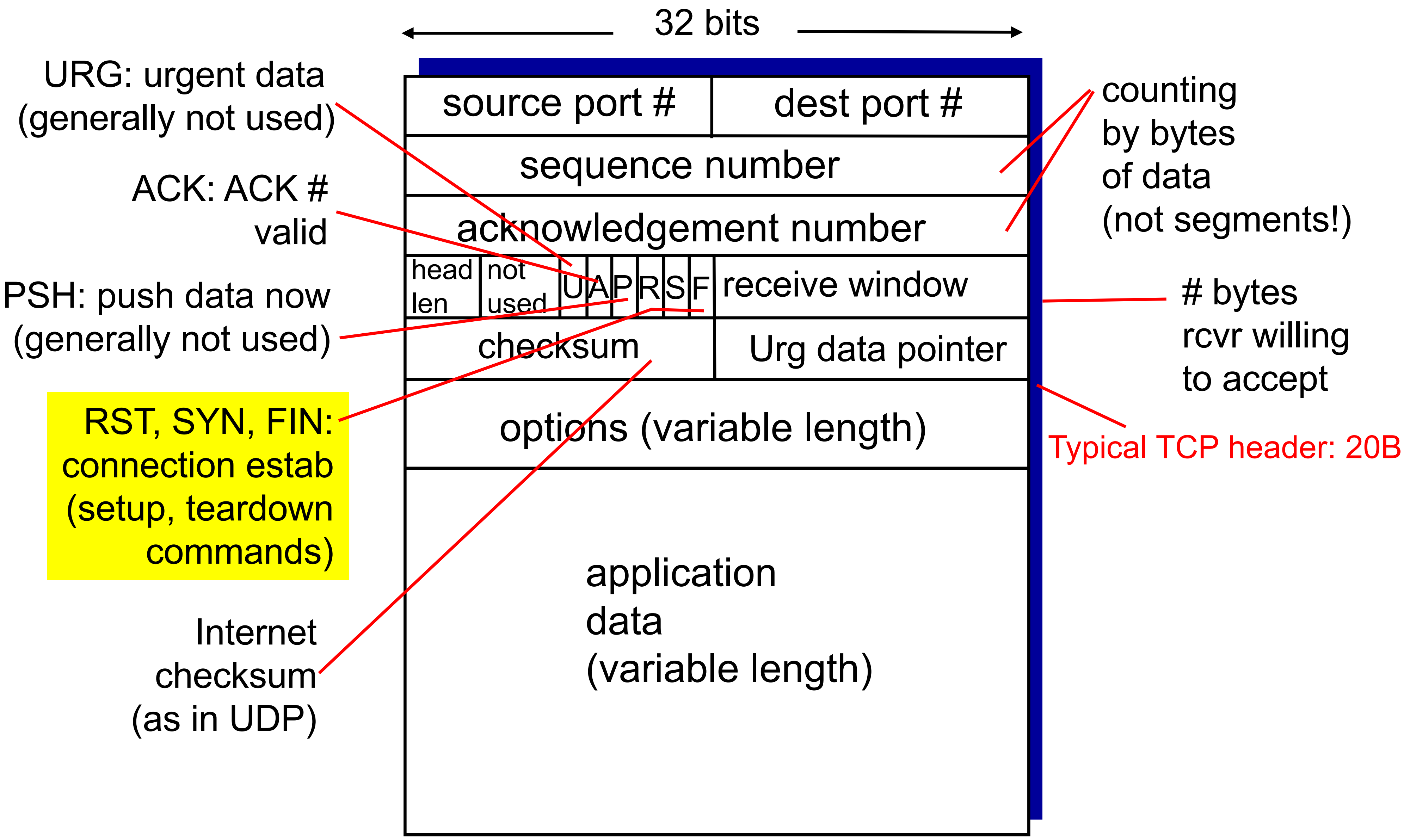
3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP segment structure



32 bits

source port #

dest port #

sequence number

acknowledgement number

head len

not used

U A P R S F

receive window

checksum

Urg data pointer

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!)

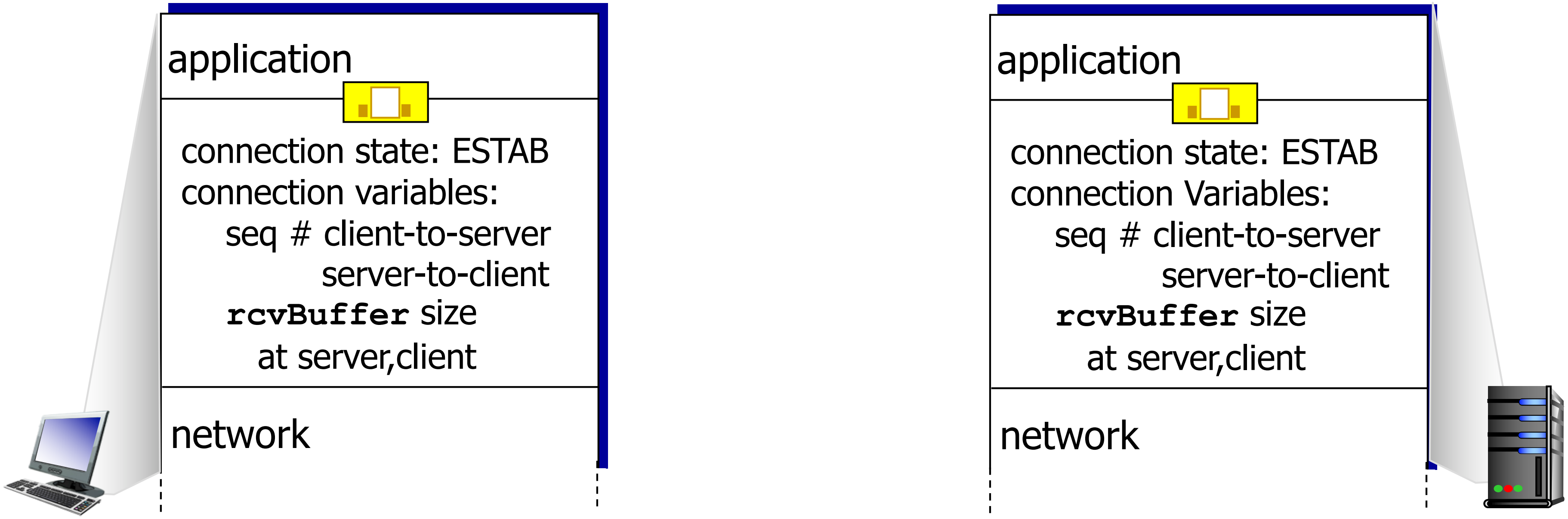
# bytes rcvr willing to accept

Typical TCP header: 20B

Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



application

connection state: ESTAB
connection variables:
seq # client-to-server
server-to-client
rcvBuffer size
at server,client

network

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

application

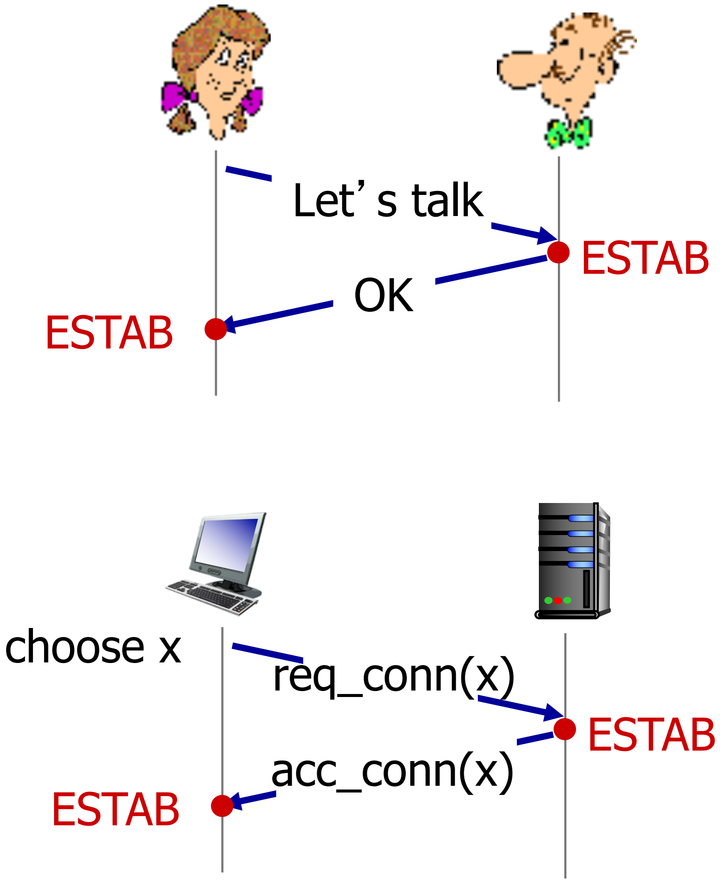
connection state: ESTAB
connection Variables:
seq # client-to-server
server-to-client
rcvBuffer size
at server,client

network

```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:

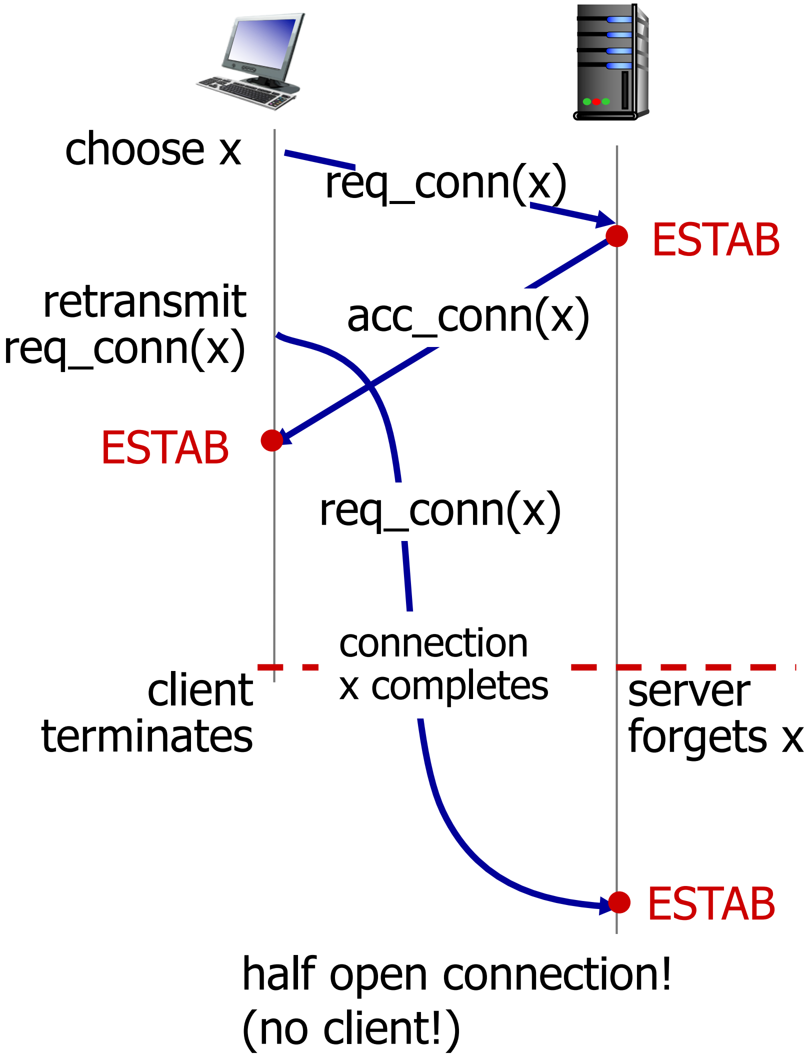


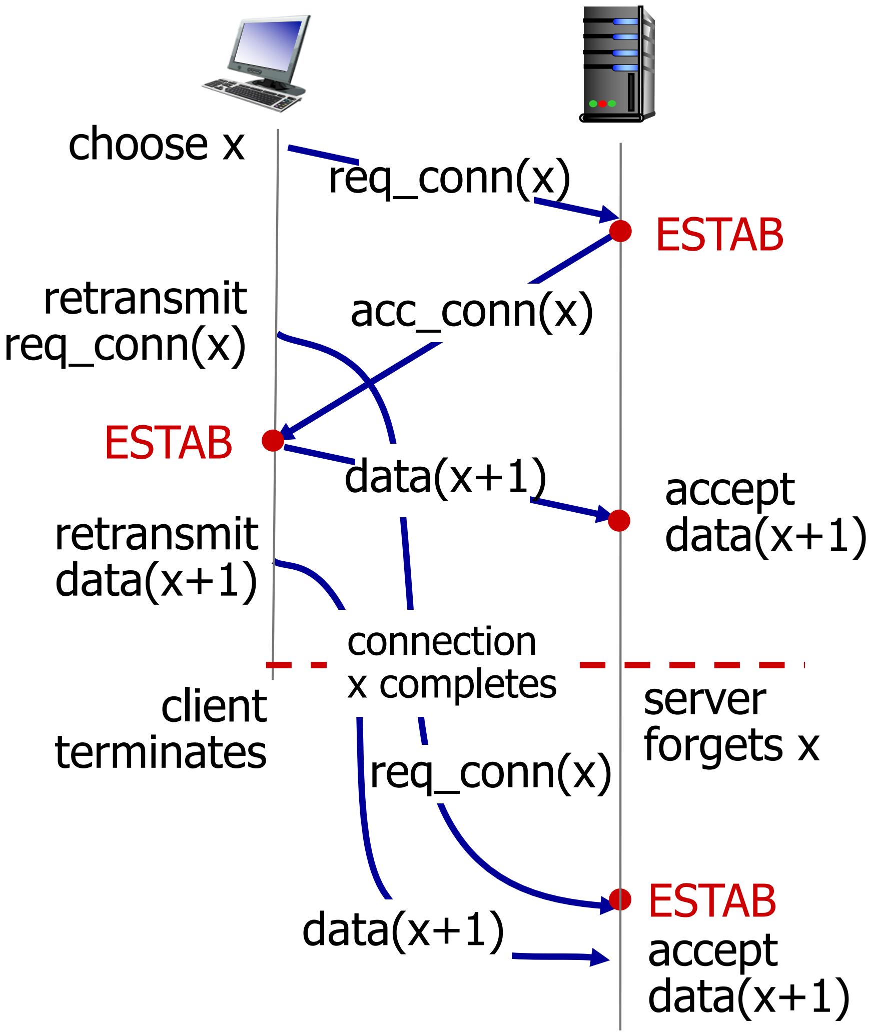
Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can' t "see" other side

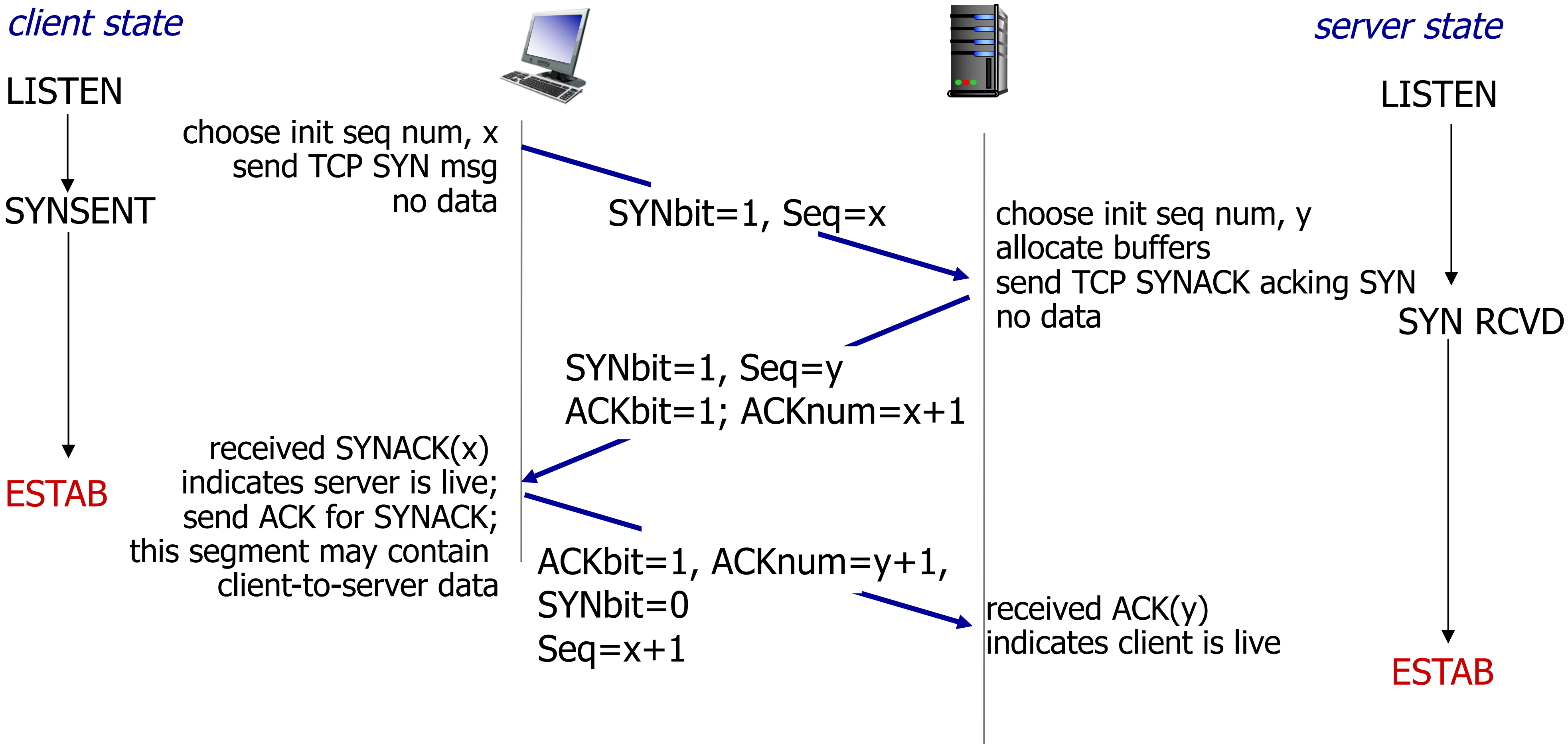
Agreeing to establish a connection

2-way handshake failure scenarios:





TCP 3-way handshake



client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg
no data

SYNbit=1, Seq=x

choose init seq num, y
allocate buffers
send TCP SYNACK acking SYN
no data

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

ACKbit=1, ACKnum=y+1,
SYNbit=0
Seq=x+1

received ACK(y)
indicates client is live

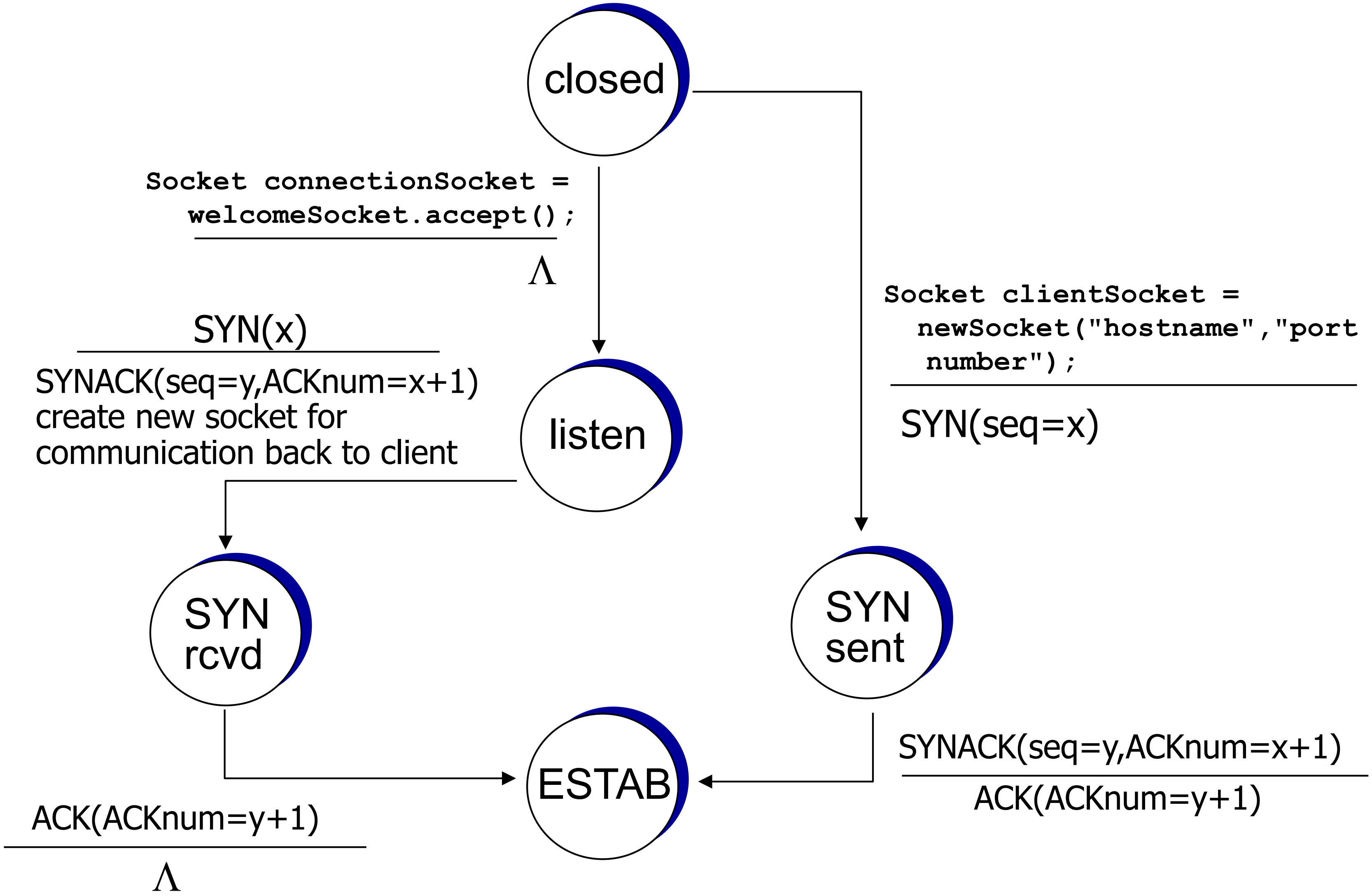
server state

LISTEN

SYN RCVD

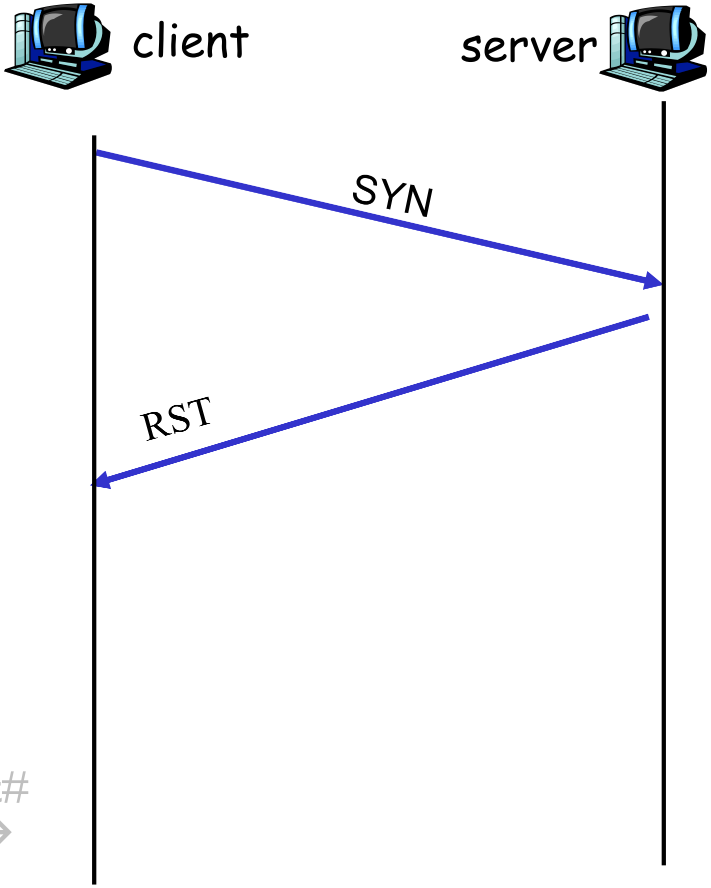
ESTAB

TCP 3-way handshake: FSM



Server may not accept the connection

- Why?
  - Server may not be accepting TCP connections to that port
  - Server may be out of resources
- What happens?
  - Sends RST
  - No connection established, no resources allocated
  - But client learns that port is open
- UDP servers do not have connections
  - just listen to a socket on a dest port#
  - If server receives a UDP packet with dest port# that does not match an existing UDP socket → Sends ICMP message back

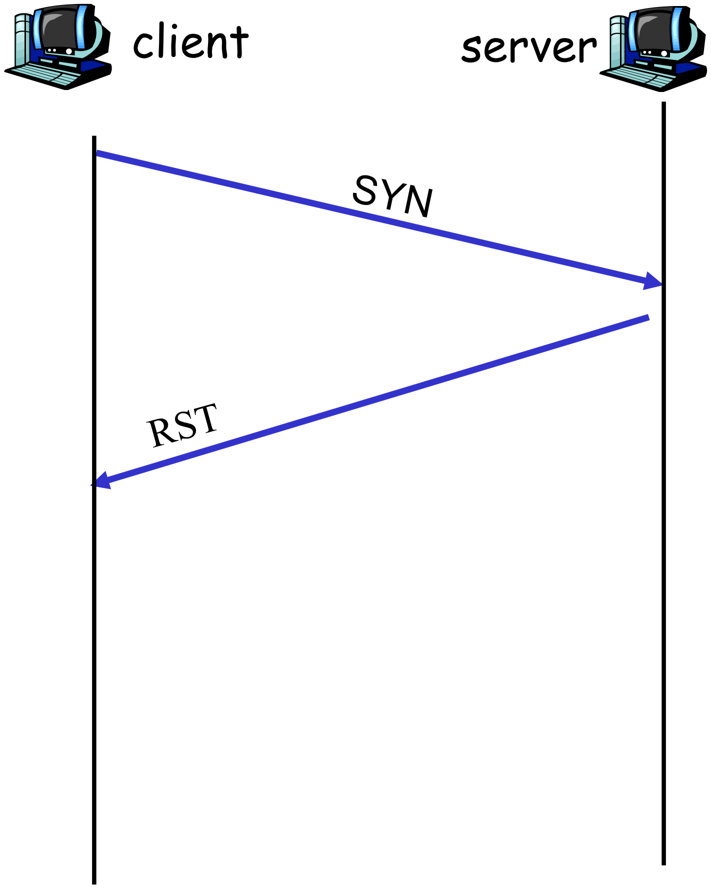


Attack 0: Scanning ports

- www.nmap.org

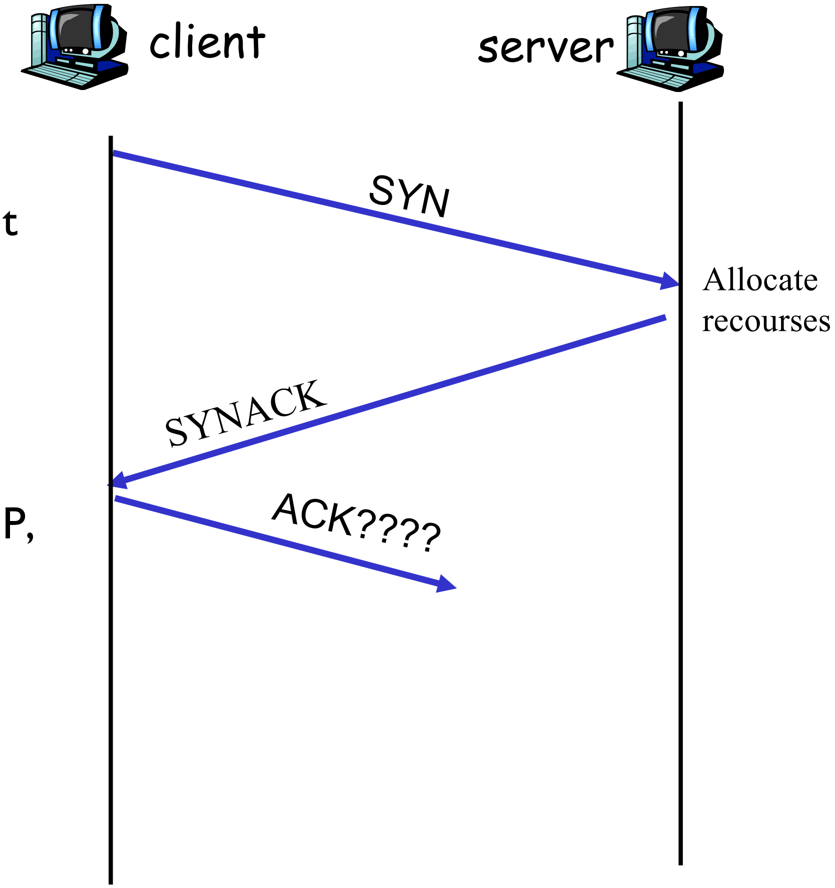
- Scanning TCP ports
  - Send TCP SYN
  - receive SYNACK, RST, nothing

- Scanning UDP ports
  - Receive ICNP messages

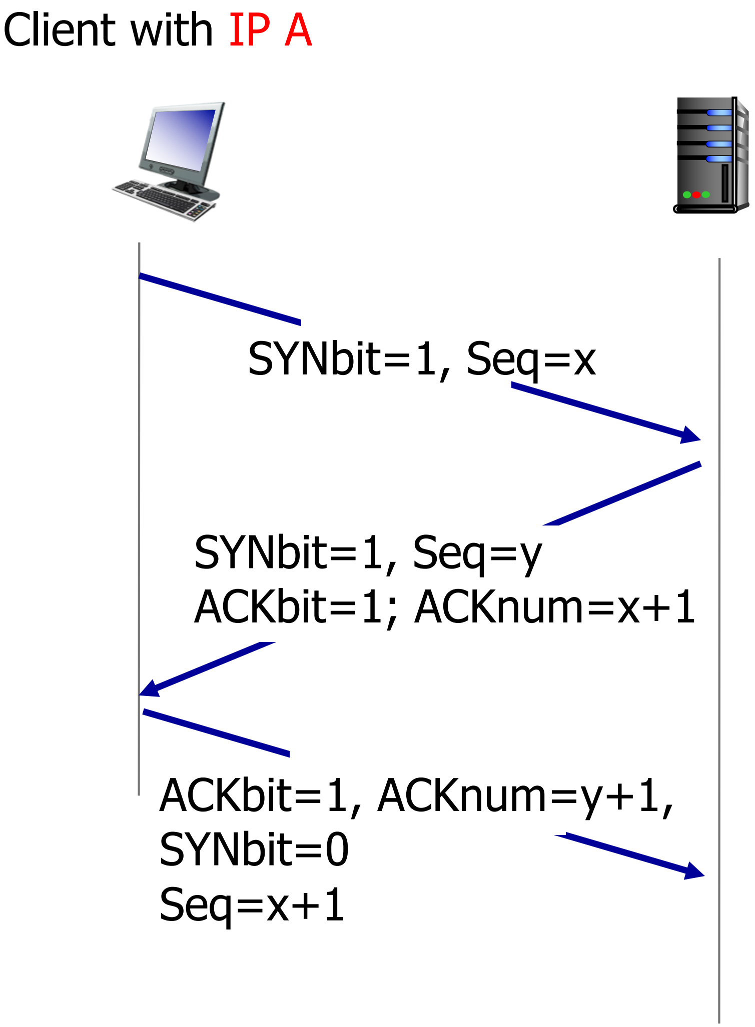


Attack I: SYN FLOOD attacks

- Client never completes the handshake
- Server allocates resources
- SYN FLOOD: the oldest (D)DoS attack
  - Server does not have resources to server legit clients
- A Solution: SYN Cookies
  - Server creates a cookie:=a hash of the client IP, port on the SYN segment (and of a secret number known to the server
  - Server sends SYNACK with that initial seqno=cookie, but does NOT allocates resources (half open connection)
  - If client is legit, it will send an ACK with initialseqno; server can verify that ACK corresponds to SYN; only then sender allocates resources (full open connection)



Attack II: Spoofing



Client with IP A

SYNbit=1, Seq=x

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

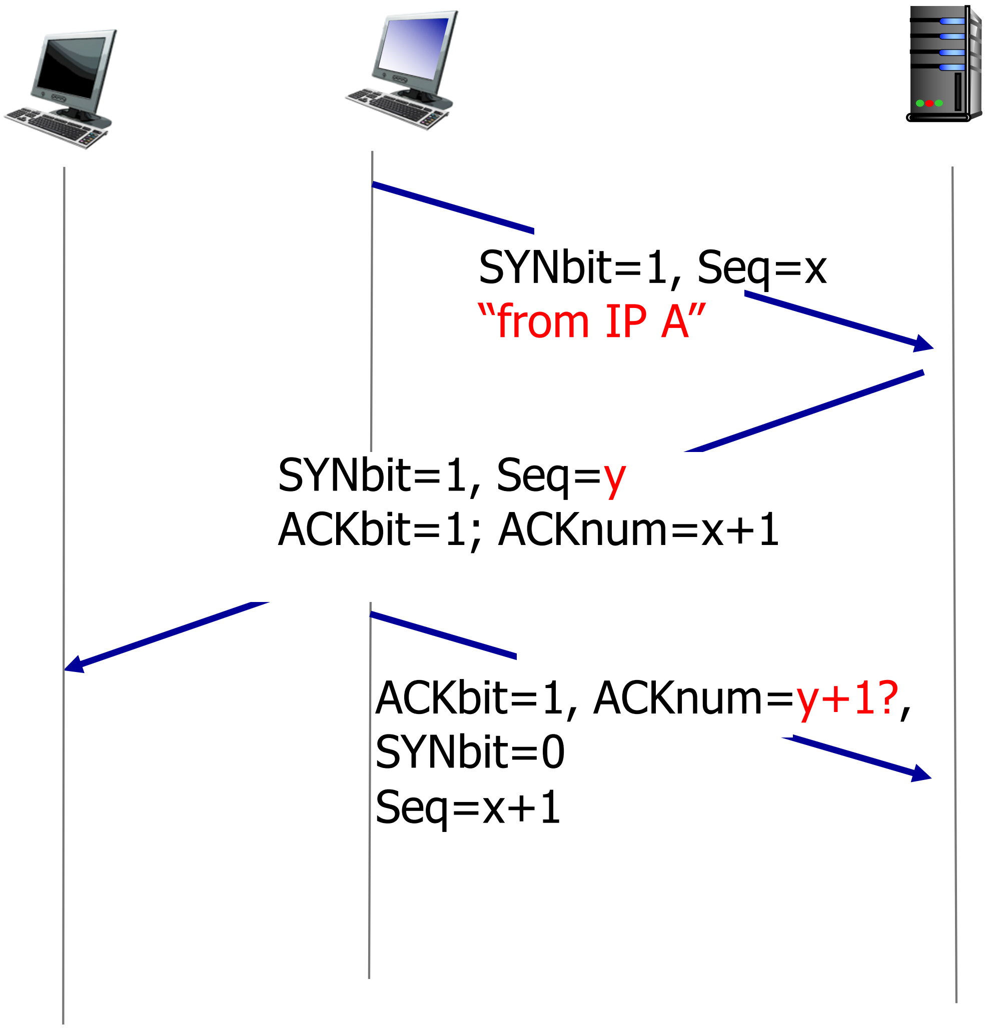
ACKbit=1, ACKnum=y+1,
SYNbit=0
Seq=x+1

Can the server be sure that this is really A, and not B pretending to be A?

Attack II: Spoofing

Attacker with IP B (pretending to be A)

Real IP A



SYNbit=1, Seq=x
"from IP A"

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

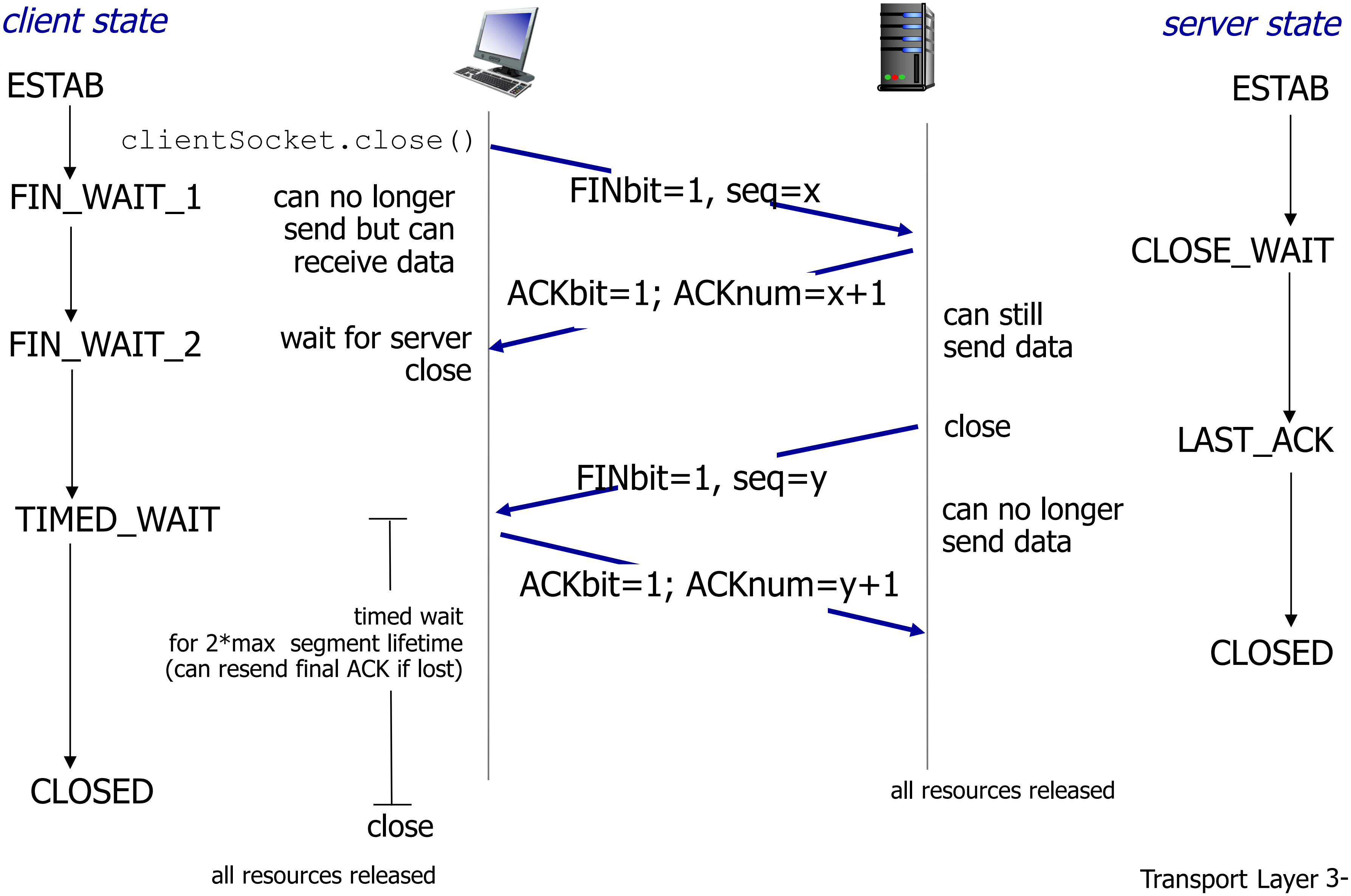
ACKbit=1, ACKnum=y+1?,
SYNbit=0
Seq=x+1

ACKnum(y+1) is really from IP A,
if there is no man-in-the-middle
attack.

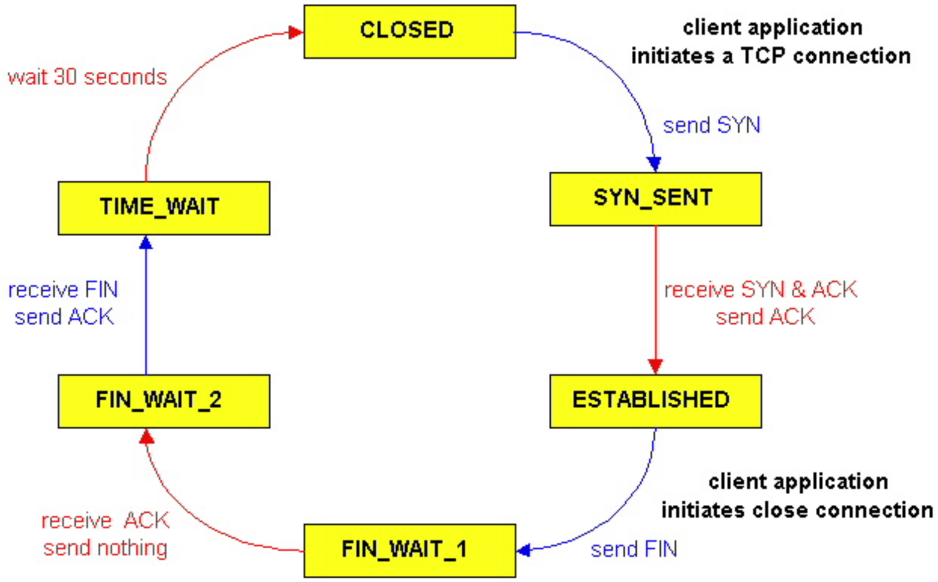
TCP: closing a connection

- remember: this is a duplex connection
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
  - either of the two can initiate the closing
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection


client state
server state
ESTAB
clientSocket.close()
FIN_WAIT_1
can no longer send but can receive data
FINbit=1, seq=x
CLOSE_WAIT
ACKbit=1; ACKnum=x+1
FIN_WAIT_2
wait for server close
can still send data
close
LAST_ACK
FINbit=1, seq=y
TIMED_WAIT
can no longer send data
ACKbit=1; ACKnum=y+1
timed wait for 2*max segment lifetime (can resend final ACK if lost)
CLOSED
close
all resources released
all resources released
ESTAB
CLOSED

TCP Connection States



CLOSED

client application initiates a TCP connection

send SYN

SYN_SENT

receive SYN & ACK
send ACK

ESTABLISHED

client application initiates close connection

send FIN

FIN_WAIT_1

receive ACK
send nothing

FIN_WAIT_2

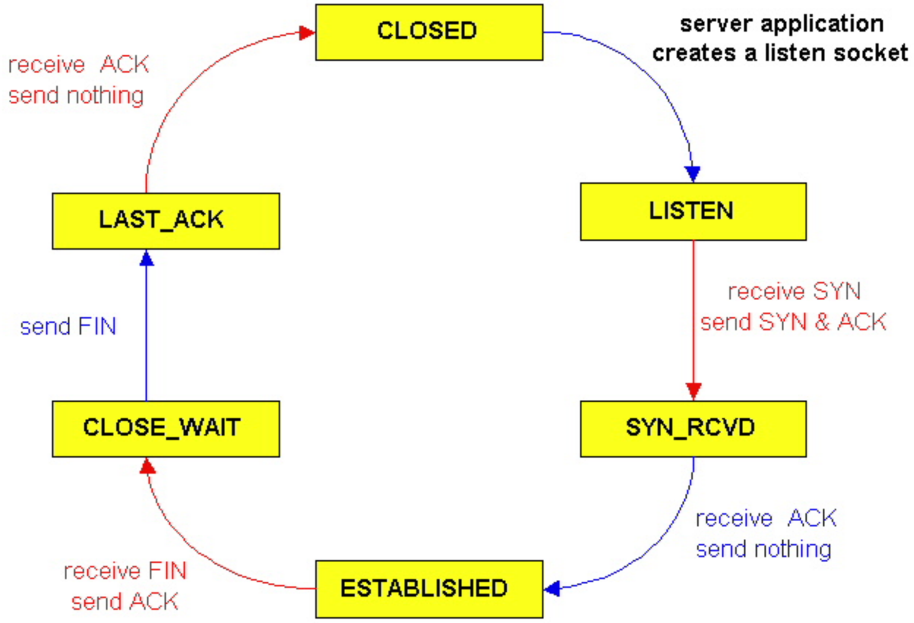
receive FIN
send ACK

TIME_WAIT

wait 30 seconds

TCP client lifecycle

TCP server lifecycle



CLOSED

server application creates a listen socket

LISTEN

receive SYN
send SYN & ACK

SYN_RCVD

receive ACK
send nothing

ESTABLISHED

receive FIN
send ACK

CLOSE_WAIT

send FIN

LAST_ACK

receive ACK
send nothing