COMP 431
Internet Services & Protocols

The Transport Layer

Reliable data delivery & flow control in TCP

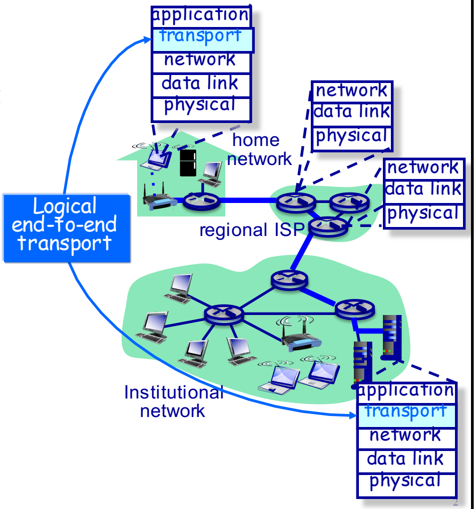
Jasleen Kaur

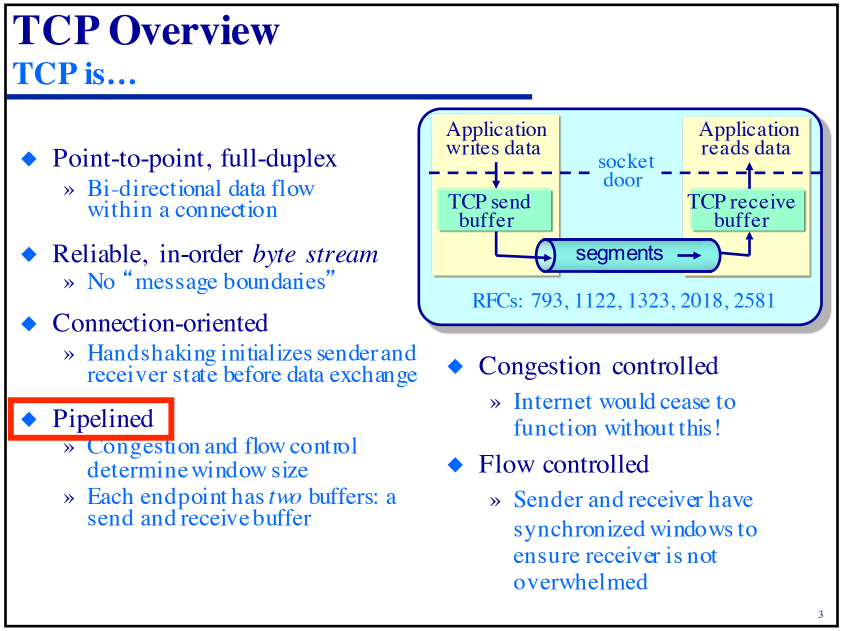
March 23, 2020

Transport Layer Protocols & Services

Outline

- Fundamental transport layer services
  - » Multiplexing/Demultiplexing
  - » Error detection
  - » Reliable data delivery
  - » Pipelining
  - » Flow control
  - » Congestion control
- Internet transport protocols
  - » UDP
  - » TCP





TCP Overview

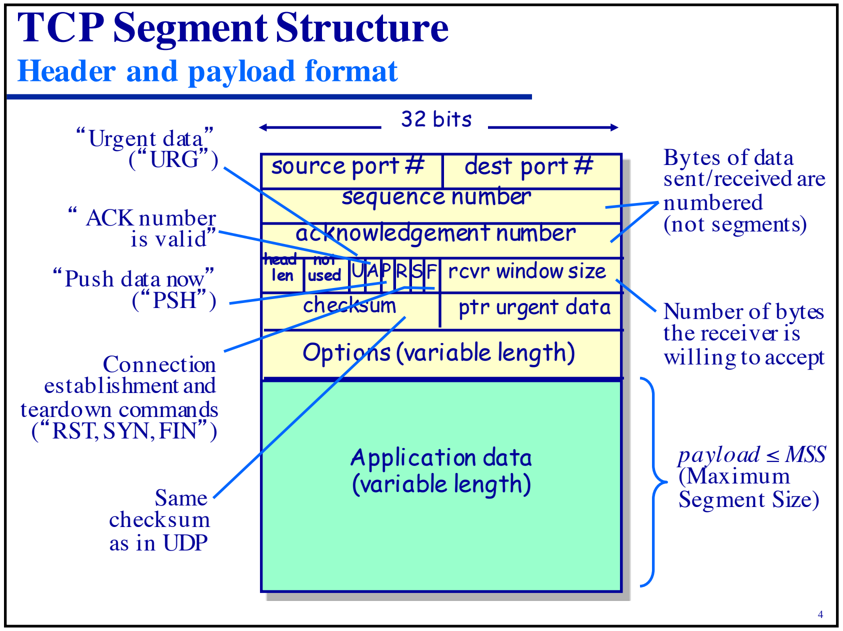
TCP is…

- Point-to-point, full-duplex
  - » Bi-directional data flow within a connection
- Reliable, in-order *byte stream*
  - » No "message boundaries"
- Connection-oriented
  - » Handshaking initializes sender and receiver state before data exchange
- Pipelined
  - » Congestion and flow control determine window size
  - » Each endpoint has *two* buffers: a send and receive buffer
- Congestion controlled
  - » Internet would cease to function without this!
- Flow controlled
  - » Sender and receiver have synchronized windows to ensure receiver is not overwhelmed

RFCs: 793, 1122, 1323, 2018, 2581

UDP: datagram oriented, connectionless, only between sender and receiver

Each endpoint in TCP is both a sender and a receiver



TCP Segment Structure

Header and payload format

"Urgent data" ("URG")

" ACK number is valid"

"Push data now" ("PSH")

Connection establishment and teardown commands ("RST, SYN, FIN")

Same checksum as in UDP

Bytes of data sent/received are numbered (not segments)

Number of bytes the receiver is willing to accept

payload ≤ *MSS* (Maximum Segment Size)

Sequence number tells where in the byte stream that data belongs

U and P are not really used

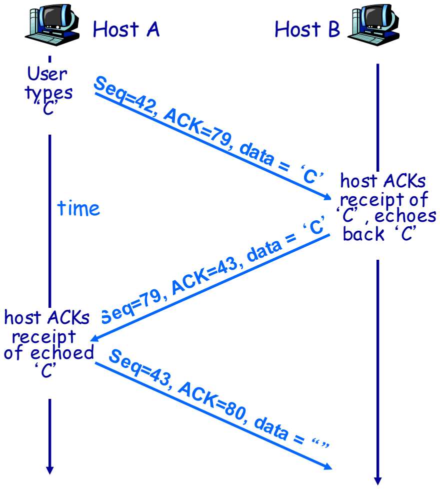
R: reset, restart to terminate connection
S: synchronize, part of 3 way handshake
F: finish, use at the end of a connection

Window size plays a part in flow control

TCP Sequence Numbers and ACKs

Telnet example

- Sequence numbers:
  - » Byte stream "index" of the first byte in the segment's payload
- ACKs:
  - » Sequence number of next byte expected from the other side
  - » ACKs are cumulative
- How does receiver handle out-of-order segments?
  - » TCP spec doesn't say, it's up to the implementor



Host A

Host B

User types 'C'

Seq=42, ACK=79, data = 'C'

host ACKs receipt of 'C', echoes back 'C'

time

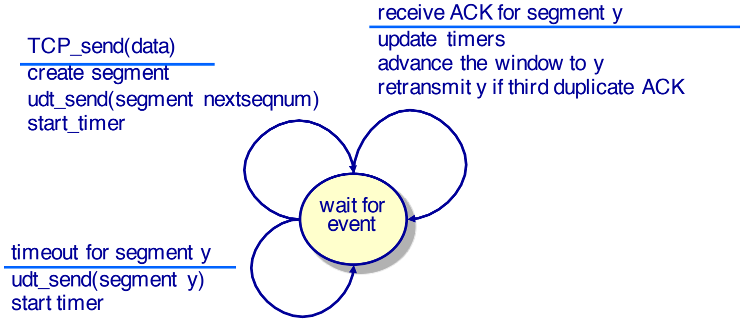
Seq=79, ACK=43, data = 'C'

host ACKs receipt of echoed 'C'

Seq=43, ACK=80, data = ""

Reliable Data Transfer in TCP

Sender's state machine (simplified!)



TCP_send(data)

create segment

udt_send(segment nextseqnum)

start_timer

receive ACK for segment y

update timers

advance the window to y

retransmit y if third duplicate ACK

wait for event

timeout for segment y

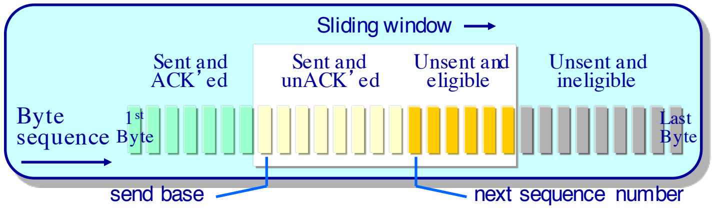
udt_send(segment y)

start timer

- TCP retransmits segments if:
  - » An expected ACK times out
  - » 3 duplicate ACKs for a segment are received

Reliable Data Transfer in TCP

Simplified sender's state machine



```
sendbase      = initial_sequence number
nextseqnum  = initial_sequence number

loop (forever) {
  switch(event)
    event: data received from application
              above
    event: timer timeout for segment with
              sequence number y
    event: ACK received with value y
  }
```

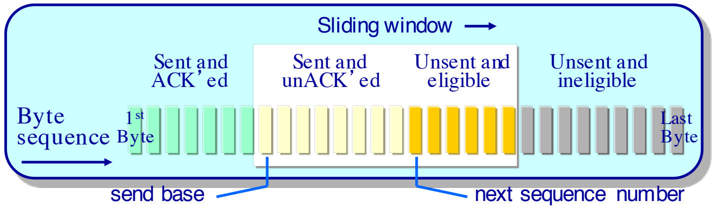
```
create TCP segment with sequence
          number nextseqnum
start timer for segment with nextseqnum
pass segment to IP
nextseqnum = nextseqnum + length(data)
```

```
retransmit segment with sequence
          number y ("segment y")
compute new timeout interval for segment y
restart timer for segment y
```

MSS: maximum segment size

Reliable Data Transfer in TCP

Simplified sender's state machine



```
sendbase      = initial_sequ
nextseqnum  = initial_seque

loop (forever) {
  switch(event)
    event: data received fro
              above
    event: timer timeout for
              sequence numbe
    event: ACK received wi
  }
```

```
if (y > sendbase) {     /* Cumulative ACK of all data up to y */
   cancel timers for segments with sequence numbers ≤ y
   sendbase := y
 }
else if ( y == sendbase) {                    /* A duplicate ACK */
   increment number of duplicate ACKs received
   if (number of duplicate ACKS received == 3)
    {                                           /* Fast retransmit */
      resend segment with sequence number sendbase
      restart timer for segment sendbase
    }
 }
```

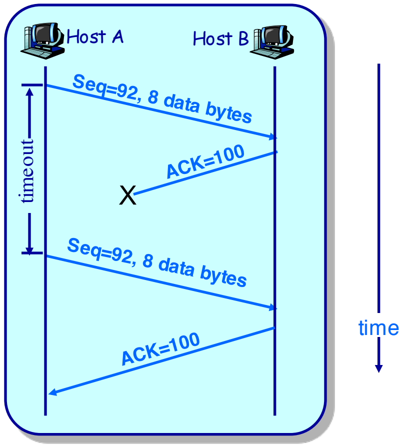
Reliable Data Transfer in TCP

Receiver ACK generation rules [RFC 1122, RFC 2581]

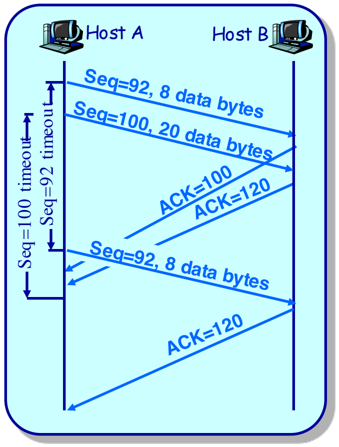
Event		TCP Receiver action
In-order segment arrival, no gaps, all previous data already ACKed	⇨	Delayed ACK. Wait 200 *ms* (up to 500 *ms* allowed) for next segment. If no segment received, send ACK
In-order segment arrival, no gaps, one delayed ACK pending	⇨	Immediately send single cumulative ACK
Out-of-order segment arrival (higher than expected sequence number) — Gap detected	⇨	Send duplicate ACK, indicating sequence number of next expected byte
Arrival of segment that partially or completely fills gap	⇨	Immediate ACK if segment starts at lower end of gap

Reliable Data Transfer in TCP

Retransmission examples



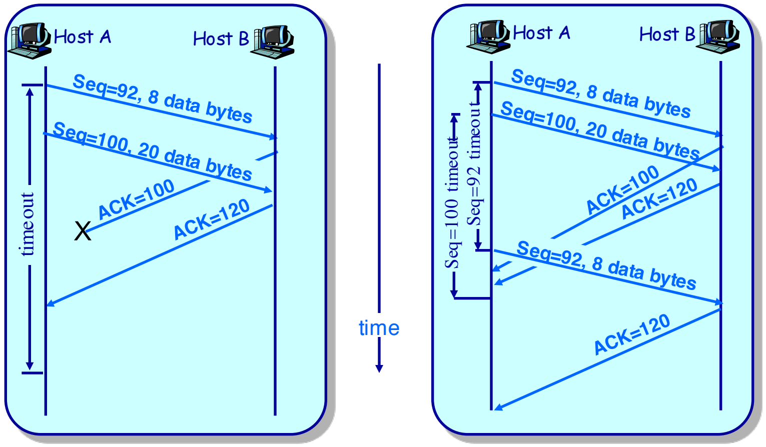
- Lost ACK scenario



- Premature timeout

Reliable Data Transfer in TCP

Retransmission examples

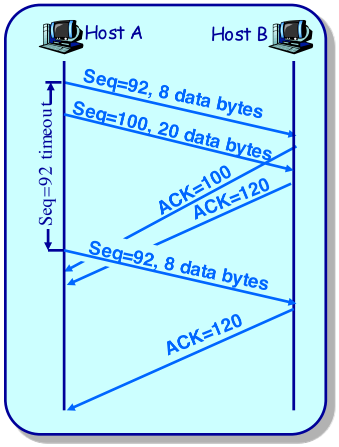


- Cumulative ACKs potentially avoid retransmissions
- Premature timeout

Reliable Data Transfer in TCP

Setting the ACK timer

- How large should the ACK timeout value be?
  - Too short: Premature timeouts result in unnecessary retransmissions
  - Too long: Slow reaction to loss results in poor performance because the sender's windows stops advancing
- Timer should be longer than the RTT, but how do we estimate RTT?

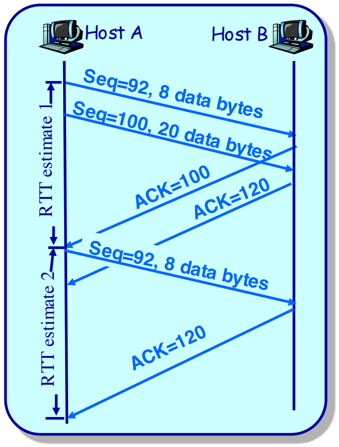


Text

Reliable Data Transfer in TCP

Setting the ACK timer

- Measure the time from segment transmission until receipt of ACK (“**`SampleRTT`**”)
  - » Ignore retransmissions
  - » Measure only one segment’s RTT at a time
- **`SampleRTT`** will vary, so we compute an average RTT based on several recent RTT samples



Reliable Data Transfer in TCP

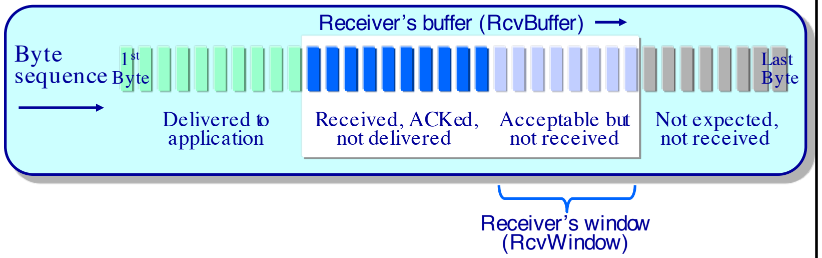
Estimating round-trip-time

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
     Timeout = EstimatedRTT + 4*Deviation
   Deviation = (1-x)*Deviation +
                          x*|SampleRTT-EstimatedRTT|
```

- The estimated RTT is an exponential weighted moving average (EWMA)
  - » Computes a “smooth” average
  - » Influence of a given sample decreases exponentially fast
    $E_n = x*S_n + x(1-x)S_{n-1} + x(1-x)^2S_{n-2} + \ldots + x(1-x)^iS_{n-i} + \ldots$
  - » Typical value of *x* is 0.125
- Timeout is **`EstimtedRTT`** plus “safety margin”
- Large variation in **`EstimatedRTT`** results in a larger safety margin

TCP Flow Control

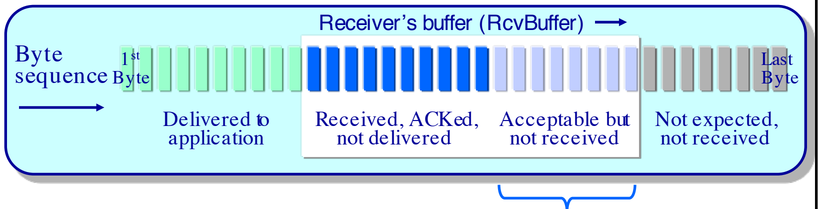
Receiver Window control


Receiver's buffer (RcvBuffer)
Byte sequence
1st Byte
Last Byte
Delivered to application
Received, ACKed, not delivered
Acceptable but not received
Not expected, not received
Receiver's window (RcvWindow)

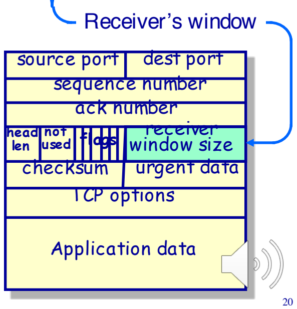
- Flow control is the problem of ensuring the receiver is not overwhelmed
  - » The receiver can become overwhelmed if the application reads too slow or the sender transmits too fast
- The receiver's window represents its remaining buffer capacity
- The window advances as the application reads received data

TCP Flow Control

Receiver window control

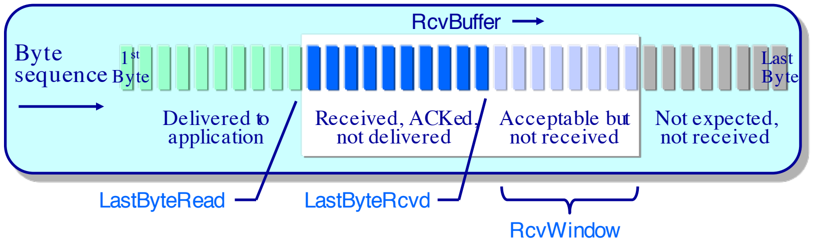

Receiver's buffer (RcvBuffer)
Byte sequence
1st Byte
Last Byte
Delivered to application
Received, ACKed, not delivered
Acceptable but not received
Not expected, not received

- The receiver explicitly informs the sender of the amount of free buffer space in `RcvBuffer`
  - » `RcvWindow` field in TCP segment
- The sender keeps the amount of transmitted, unACKed data less than most recently received `RcvWindow`



TCP Flow Control

Receiver window control



- The goal is to ensure:

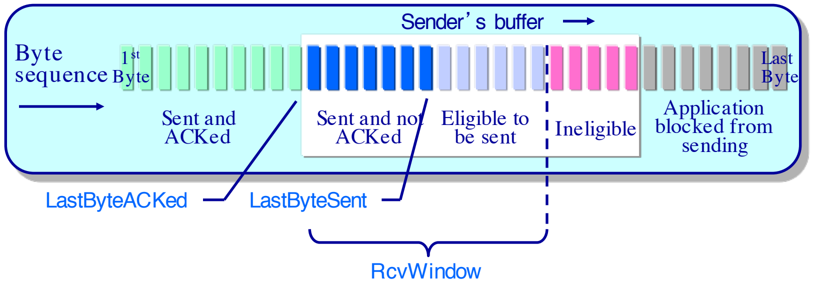
  LastByteRcvd - LastByteRead ≤ RcvBuffer

- Sender is sent:

  RcvWindow = RcvBuffer - (LastByteRcvd-LastByteRead)

TCP Flow Control

Sender Window control

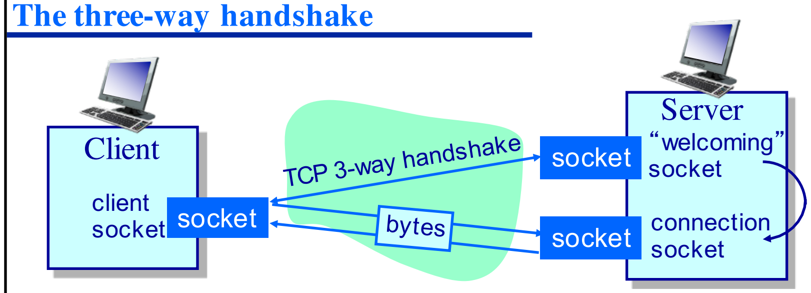


- The sender ensures:

  LastByteSent - LastByteACKed ≤ RcvWindow

TCP Connection Management

The three-way handshake



- TCP endpoints establish a "connection" before exchanging data segments
  - *client:* connection initiator

```
clientSocket = socket(AFNET, SOCK_STREAM)
clientSocket.connect(serverName, serverPort)
```

  - *server:* contacted by client

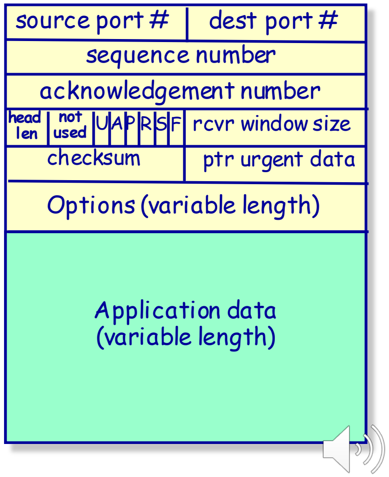
```
connectionSocket, addr = serverSocket.accept()
```

Client reaches out first

TCP Connection Management

The three-way handshake

- Client sends SYN segment to server
  - The SYN specifies the client's initial sequence number
  - The ACK number in the SYN will be 0

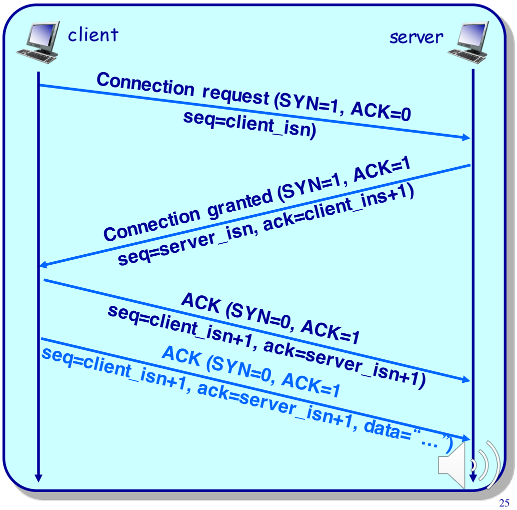


source port #	dest port #
sequence number	
acknowledgement number	
head len | not used | U | A | P | R | S | F	rcvr window size
checksum	ptr urgent data
Options (variable length)	
Application data (variable length)	

TCP Connection Management

The three-way handshake

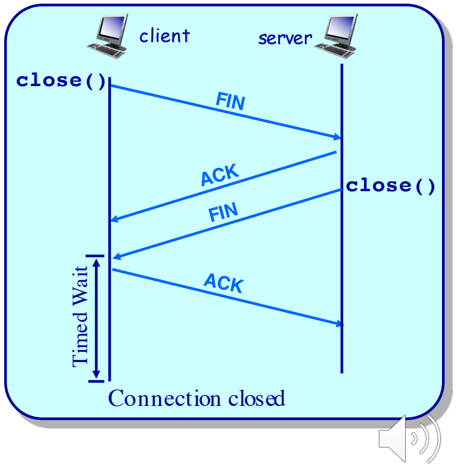
- Client sends SYN segment to server
  - » The SYN specifies the client's initial sequence number
- Server receives SYN, replies with SYN+ACK segment
  - » ACKs received SYN
  - » Allocates buffers
  - » Specifies server's initial sequence number
- Third segment may be an ACK only or an ACK+data


client
server
Connection request (SYN=1, ACK=0
seq=client_isn)
Connection granted (SYN=1, ACK=1
seq=server_isn, ack=client_ins+1)
ACK (SYN=0, ACK=1
seq=client_isn+1, ack=server_isn+1)
ACK (SYN=0, ACK=1
seq=client_isn+1, ack=server_isn+1, data="…")

TCP Connection Management

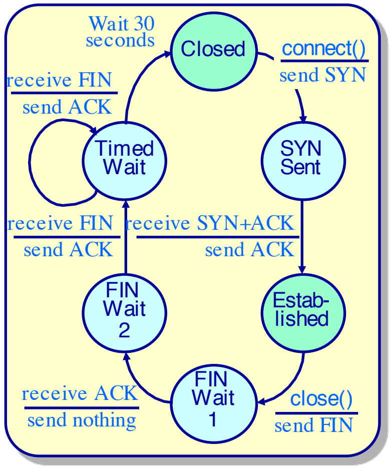
Closing a connection

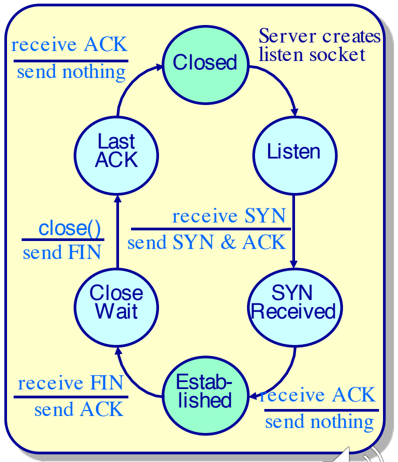
- Client sends FIN segment to server
- Server receives FIN, replies with ACK
  - » Server closes connection, sends FIN
- Client receives FIN, replies with ACK
- Client enters "timed wait" state
  - » Client will ACK any received FIN



TCP Connection Management

Client/Server lifecycles





- TCP client lifecycle
- TCP server lifecycle