Distributed Systems

5. Transport Protocols

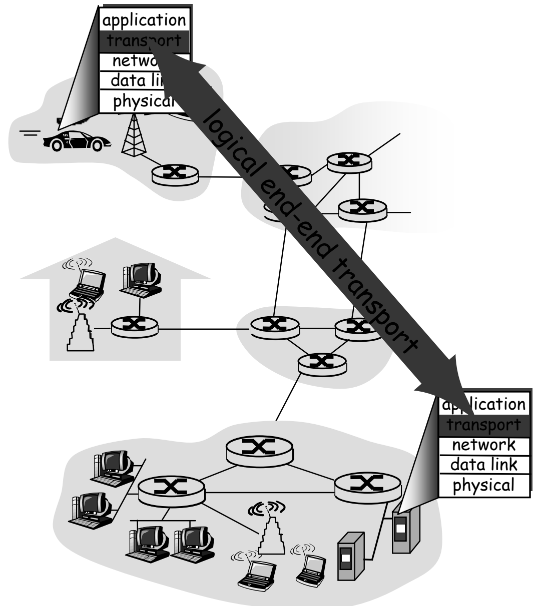
Werner Nutt

5. Transport Protocols

5.1 Transport-layer Services

Transport Services and Protocols

- Provide *communication* between application processes running on different hosts
- Transport protocols run in end systems
  – send side: breaks application messages into segments, passes to network layer
  – receive side: reassembles segments into messages, passes to application layer
- Two transport protocol available to Internet applications
  – TCP and UDP



Transport vs. Network Layer

- *Network layer:* communication between hosts
- *Transport layer:* communication between processes
  – relies on, enhances, network layer services

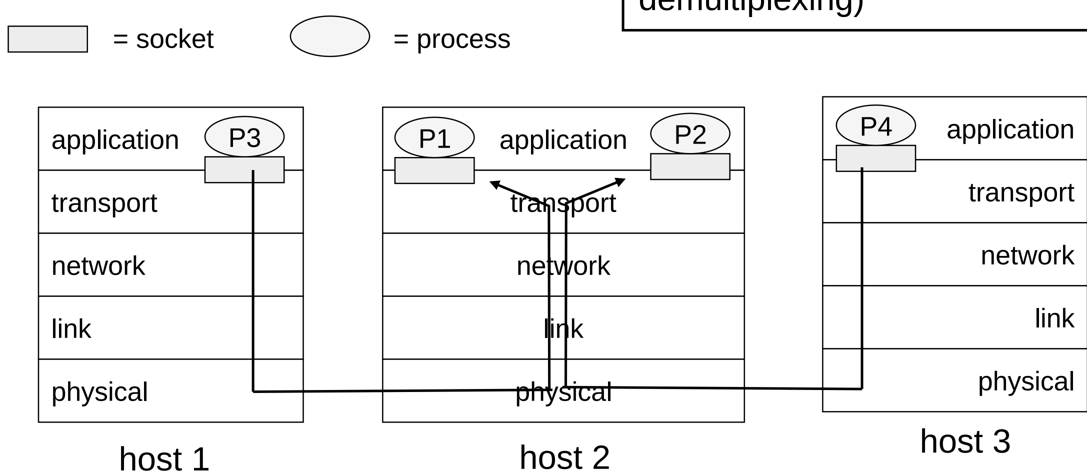
5. Transport Protocols

5.2 Multiplexing and Demultiplexing

Multiplexing/Demultiplexing

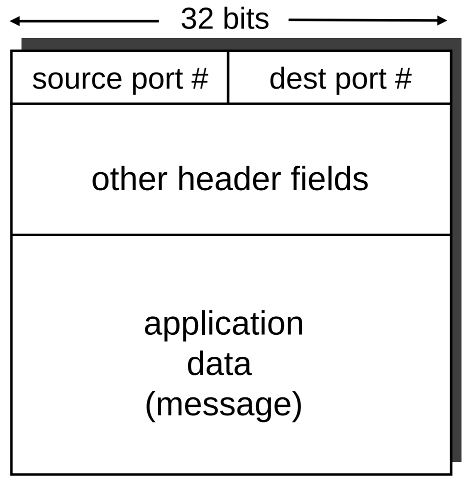
Demultiplexing at receive host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How Demultiplexing Works

- Host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- Host uses IP addresses and port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless Demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 =
  new DatagramSocket(12534);
DatagramSocket mySocket2 =
  new DatagramSocket(12535);
```

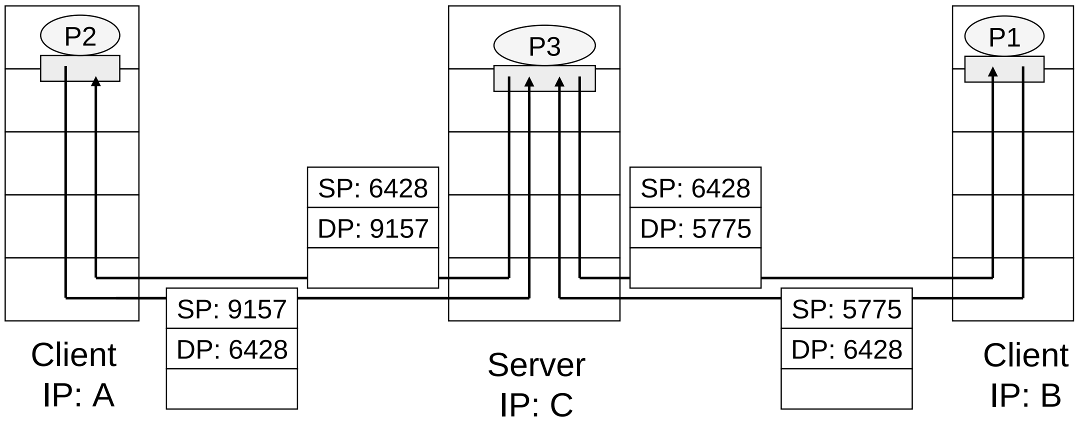
- UDP socket identified by 2-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless Demultiplexing (cntd)

```
DatagramSocket serverSocket =
                         new DatagramSocket(6428);
```



SP provides “return address”

Connection-oriented Demultiplexing

- A TCP socket is identified by a 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Receiving host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have a different socket for each request

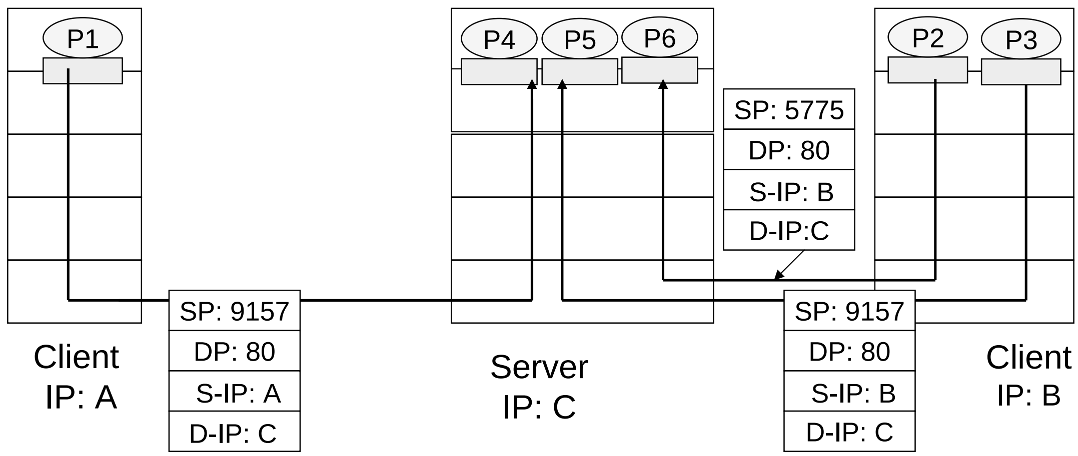
Exercise: In Firefox, type

`about:config`

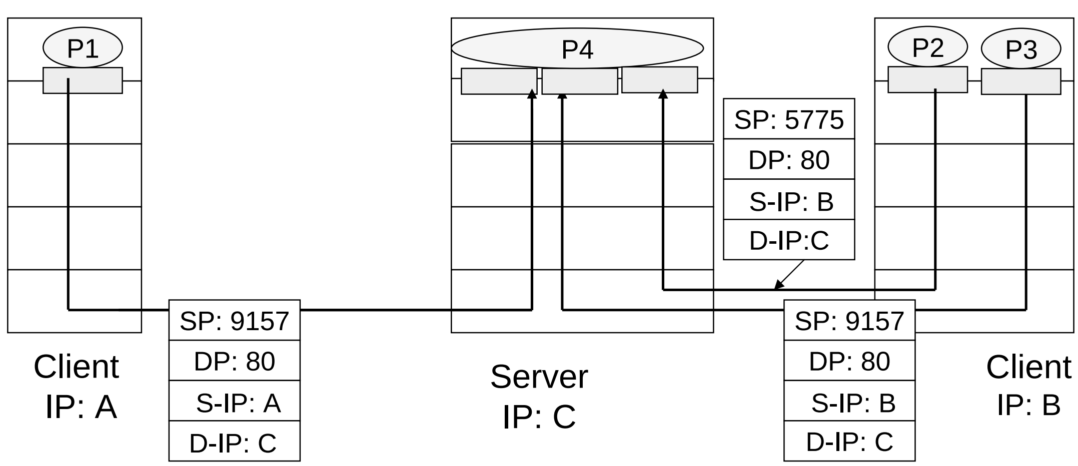
and check out

`network.http.max-connections-per-server`

Connection-oriented Demultiplexing (cntd)



Connection-oriented Demultiplexing: Threaded Server



5. Transport Protocols

5.3 Connectionless Transport: UDP

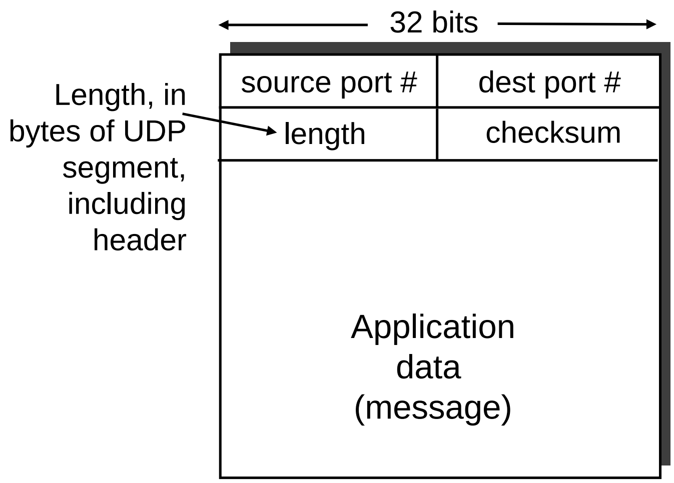
UDP: User Datagram Protocol [RFC 768]

- "No frills" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - lost
  - delivered out of order to application
- *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
- no congestion control: UDP can blast away as fast as desired

UDP Segment Format



UDP Checksum

Goal: Detect “errors” (e.g., flipped bits) in transmitted segment

Sender:

- Treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?*

Internet Checksum Example

- Note

  when adding numbers, a carry from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

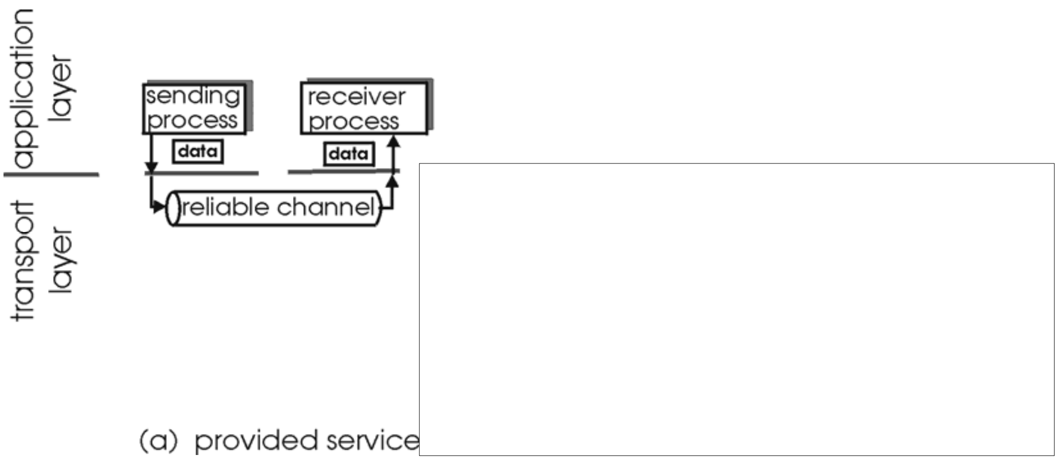
	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum	1	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	1	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

5. Transport Protocols

5.4 Principles of Reliable Data Transfer

Principles of Reliable Data Transfer

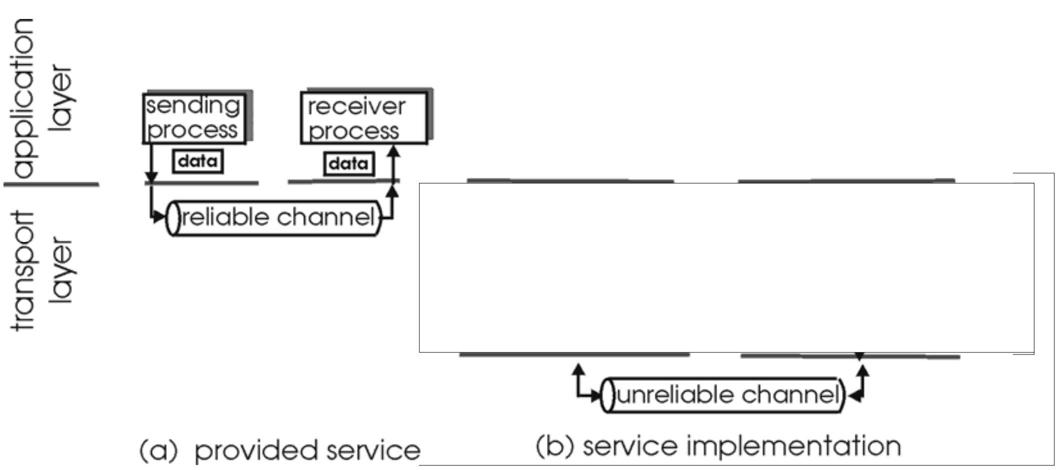
- Important in application, transport, data link layers



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (RDT)

Principles of Reliable Data Transfer

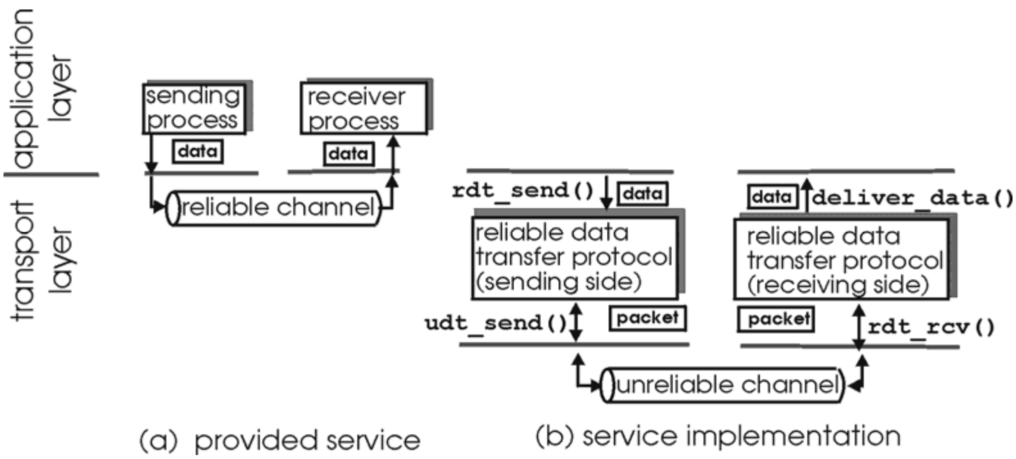
- Important in application, transport, data link layers



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (RDT)

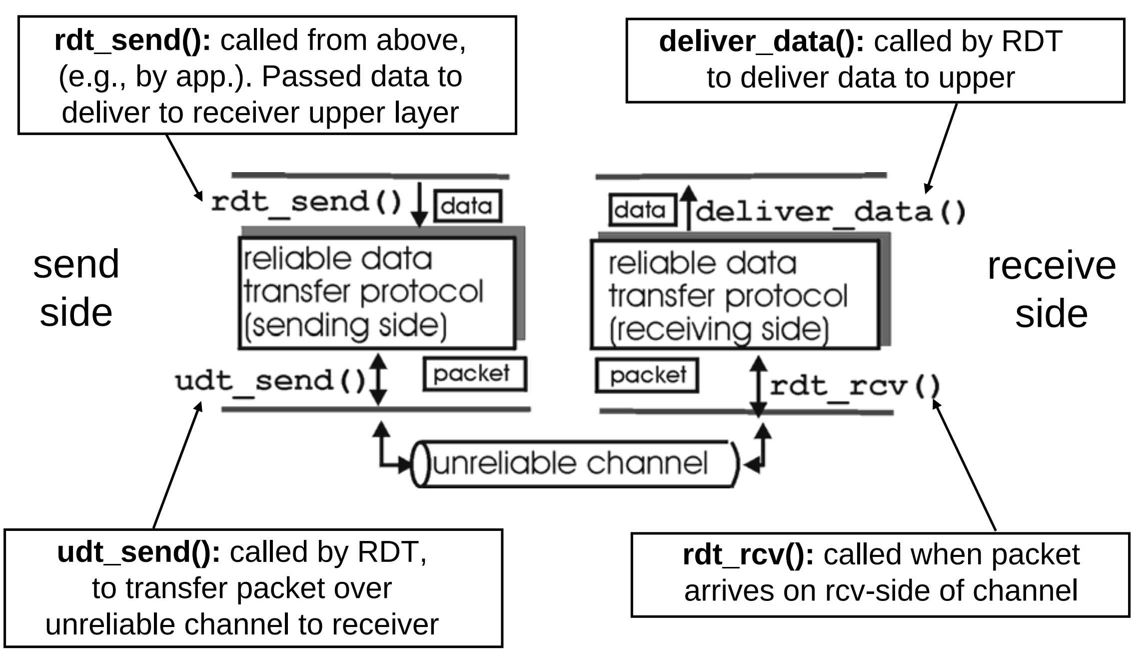
Principles of Reliable Data Transfer

- Important in application, transport, data link layers



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (RDT)

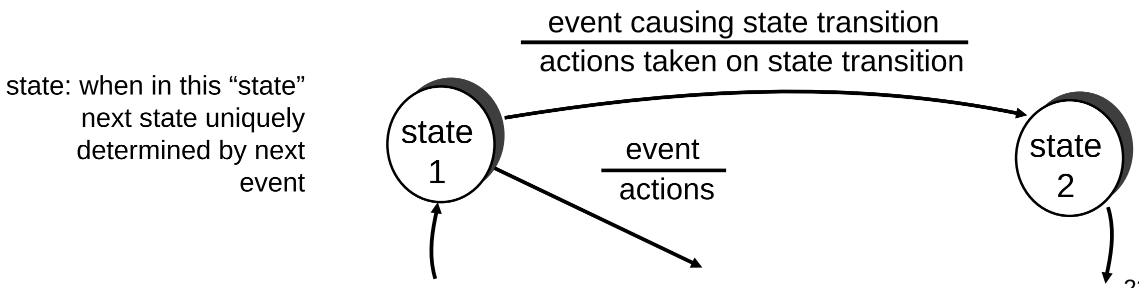
Reliable Data Transfer: Getting Started



Reliable Data Transfer: Getting Started

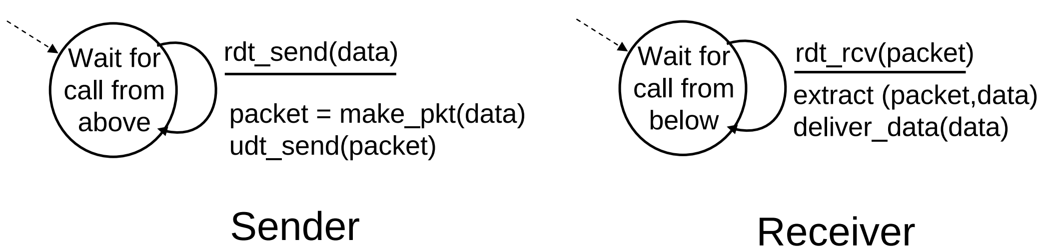
We will:

- incrementally develop the sender, receiver sides of a reliable data transfer protocol (RDT)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



RDT1.0: Reliable Transfer over a Reliable Channel

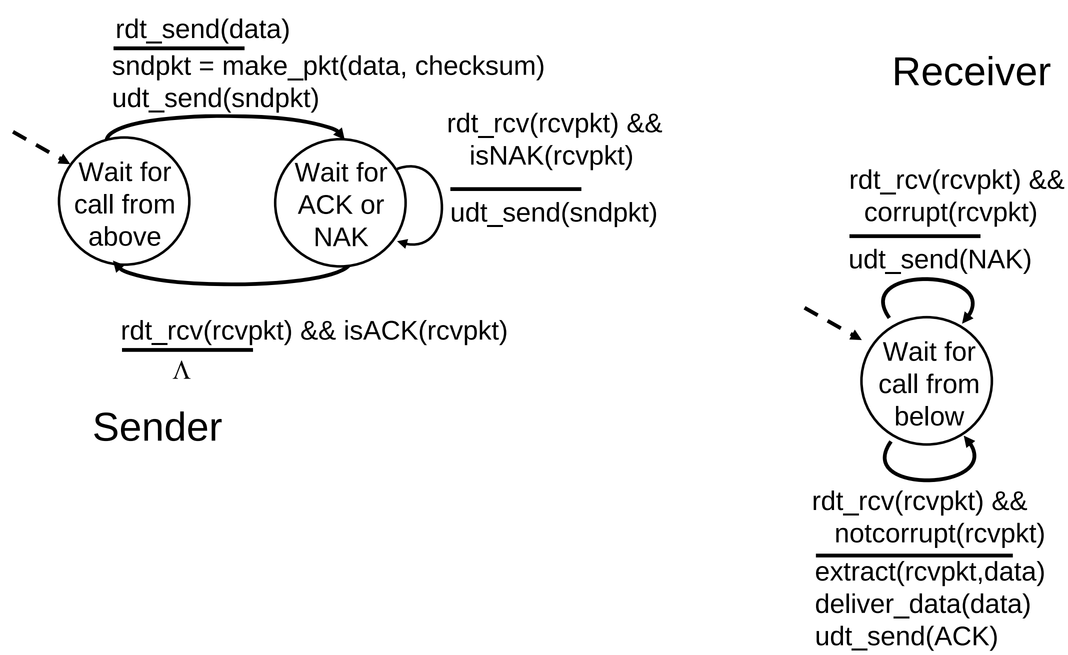
- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



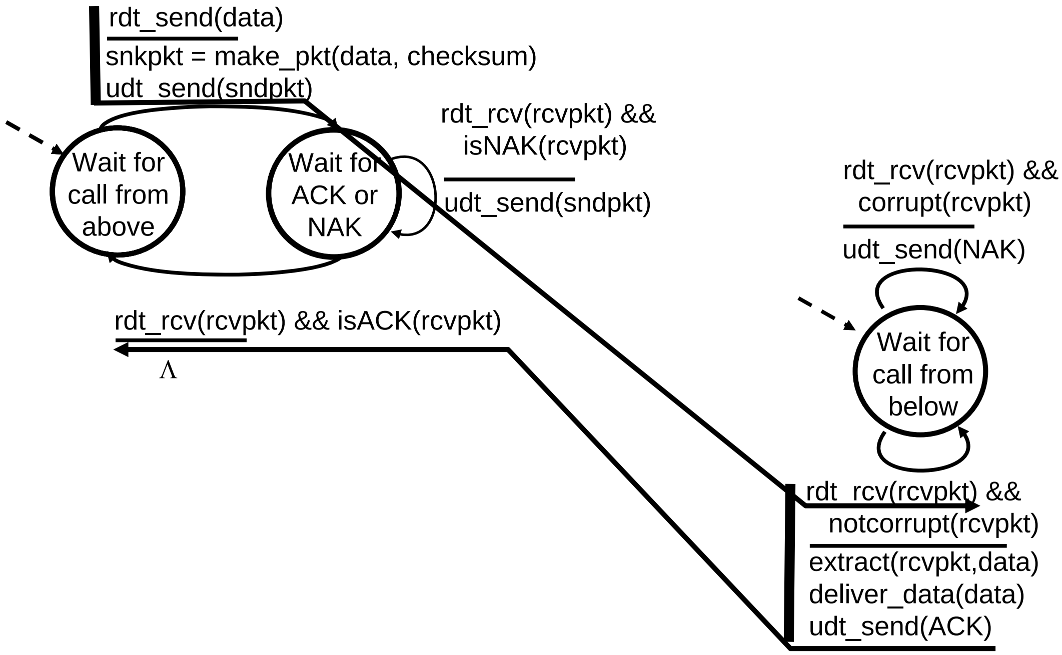
RDT2.0: Channel with Bit Errors

- Underlying channel may flip bits in packet
  - *checksum* to detect bit errors
- *The* question: how to *recover* from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet was received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK
- New mechanisms in **RDT2.0** (beyond **RDT1.0**):
  - error detection
  - receiver feedback: control messages (ACK,NAK) receiver $\rightarrow$ sender

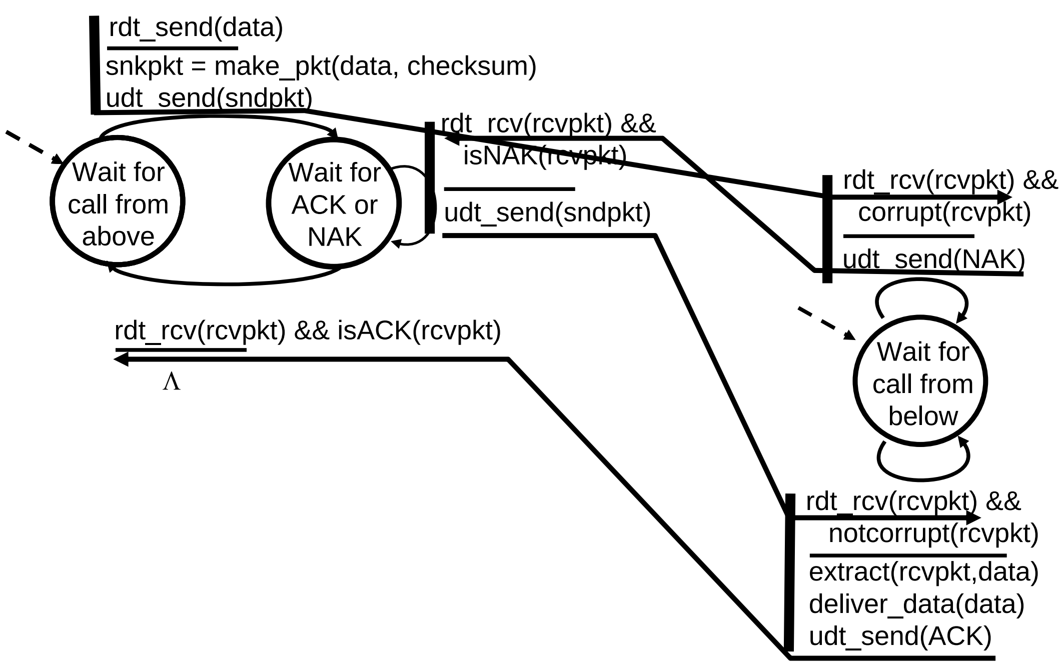
RDT2.0: FSM Specification



RDT2.0: Operation without Errors



RDT2.0: Error Scenario



RDT2.0 Has a Fatal Flaw!

- What happens if ACK/NAK is corrupted?
- Sender doesn't know what happened at the receiver!
- It can't just retransmit: possible duplicate

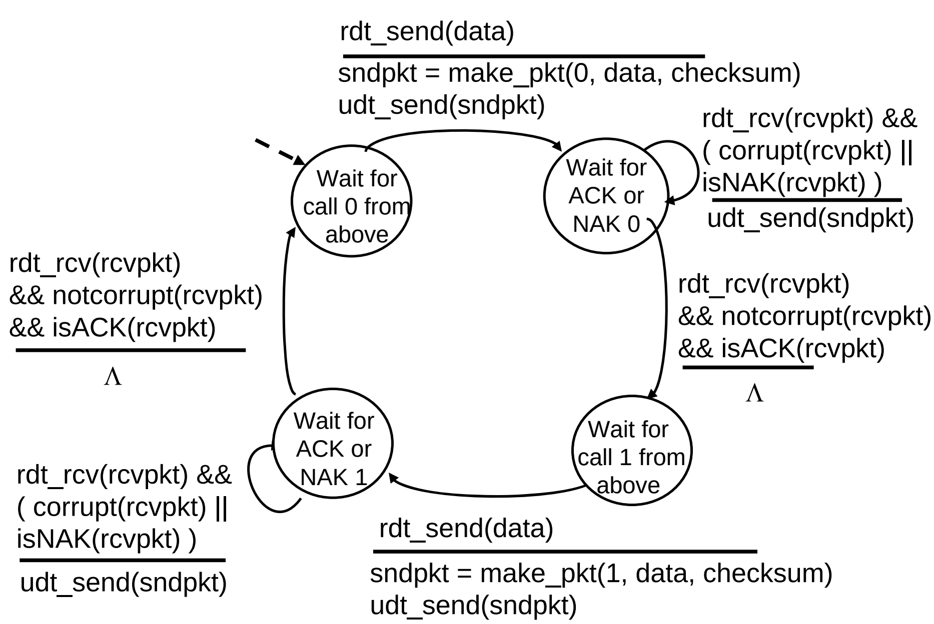
Handling duplicates:

- Sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

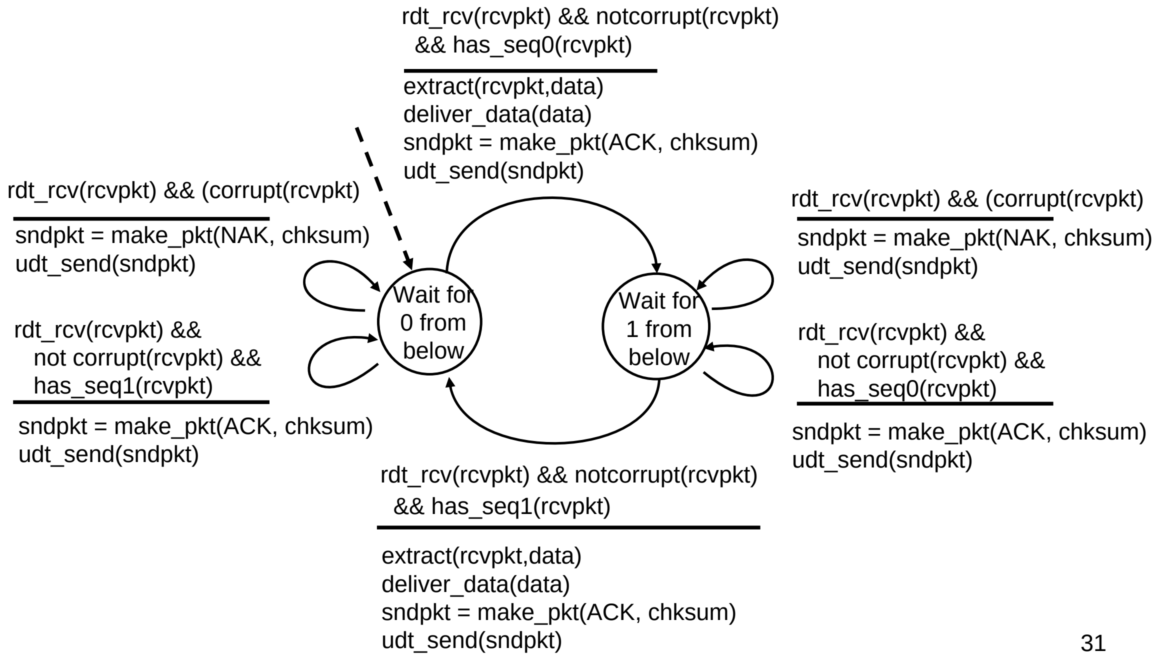
"Stop and Wait" Protocol

Sender sends one packet, then waits for receiver response

RDT2.1: Sender, Handles Corrupted ACK/NAKs



RDT2.1: Receiver, Handles Corrupted ACK/NAKs



RDT2.1: Discussion

Sender:

- Sequence # added to packet
- Two sequence #'s (0,1) will suffice. *Why?*
- Must check if received ACK/NAK corrupted
- Twice as many states
  - – state must "remember" whether "current" packet has sequence# 0 or 1

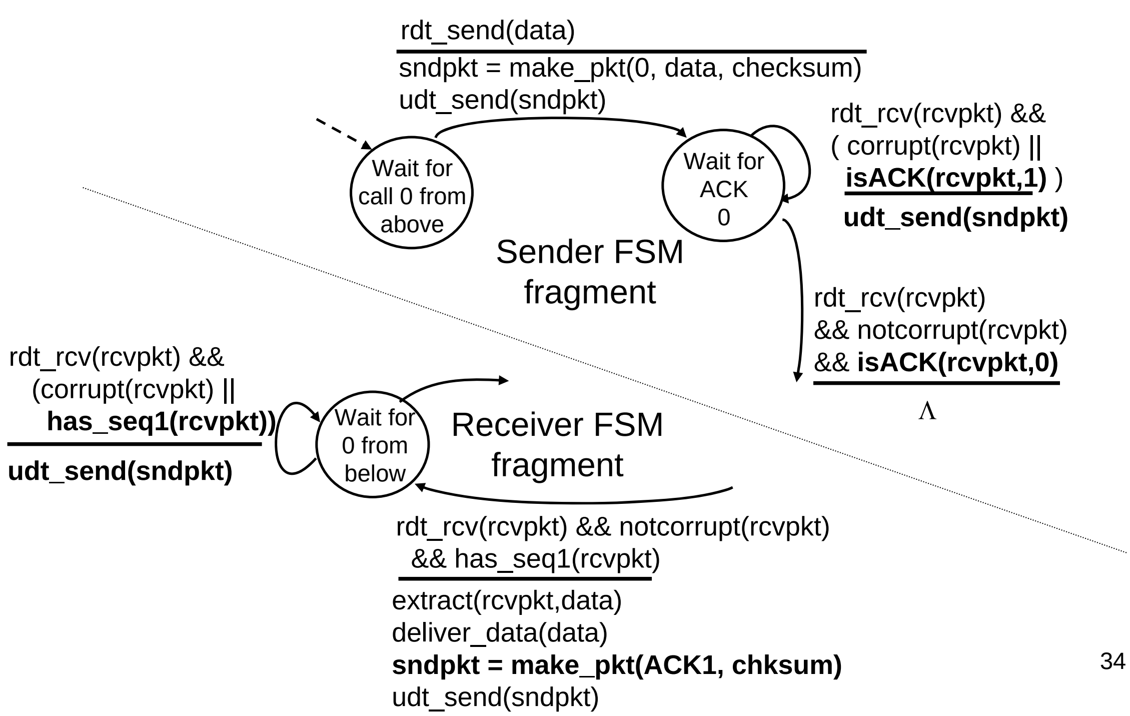
Receiver:

- Must check if received packet is duplicate
  - – state indicates whether 0 or 1 is expected packet sequence #
- Note: receiver *cannot* know if sender received its last ACK/NAK OK

RDT2.2: A Protocol w/o NAK

- Same functionality as RDT2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last packet received OK
  - receiver must *explicitly* include seq# of packet being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current packet*

RDT2.2: Sender, Receiver (Fragments)



RDT3.0: Channels with Errors *and* Loss

New assumption:

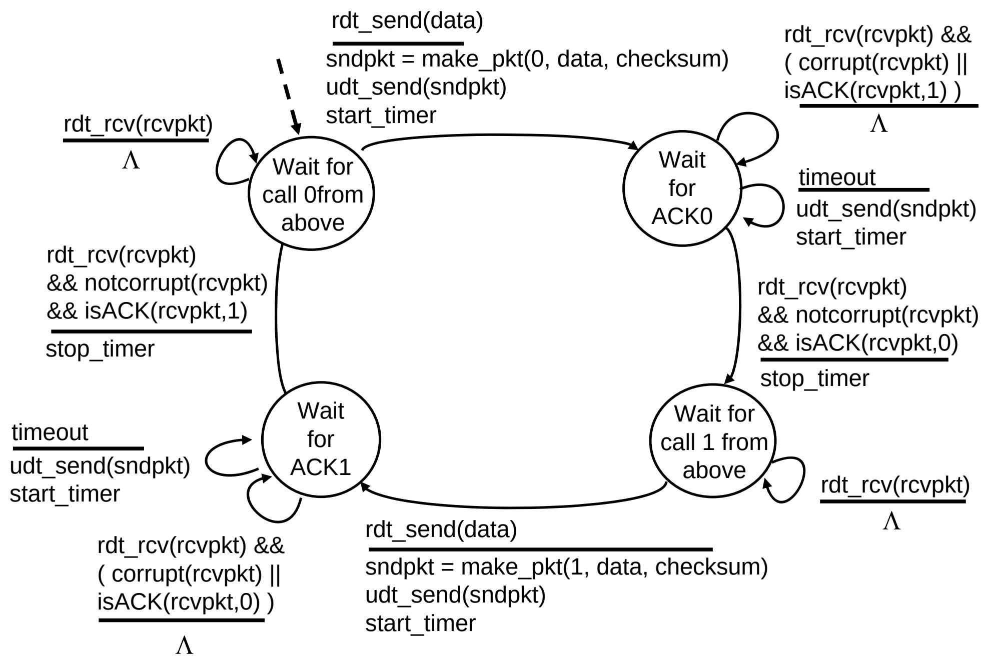
- Underlying channel can also lose packets (data or ACKs):
  - checksum
  - sequence #s
  - ACKs
  - retransmissions

  will be of help, but not enough

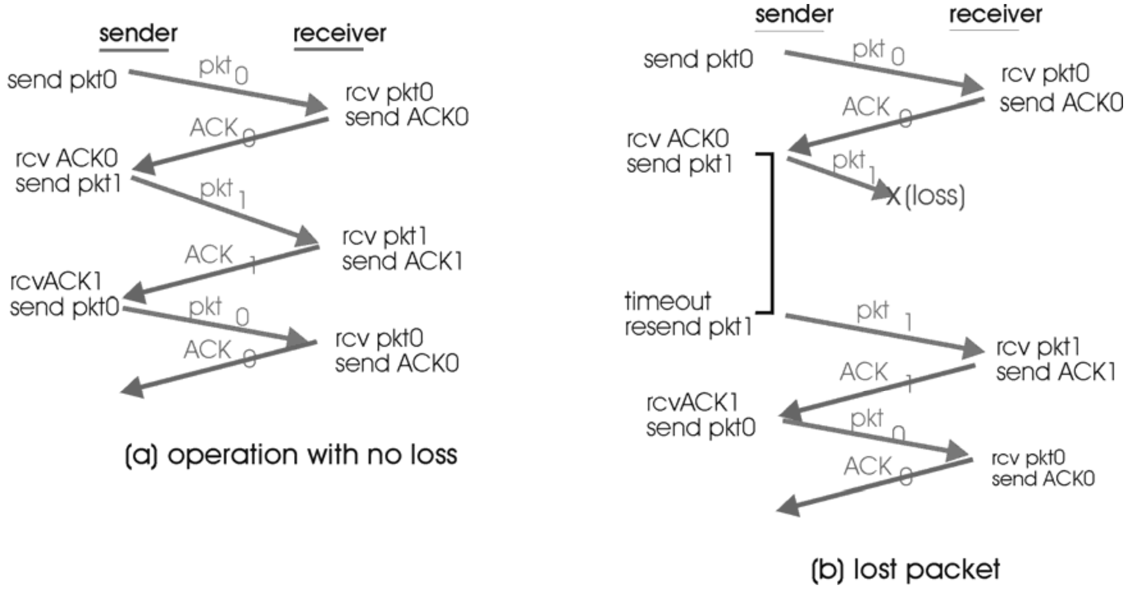
Approach:

- Sender waits "reasonable" amount of time for ACK
- Retransmits if no ACK received in this time
- If packet (or ACK) is just delayed (not lost):
  - retransmission will be duplicate, but use of seq #'s already handles this
  - receiver must specify seq # of packet being ACKed
- Requires countdown timer

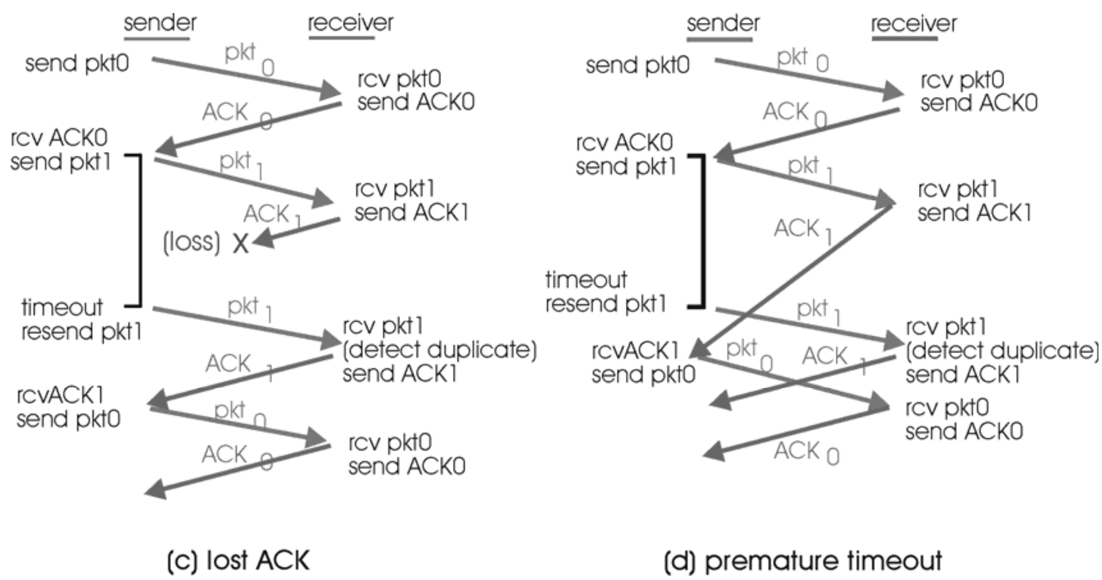
RDT3.0 Sender



RDT3.0 in Action



RDT3.0 in Action



Performance of RDT3.0

- RDT3.0 works, but performance is poor
- Example: 1 Gbps link, 15 ms propagation delay, 8000 bit packet:

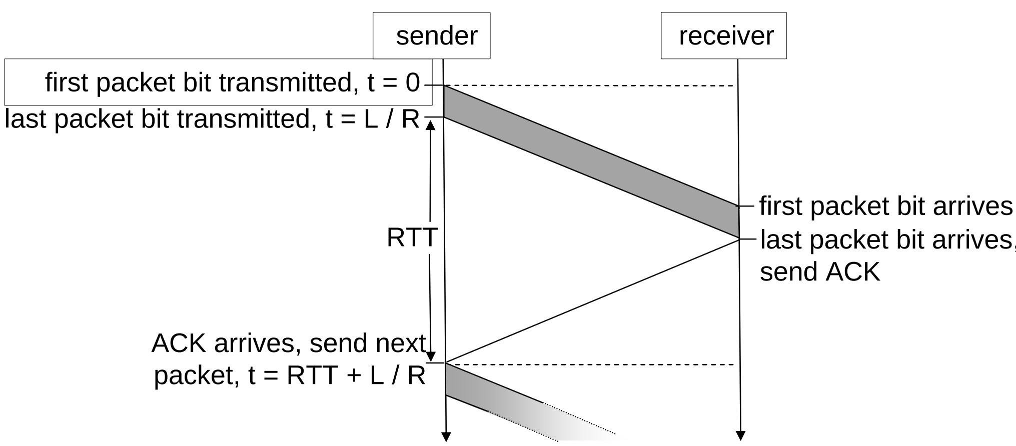
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\text{microseconds}$$

U_{sender}: utilization – fraction of time sender is busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- 1KB packet every 30 msec
  $\rightarrow$ 33KB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!

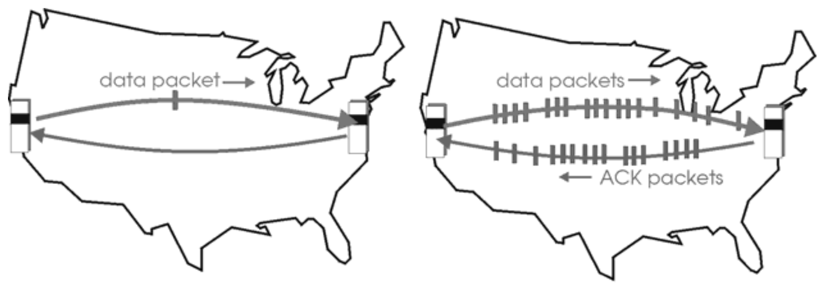
RDT3.0: Stop-and-wait Operation



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined Protocols

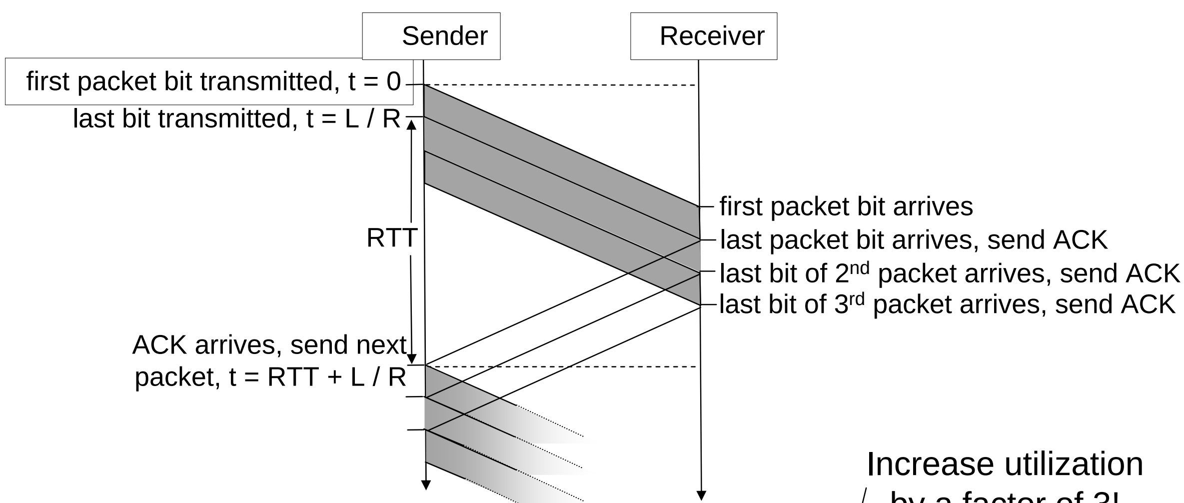
- Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *Go-Back-N* and *Selective Repeat*

Pipelining: Increased Utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Pipelining Protocols

Go-back-N: Overview

- *Sender:* up to N unACKed packets in pipeline
- *Receiver:* only sends cumulative ACKs
  - does not ACK packet if there is a gap
- *Sender:* has timer for oldest unACKed packet
  - if timer expires: retransmit all unACKed packets

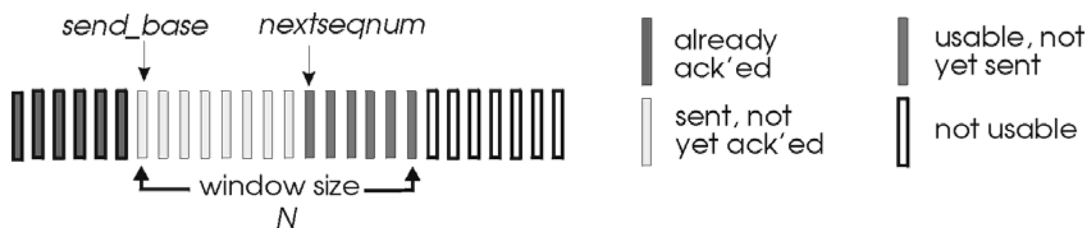
Selective Repeat: Overview

- *Sender:* up to N unACKed packets in pipeline
- *Receiver:* ACKs individual pkts
- *Sender:* maintains timer for each unACKed packet
  - if timer expires: retransmit only unACKed packet

Go-Back-N

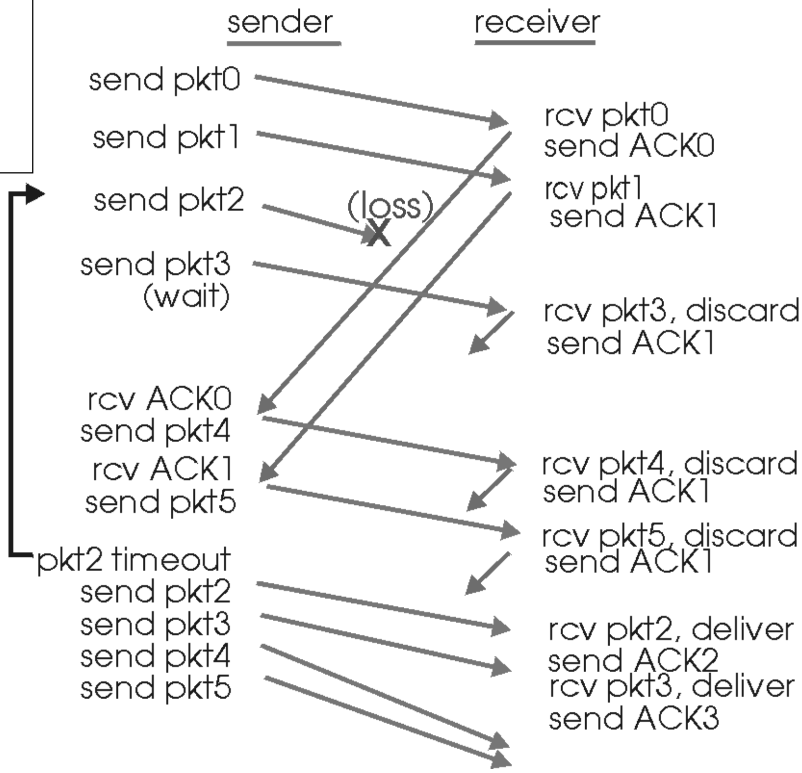
Sender:

- k-bit sequence # in packet header
- "window" of up to N, consecutive unACKed packets allowed



- ACK(n): ACKs all packets up to, including seq # n — "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- *Timeout(n):* retransmit packet n and all higher seq # packets in window

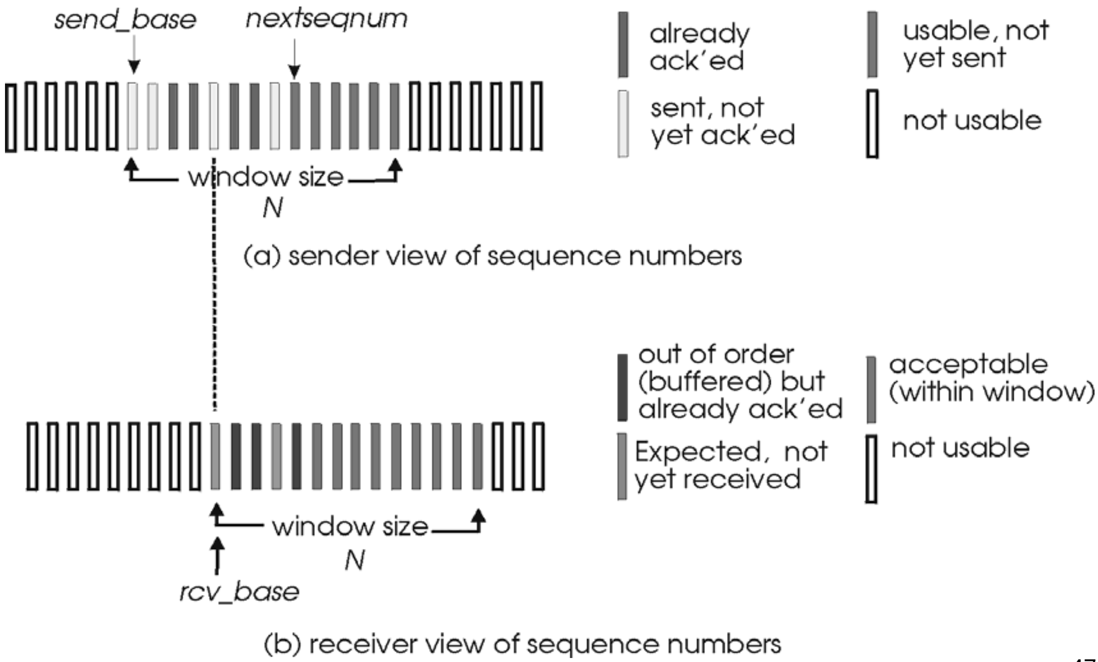
Go-Back-N in Action



Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- Sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective Repeat: Sender, Receiver Windows



Selective Repeat

Sender

Data from application (above):

- if next available seq # in window, send pkt

Timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

Receiver

Pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

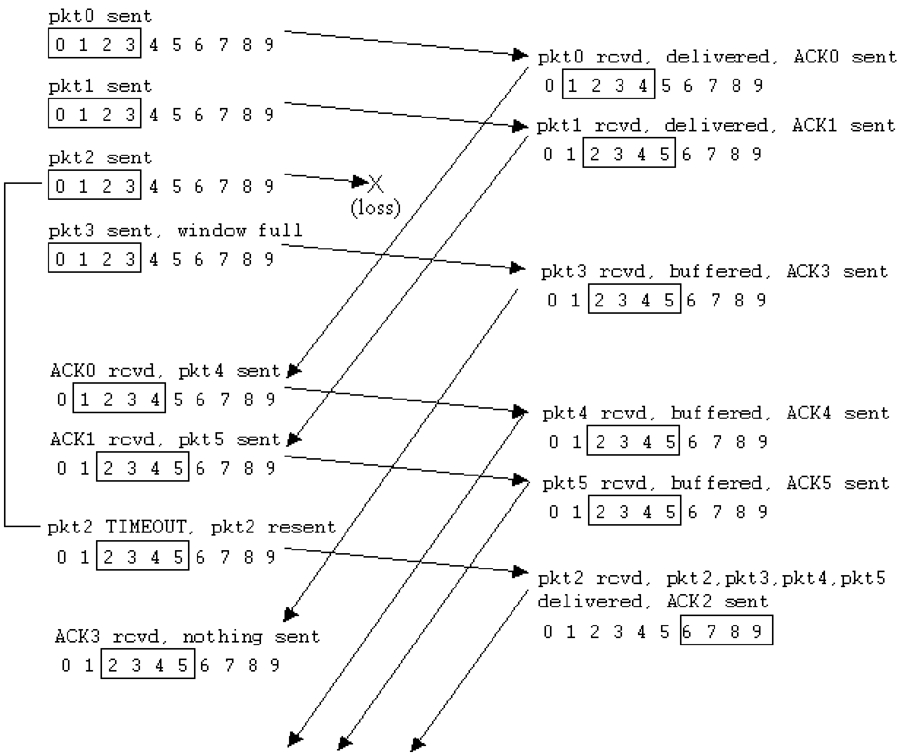
Pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

Otherwise:

- ignore

Selective Repeat in Action



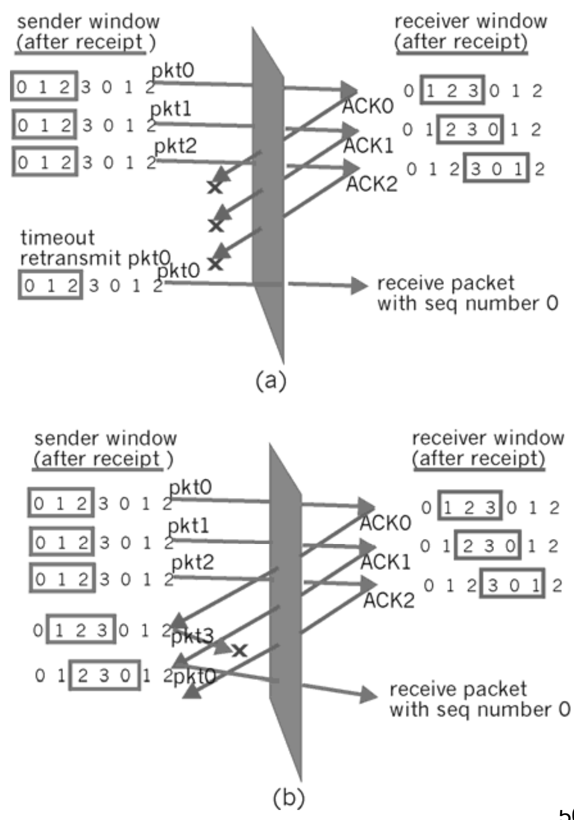
Selective Repeat: Dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- Receiver sees no difference in two scenarios!
- Incorrectly passes duplicate data as new in (a)

Q: what relationship should hold between seq # size and window size?



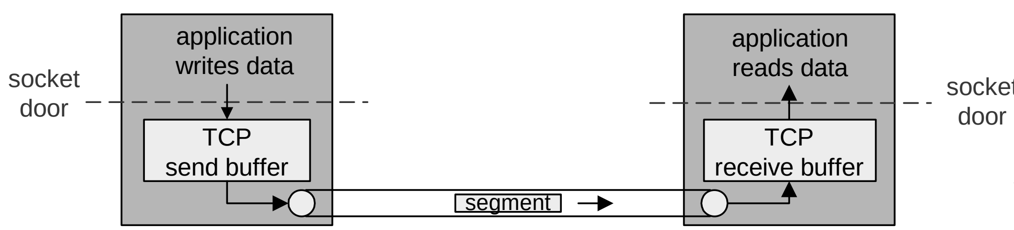
5. Transport Protocols

5.5 Connection-oriented Transport: TCP

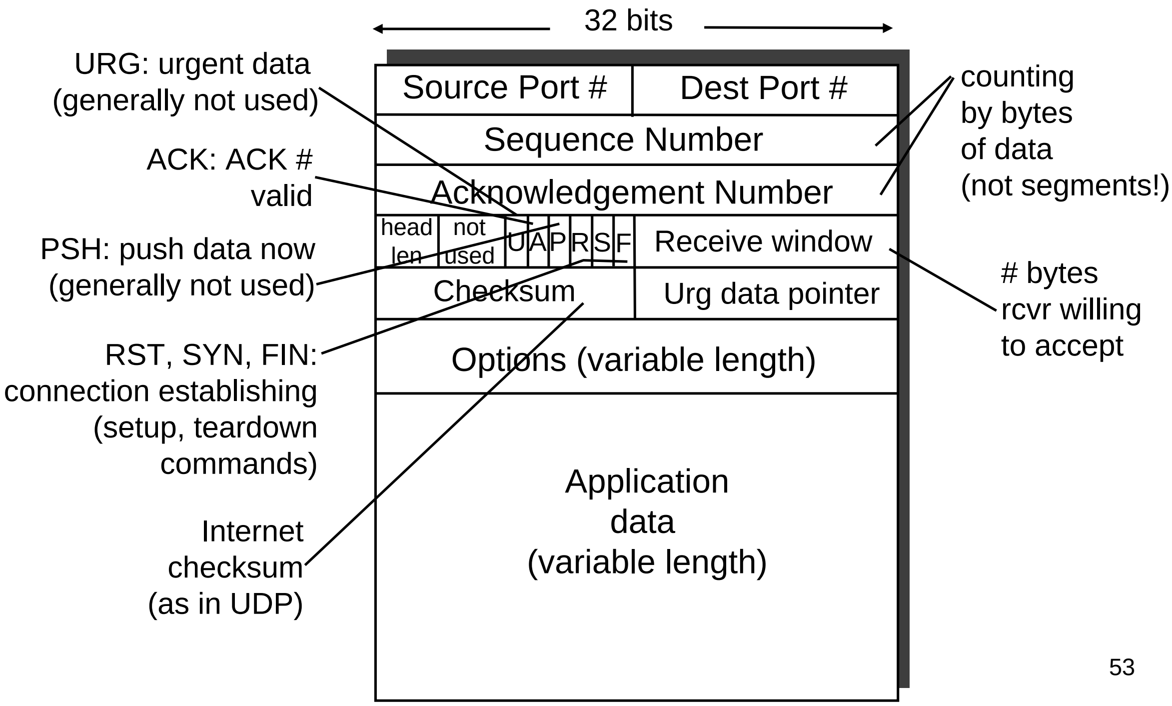
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - one sender, one receiver
- Reliable, in-order *byte steam:*
  - no “message boundaries”
- Pipelined:
  - TCP congestion and flow control set window size
- *Send & receive buffers*
- Flow controlled:
  - sender will not overwhelm receiver
- Full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-oriented:
  - handshaking (exchange of control msgs) initialises sender, receiver state before data exchange

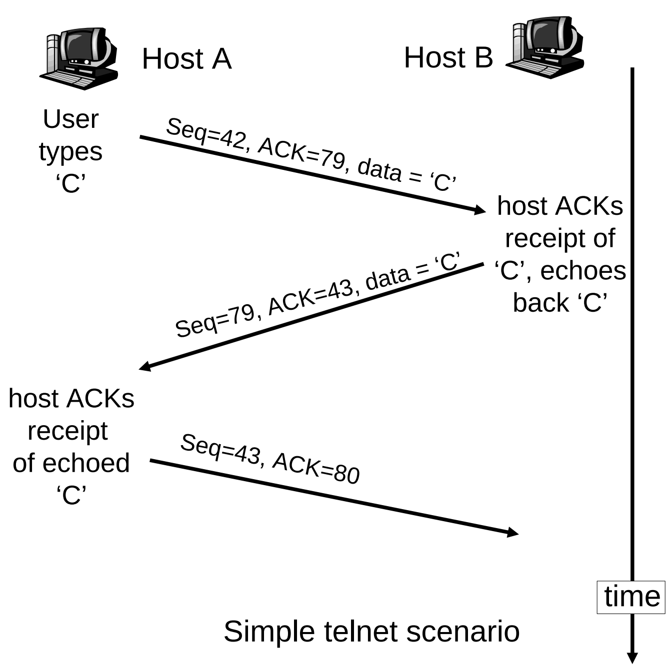


TCP Segment Structure



TCP Sequence #'s and ACKs

- 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 implementer



Simple telnet scenario

TCP Round Trip Time and Timeout

Q: How to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: How to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

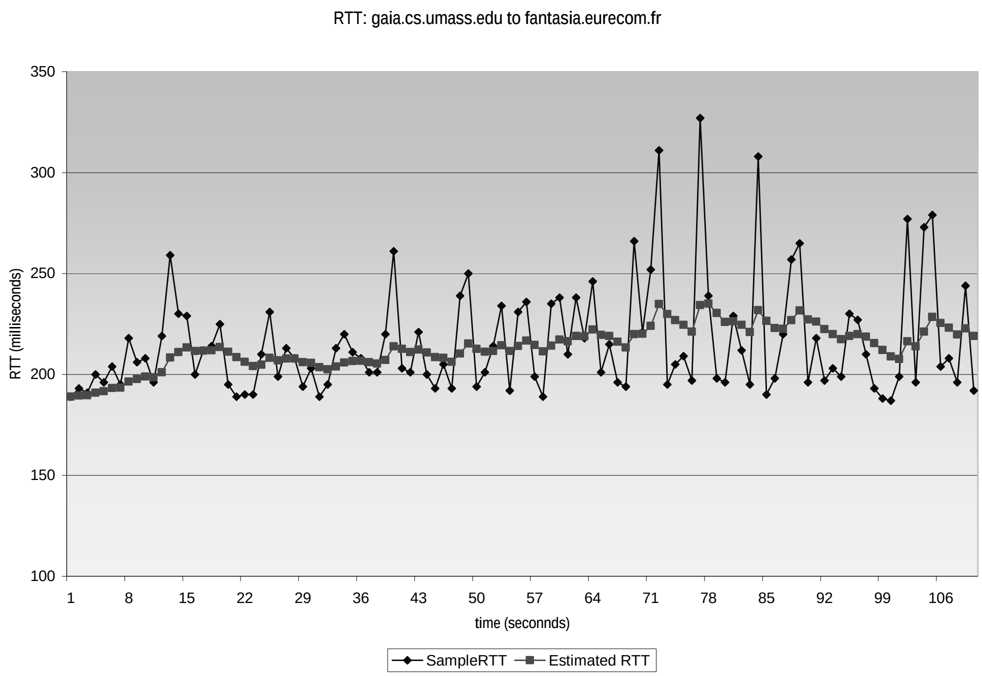
TCP Round Trip Time and Timeout

$$\textbf{EstimatedRTT} = (1-\alpha) * \textbf{EstimatedRTT} + \alpha * \textbf{SampleRTT}$$

Exponential weighted moving average

- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

Example RTT Estimation



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** $\rightarrow$ larger safety margin
- First, estimate of how much SampleRTT deviates from **`EstimatedRTT`**:

```
DevRTT = (1-β)* DevRTT +
             β * |SampleRTT - EstimatedRTT|
```

(typically, β = 0.25)

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP Reliable Data Transfer

- TCP creates RDT service on top of IP's unreliable service
- TCP features
  – pipelined segments
  – cumulative ACKs
- TCP uses *by default* single retransmission timer
- Retransmissions are triggered by:
  – timeout events
  – duplicate ACKs
- Consider simplified TCP sender:
  – ignore congestion control

TCP Sequence Numbers

Sequence number of a segment:

Byte stream number of first byte in segment

Example: A sends to B over TCP

- 500k image with MSS = 1k,
  initial sequence number = 0

➔ 500 segments,
with sequence numbers 0, 1024, 2048, ...

TCP Acknowledgement Numbers

Acknowledgement number in segment sent from B to A:
Sequence number of next byte B is expecting from A

Example:

- B has received segments 1, 2, and 4, but not 3.
- Acknowledgement number is 2048
  (= 1^{st} byte of segment 3)

Example shows:

- Acknowledgement is cumulative
  (acknowledges all bytes up to Ack - 1)
- No mention of out-of-order segments

TCP Sender Events:

Data received from application:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unACKed segment)
- expiration interval: **`TimeOutInterval`**

Timeout:

- retransmit segment that caused timeout
- restart timer

ACK received:

- if acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are outstanding segments

TCP Sender Actions

Client variables

```
ackSNo  = initialSequenceNumber  // ack'ed sequence #
nextSNo = initialSequenceNumber // next sequence #
```

Loop through the following cases:

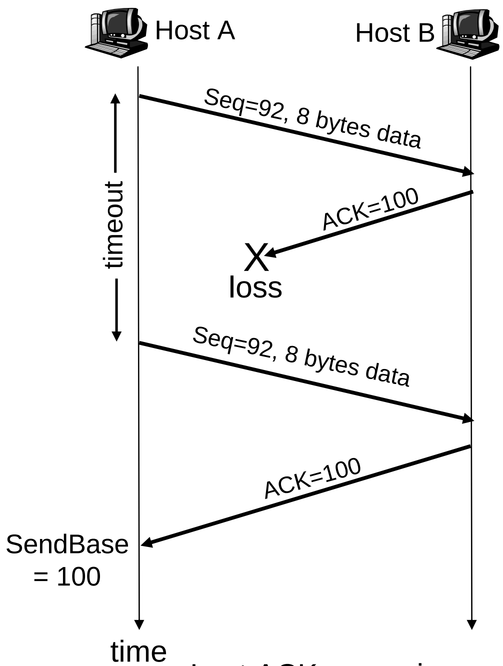
```
if (data received from application){
   create segment with sequence number nextSNo;
   start timer for segment nextSNo;
   pass segment to IP;
   nextSNo = nextSNo + data.length}

if (timeout for segment with sNo y){
   retransmit segment y;
   restart timer for segment y}
```

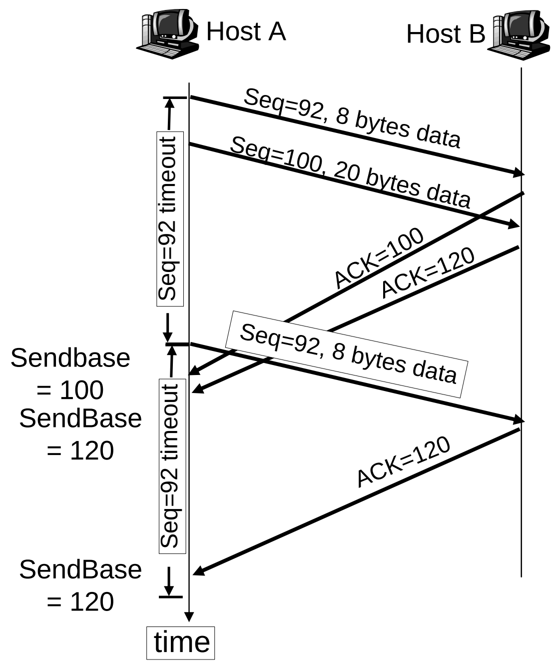

TCP Sender Actions (cntd)

```
if (ACK received with AckNo = y)
   if (y > ackSNo){ // cumulative ack
      cancel timers for segments with lower SNos;
      ackSNo = y}
   else { // duplicate ack
      increment counter for duplicate acks for y;
      if (number of duplicate acks for y == 3) {
         retransmit segment y;
         restart timer for segment y
         }
      }
```

TCP: Retransmission Scenarios

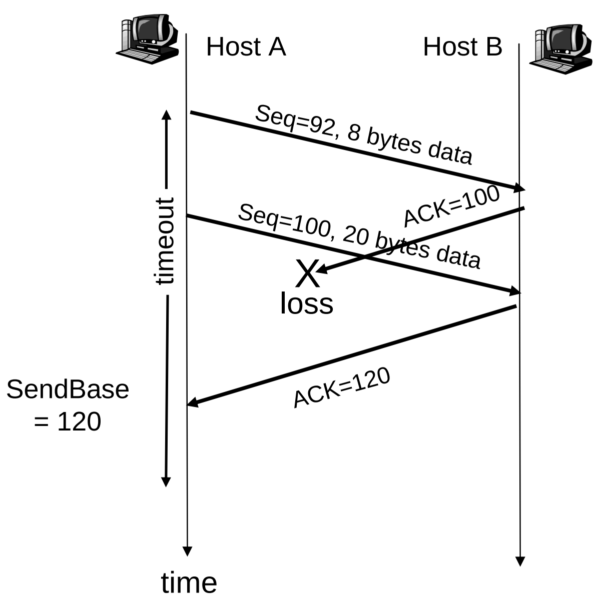


Lost ACK scenario



Premature timeout

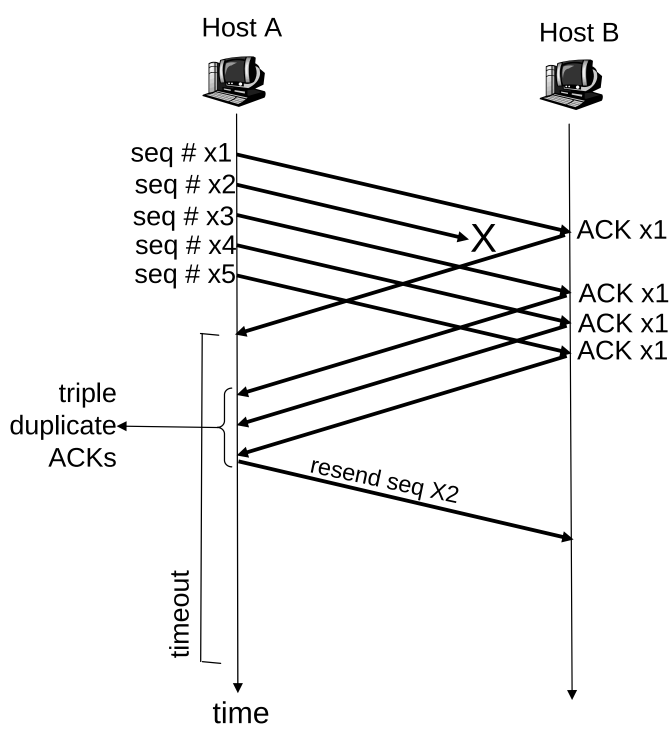
TCP retransmission scenarios (cntd)



Cumulative ACK scenario

Fast Retransmit

- Time-out period often relatively long:
  – long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  – sender often sends many segments back-to-back
  – if segment is lost, there will likely be many duplicate ACKs for that segment
- If sender receives 3 ACKs for same data, it assumes that the segment after ACKed data was lost:
  – fast retransmit: resend segment before timer expires



TCP Receiver Actions

Event	Action
▪ Segment arrives with expected SNo, all previous data already ack'ed	▪ Wait up to 500 ms for arrival of another segment. Then send ack
▪ Segment arrives with expected SNo, preceding segment received, but not ack'ed	▪ Send cumulative ack
▪ Out-of-order segment arrives with higher SNo than expected	▪ Send duplicate ack, indicating SNo of next expected byte
▪ Out-of-order segment arrives with lower SNo than expected	▪ Send duplicate ack, indicating SNo of next expected byte

Flow Control

Receiver's buffer has size **`RcvBuffer`**

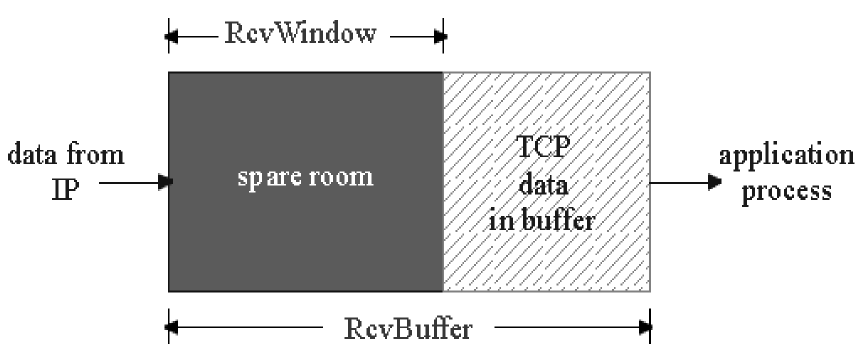
Receiver maintains variables

`LastByteRead`

`LastByteReceived`

Constraint:

`LastByteReceived - LastByteRead <= RcvBuffer`



Flow Control (cntd)

Receiver communicates to sender

```
RcvWindow =
      RcvBuffer - (LastByteReceived - LastByteRead)
```

Sender maintains variables

`LastByteSent`

`LastByteAcked`

Sender makes sure

```
LastByteSent - LastByteAcked <= RcvWindow
```


TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - sequence #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *Client:* connection initiator

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

- *Server:* contacted by client

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

Three Way Handshake

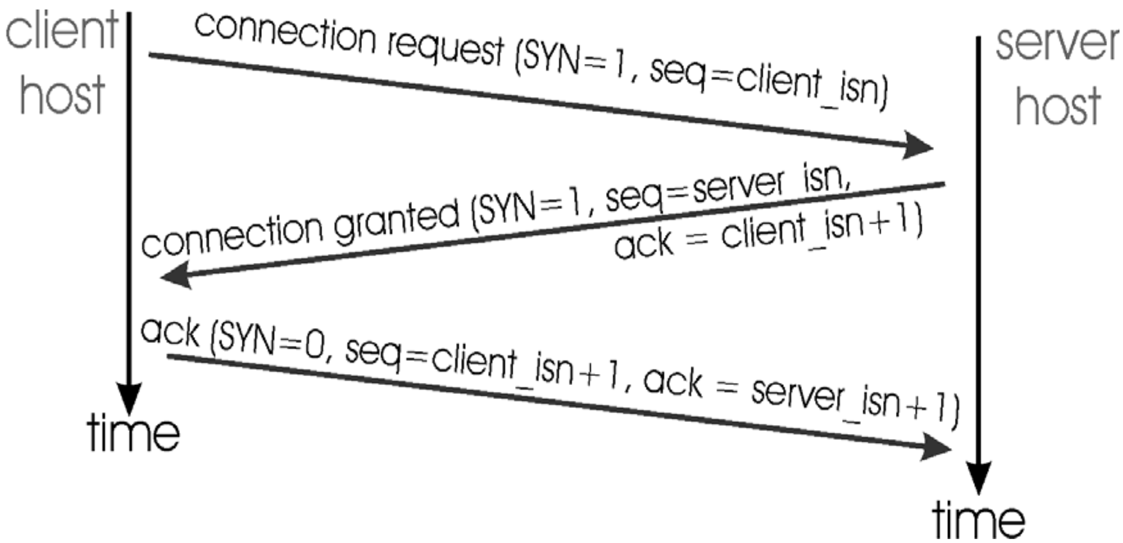
Step 1: client host sends TCP SYN segment to server
- specifies initial sequence #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial sequence #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

Establishing a TCP Connection

"Three way handshake"



Why are sequence numbers exchanged?
Why does the sender acknowledge?

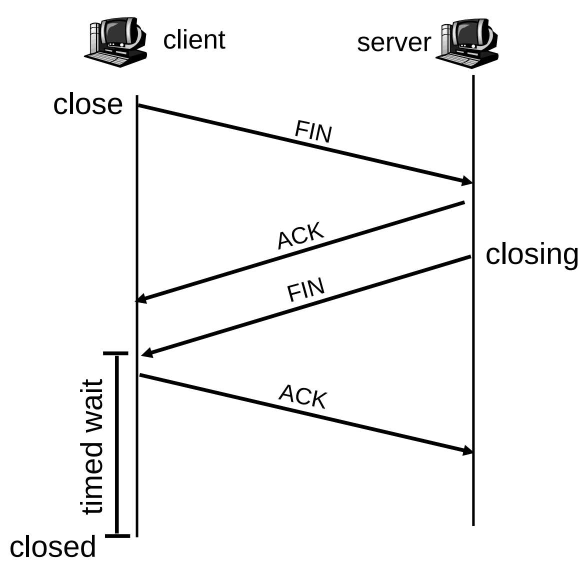
TCP Connection Management (cntd)

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.

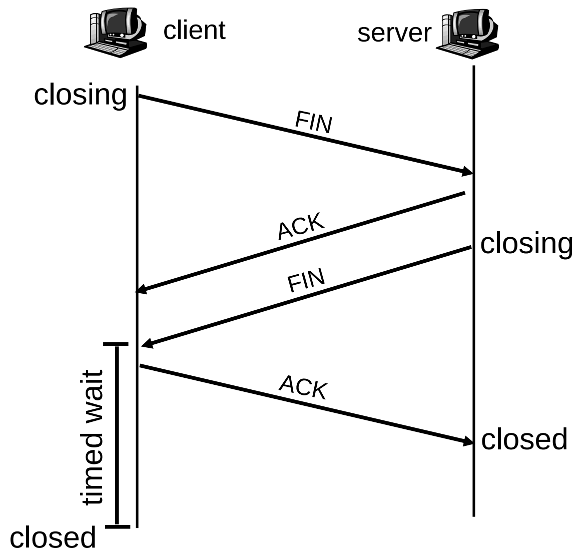


TCP Connection Management (cntd)

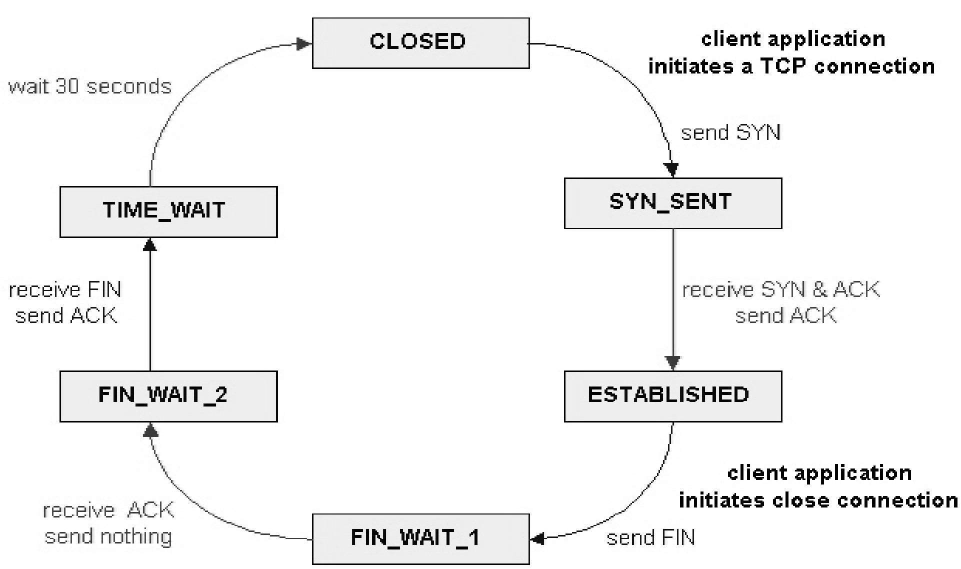
Step 3: client receives FIN, replies with ACK

– Enters "timed wait" - will respond with ACK to received FINs

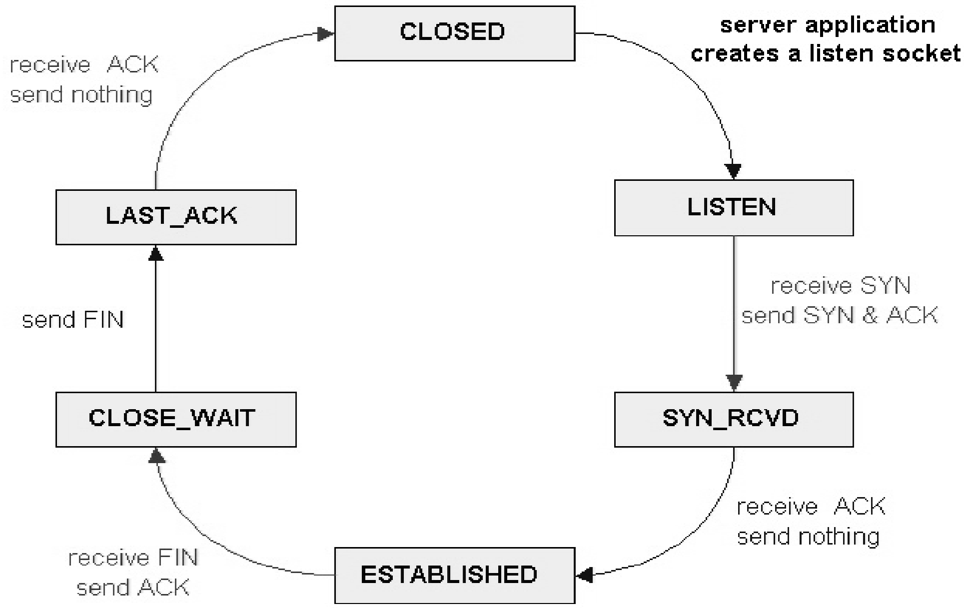
Step 4: server, receives ACK. Connection closed



TCP Life Cycle of a Client



TCP Life Cycle of a Server



References

The slides of this lecture are almost exclusively based on

Books:

- Kurose/Ross. Computer Networking: A Top-Down Approach

Slides:

- Kurose/Ross, Material for lecturers