Chapter 3
Transport Layer

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in powerpoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:
❑ If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
❑ If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach Featuring the Internet, 2nd edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2002.

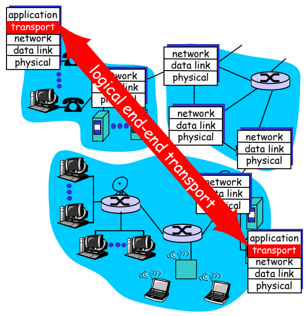
Chapter 3: Transport Layer

Our goals:
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

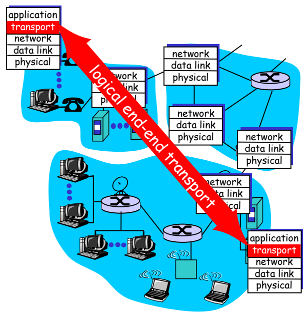
- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

Household analogy:

12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

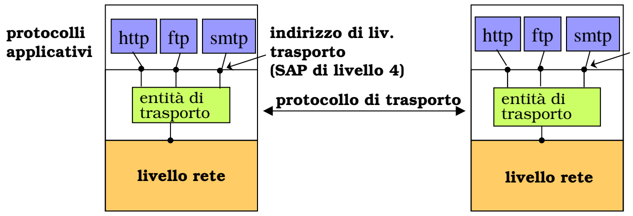


Chapter 3 outline

- 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

Servizio di trasporto

- Più applicazioni possono essere attive su un end system
  - il livello di trasporto svolge funzioni di multiplexing/demultiplexing
  - ciascun collegamento logico tra applicazioni è indirizzato dal livello di trasporto



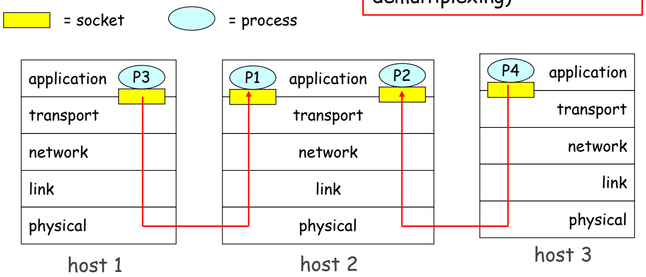
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

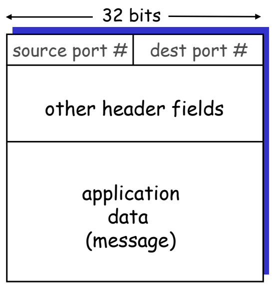
Multiplexing at send host:

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 (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

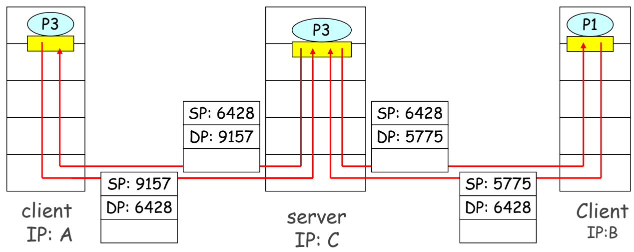
- UDP socket identified by two-tuple:

(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 demux (cont)

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

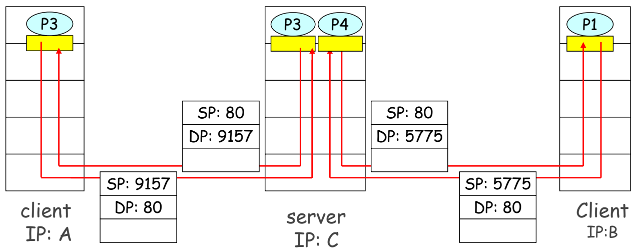


SP provides “return address”

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv 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 different socket for each request

Connection-oriented demux (cont)



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

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app

reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

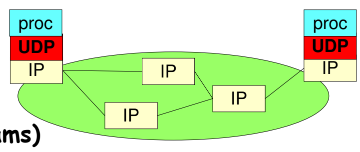
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive

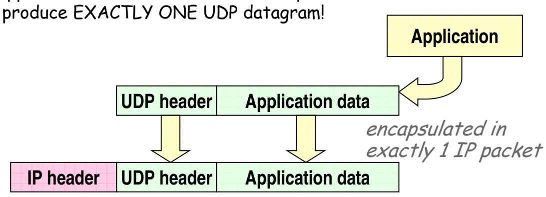
other UDP uses: DNS, SNMP..

UDP Packets

- **Connection-Less**
  - (no handshaking)



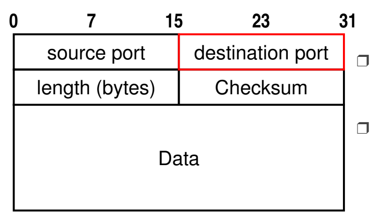
- **UDP packets (Datagrams)**
  - Each application interacts with UDP transport sw to produce EXACTLY ONE UDP datagram!



This is why, improperly, we use the term UDP packets

UDP datagram format

8 bytes header + variable payload



- **UDP length field**
  - all UDP datagram
  - (header + payload)
- **payload sizes allowed:**
  - Empty
  - even size (bytes)

➔**UDP functions limited to:**

⇨**addressing**

→which is the only strictly necessary role of a transport protocol

⇨**Error checking**

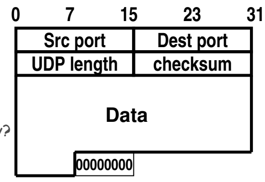
→which may even be **disabled** for performance

Maximum UDP datagram size

- 16 bit UDP length field:
  - Maximum up to 2^{16}-1 = 65535 bytes
  - Includes 8 bytes UDP header (max data = 65527)
- But max IP packet size is also 65535
  - Minus 20 bytes IP header, minus 8 bytes UDP header
  - Max UDP_data = **65507** bytes!
- Moreover, most OS impose further limitations!
  - most systems provide 8192 bytes maximum (max size in NFS)
  - some OS had (still have?) internal implementation features (bugs?) that limit IP packet size
    - SunOS 4.1.3 had 32767 for max tolerable IP packet transmittable (but 32786 in reception…) – bug fixed only in Solaris 2.2
- Finally, subnet Maximum Transfer Unit (MTU) limits may fragment datagram – annoying for reliability!
  - E.g. ethernet = 1500 bytes; PPP on your modem = 576

Error checksum

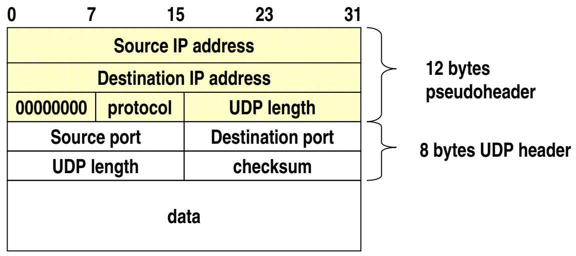
- 16 bit checksum field, obtained by:
  - summing up all 16 bit words in header data and **pseudoheader**, in 1's complement (checksum fields filled with 0s initially)
  - take 1's complement of result
  - if result is 0, set it to 111111…11 (65535==0 in 1's complement) Why?
  - Sender puts checksum value into UDP checksum field
- at destination:
  - 1's complement sum should return 0, otherwise error detected
  - upon error, no action (just packet discard)
- efficient implementation RFC 1071



- Zero padding
  - To multiple of 16 bits
- checksum disabled
  - by source, by setting 0 in the checksum field

Pseudo header

- Is not transmitted!
  - But it is information available at transmitter and at receiver
  - intention: double check that packet has arrived at correct destination



Protocol field (TCP=6,UDP=17) necessary, as same checksum calculation used in TCP. UDP length duplicated.

Q: NON SODDISFA IL PRINCIPIO DELLA SUDDIVISIONE IN LIVELLI?

disabling checksum

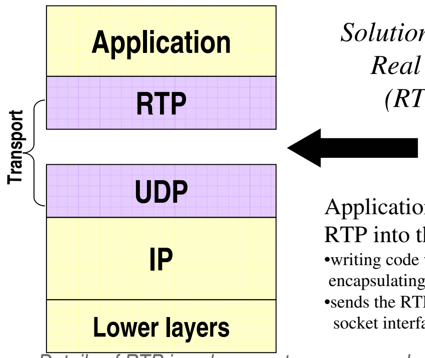
- In principle never!
  - Remember that IP packet checksum DOES NOT include packet payload.
- In practice, often done in NFS
  - sun was the first, to speed up implementation
- may be tolerable in LANs under one's control.
- Definitely dangerous in the wide internet
  - Exist layer 2 protocols without error checking

UDP: a lightweight protocol

- No connection establishment
  - no initial overhead due to handshaking
- No connection state
  - greater number of supported connections by a server!
- Small packet header overhead
  - 8 bytes only vs 20 in TCP
- originally intended for simple applications, oriented to short information exchange
  - DNS
  - management (e.g. SNMP)
  - etc
- No rate limitations
  - No throttling due to congestion & flow control mechanisms
  - No retransmission (for certain application loss tolerable)
- extremely important features for today multimedia applications! Expecially for real time applications which can tolerate some packet loss but require a minimum send rate.

RTP as seen from Application

Be careful: UDP ok for multimedia because it does not provide anything at all (no features = no limits!). Application developers have to provide supplementary transport capabilities at the application layer!



Solution for audio/video:
Real Time Protocol
(RTP, RFC 1889)

SOCKET INTERFACE

Application developer integrates RTP into the application by:
- writing code which creates the RTP encapsulating packets;
- sends the RTP packets into a UDP socket interface.

Details of RTP in subsequent courses – unless we are ahead of schedule

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

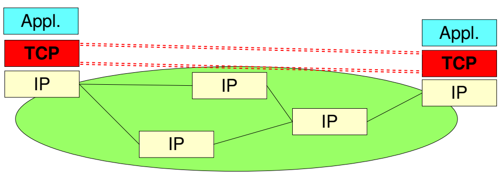
A MUCH more complex transport
for three main reasons

- Connection oriented
  - implements mechanisms to setup and tear down a full duplex connection between end points
- Reliable
  - implements mechanisms to guarantee error free and ordered delivery of information
- Flow & Congestion controlled
  - implements mechanisms to control traffic

TCP services

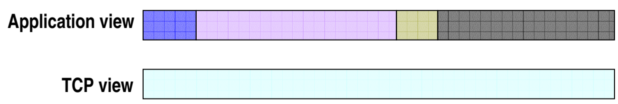
- connection oriented
  - TCP connections
- *reliable* transfer service
  - all bytes sent are received

→TCP functions
- → application addressing (ports)
- → error recovery (acks and retransmission)
- → reordering (sequence numbers)
- → flow control
- → congestion control



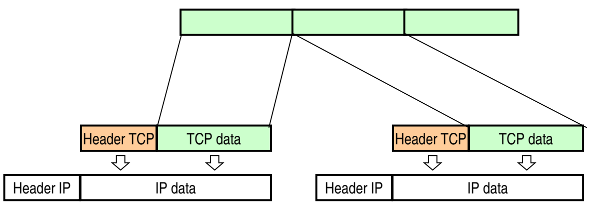
Byte stream service

- TCP exchange data between applications as a stream of bytes.
- It does not introduce any data delimiter (an application duty)
  - source application may enter 10 bytes followed by 1 and 40 (grouped with some semantics)
  - data is buffered at source, and transmitted
  - at receiver, may be read in the sequence 25 bytes, 22 bytes and 4 bytes...

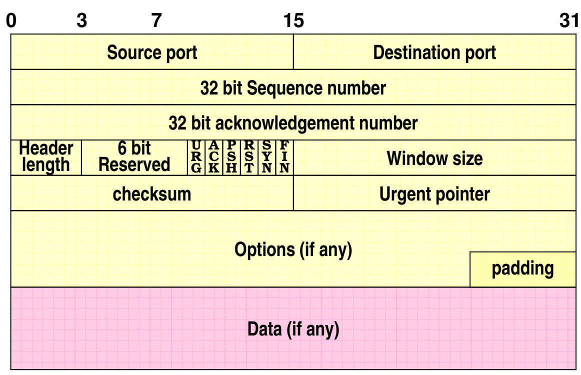


TCP segments

- Application data broken into segments for transmission
- segmentation totally up to TCP, according to what TCP considers being the best strategy
- each segment placed into an IP packet
- very different from UDP!!



TCP segment format
20 bytes header (minimum)



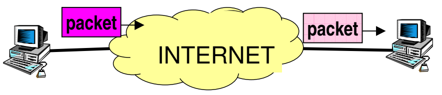
Source port			Destination port
32 bit Sequence number			
32 bit acknowledgement number			
Header length	6 bit Reserved	URG ACK PSH RST SYN FIN	Window size
checksum			Urgent pointer

- Source & destination port + source and destination IP addresses
  - univocally determine TCP connection
- checksum as in UDP
  - same calculation including same pseudoheader
- no explicit segment length specification

Source port			Destination port
32 bit Sequence number			
32 bit acknowledgement number			
Header length	6 bit Reserved	URG ACK PSH RST SYN FIN	Window size
checksum			Urgent pointer
Options (if any)			00000000

- Header length: 4 bits
  - specifies the header size (n*4byte words) for options
  - maximum header size: 60 (15*4)
  - option field size must be multiple of 32bits: zero padding when not.
- Reserved: 000000 (still today!)

Reliable data transfer: issues



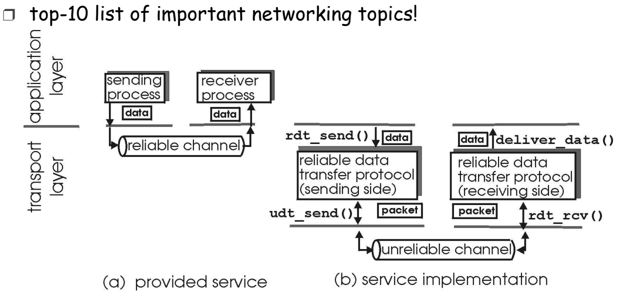
PROBLEMS:
1) Packet received with errors
2) Packet not received at all

Same problem considered at DATA LINK LAYER
(although it is less likely that a whole packet is lost at data link)

- mechanisms to guarantee correct reception:
  - Forward Error Correction (FEC) coding schemes
    - Powerful to correct bits affected by error, not effective in case of packet loss
    - Mostly used at link layer
  - Error detection (e.g. checksum used in UDP)
  - Retransmission – issues:
    - ACK
    - NACK
    - TIMEOUT

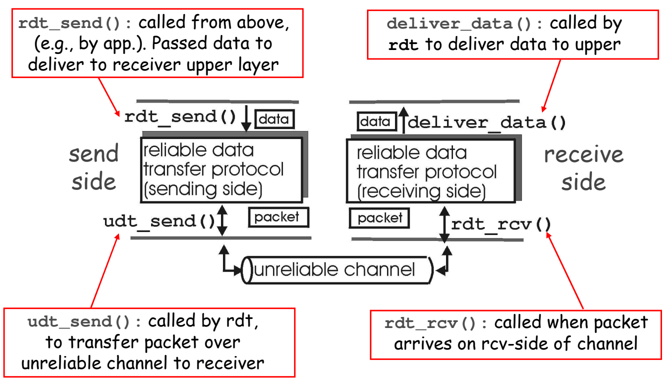
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



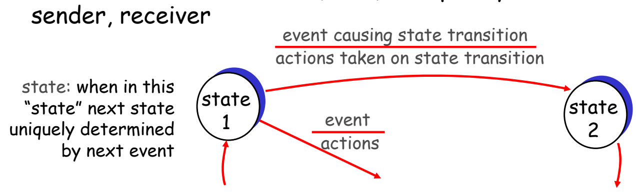
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started



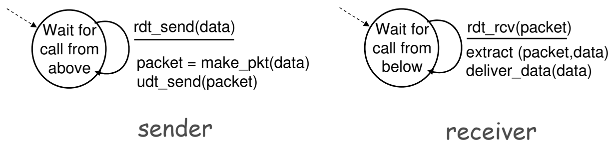
Reliable data transfer: getting started

We'll:
- incrementally develop sender, receiver sides of 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

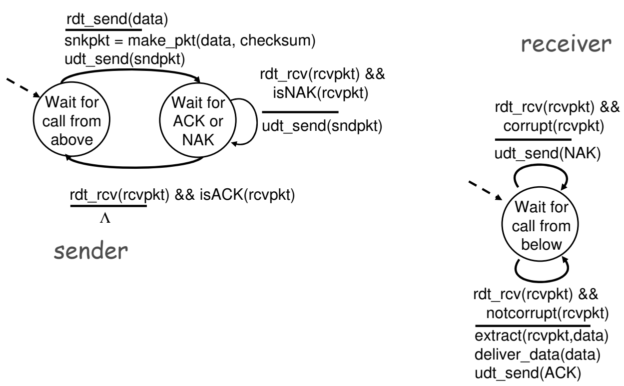
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets (→no congestion, no buffer overflows)
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



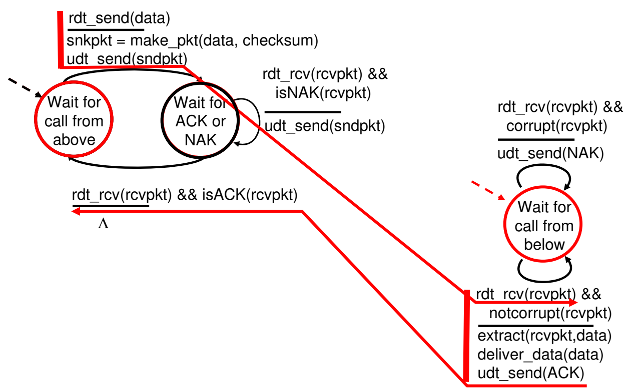
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP checksum to detect bit errors
- **Still no loss!!**
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
  - human scenarios using ACKs, NAKs?
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

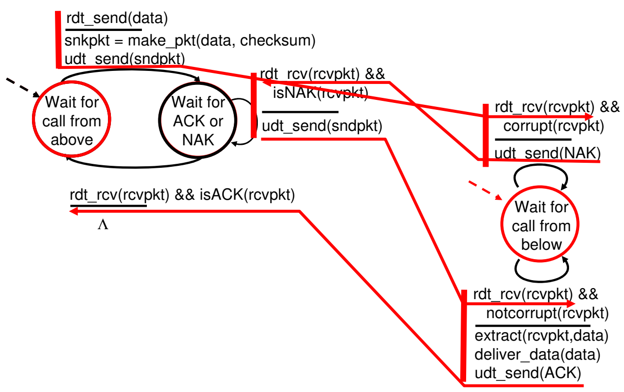
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

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

What to do?
- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

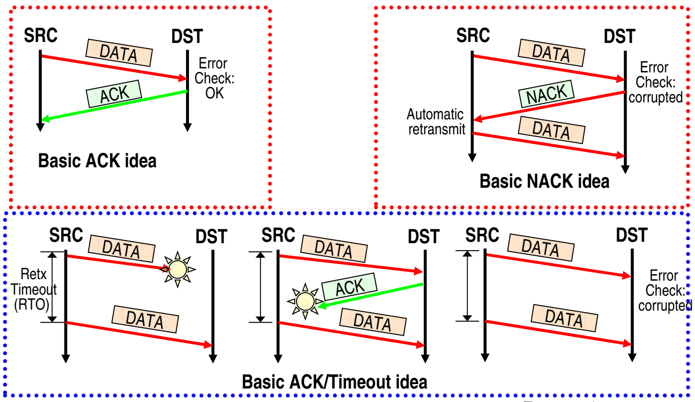
Handling duplicates:
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

> stop and wait
> Sender sends one packet, then waits for receiver response

Retransmission scenarios

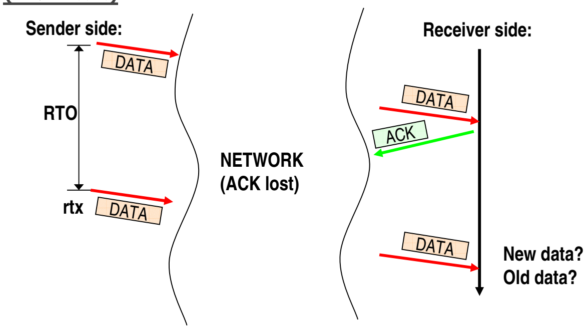
referred to as ARQ schemes (Automatic Retransmission reQuest)

COMPONENTS: a) error checking at receiver; b) feedback to sender; c) retx



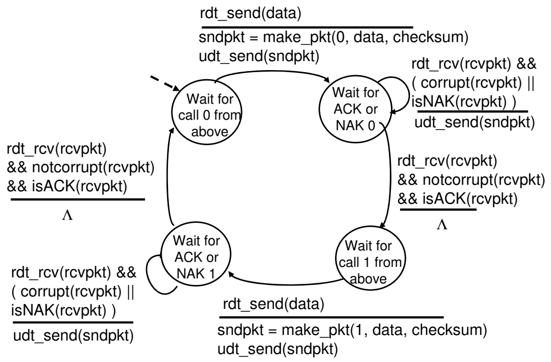
Why sequence numbers?

(on data)

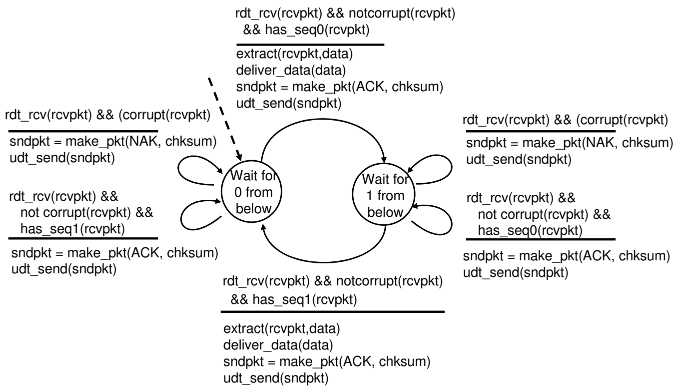


Need to univocally "label" all packets circulating in the network between two end points. 1 bit (0-1) enough for Stop-and-wait

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

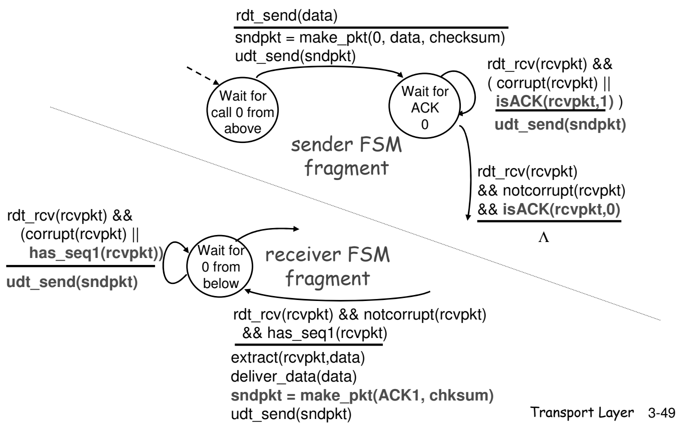
Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using NAKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender, receiver fragments



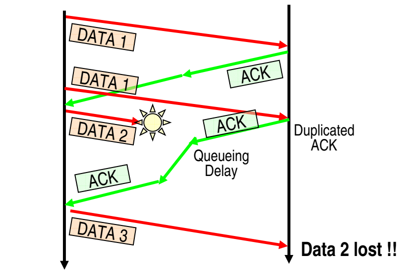
rdt3.0: channels with errors *and* loss

New assumption: underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: how to deal with loss?
- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

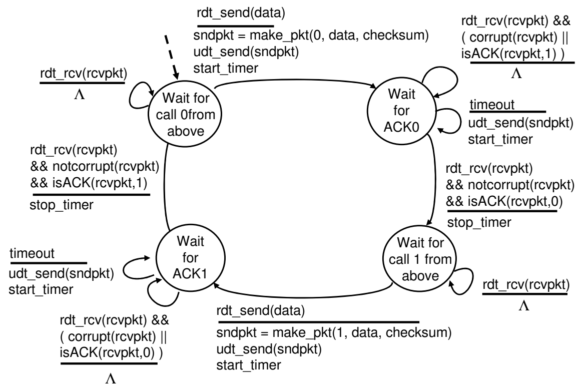
Approach: sender waits "reasonable" amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

Why sequence numbers?
(on ack)

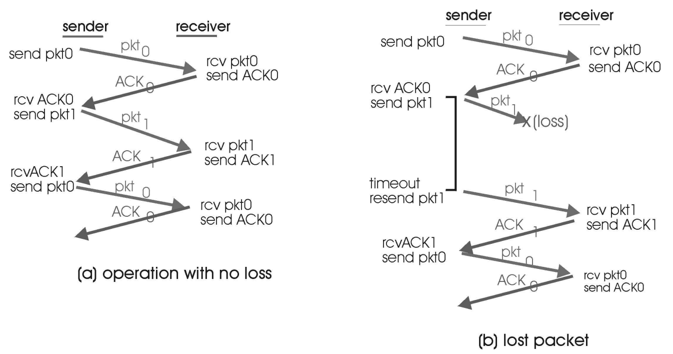


With pathologically critical network (as the Internet!) also need to univocally "label" all acks circulating in the network between two end points. 1 bit (0-1) enough for Stop-and-wait ?

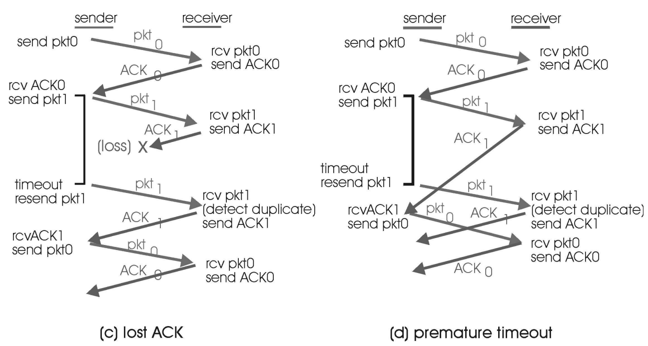
rdt3.0 sender



rdt3.0 in action



rdt3.0 in action



Performance of rdt3.0

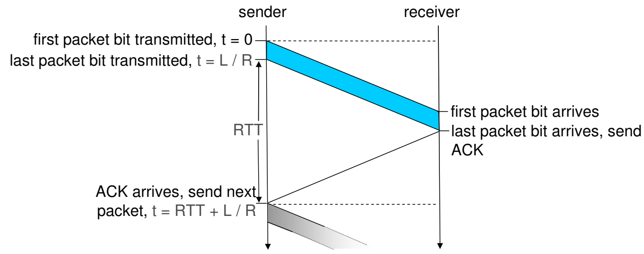
- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

  - U_{sender}: utilization – fraction of time sender busy sending
  - 1KB pkt every 30 msec -> 33kB/sec throuput over 1 Gbps link
  - network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

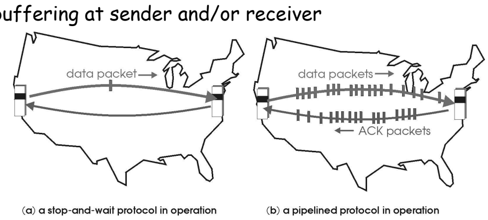


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

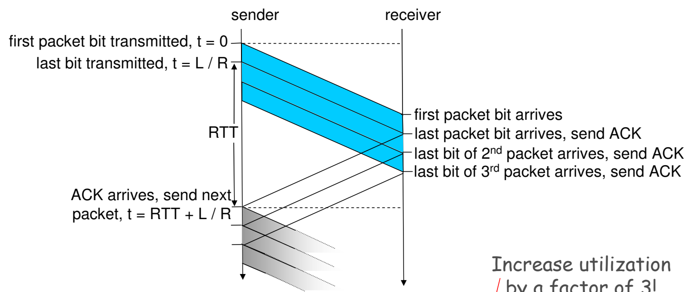
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



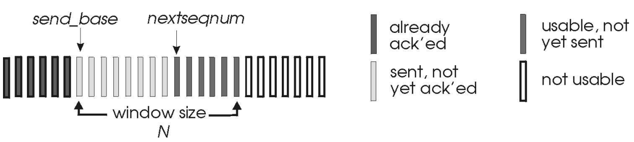
Increase utilization by a factor of 3!

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

Go-Back-N

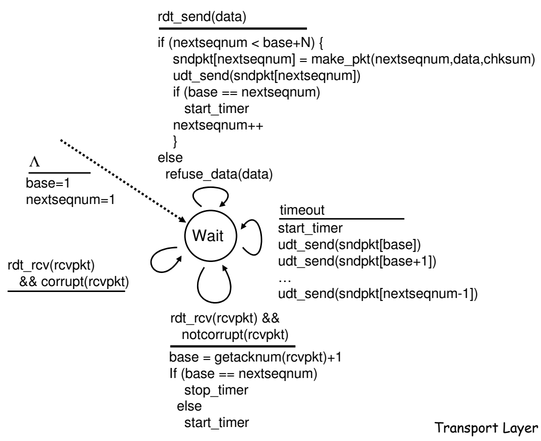
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

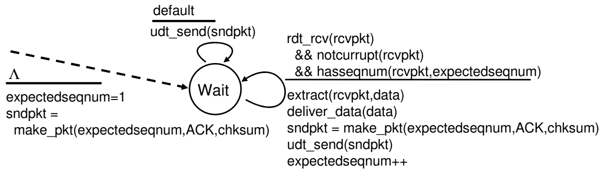


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

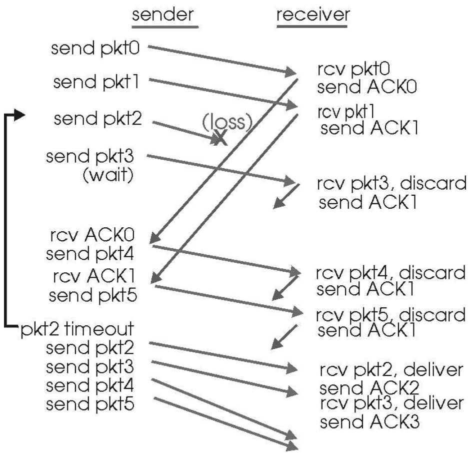


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

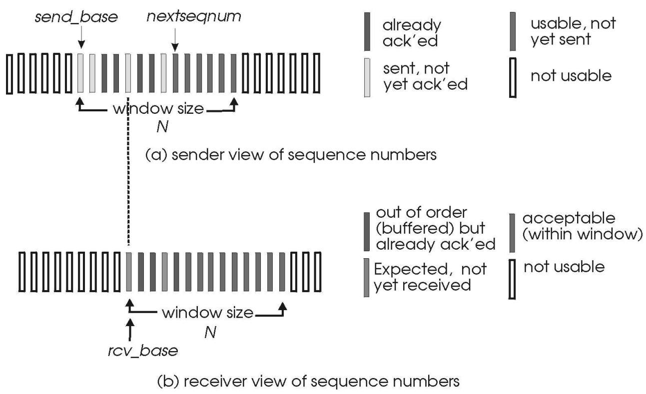
GBN in action



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 above :

- if next available seq # in window, send pkt

timeout(n): Each packet has one Logical timer

- 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]

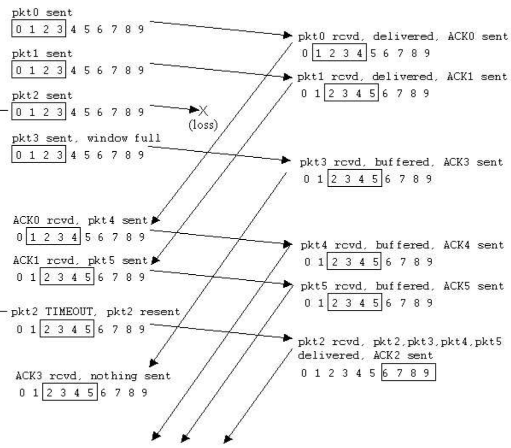
- ACK(n)

otherwise:

- ignore

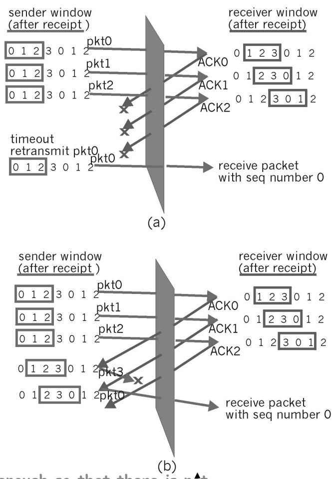
Important!! Sender and receiver may have different views!!

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 between seq # size and window size?

Clearly at least the window must be small enough so that there is not ambiguity on sequence numbers!!! Is it enough in Selective Repeat??

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

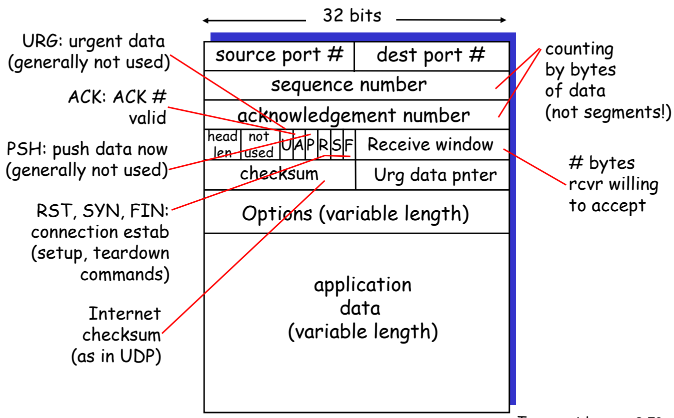
TCP: Overview

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*

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



Source port								Destination port
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	URG	ACK	PSH	RST	SYN	FIN	Window size
checksum								Urgent pointer

- Sequence number:
  - Sequence number of the ***first*** byte in the segment.
  - When reaches 2^{32}-1, next wraps back to 0
- Acknowledgement number:
  - valid only when ACK flag on
  - Contains the ***next*** byte sequence number that the host ***expects*** to receive (= last successfully received byte of data + 1)
  - grants successful reception for all bytes up to ack# - 1 (cumulative)
- When seq/ack reach 2^{32}-1, next wrap back to 0

TCP data transfer management

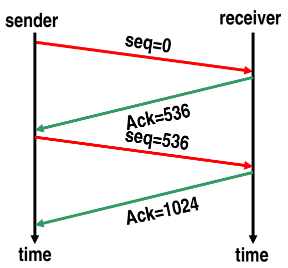
- Full duplex connection
  - data flows in both directions, independently
  - To the application program these appear as two unrelated data streams
- each end point maintains a sequence number
  - Independent sequence numbers at both ends
  - Measured in bytes
- acks often carried on top of reverse flow data segments (piggybacking)
  - But ack packets alone are possible

Byte-oriented

Example: 1 Kbyte message – 1024 bytes

0	1	...	...	100	...	...	535	...	...	...	1023

Example: segment size = 536 bytes → 2 segments: 0-535; 536-1023



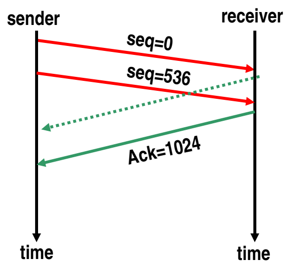
→No explicit segment size indication

- Seq = first byte number
- Returning Ack = last byte number + 1
- Segment size = Ack-seq#

Pipelining – cumulative ack

Example: 1024 bytes msg; seg_size = 536 bytes → 2 segments: 0-535; 536-1023

0	1	...	...	100	...	...	535	...	...	...	1023



→Cumulative ack

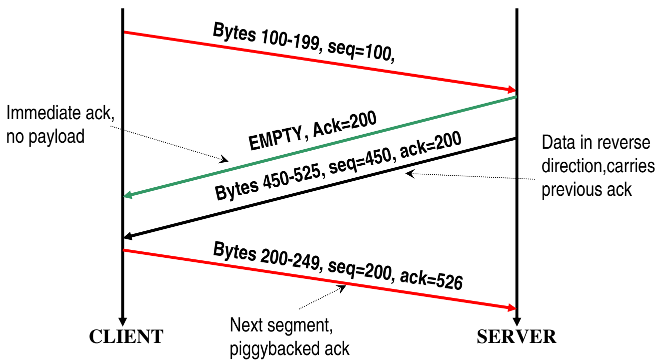
- ACK = **all** previous bytes correctly received!
- E.g. ACK=1024: all bytes 0-1023 received

Other names of pipelining:

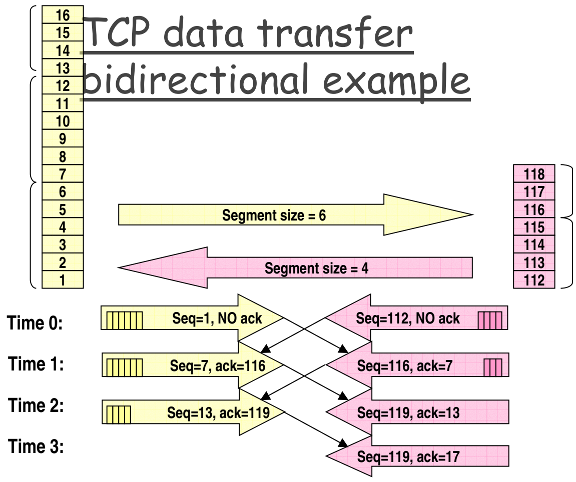
- Go-Back-N ARQ mechanisms
- Sliding window mechanisms

Why pipelining? Dramatic improvement in efficiency!

Multiple acks; Piggybacking



TCP data transfer bidirectional example



TCP seq. #'s and ACKs

Seq. #'s:

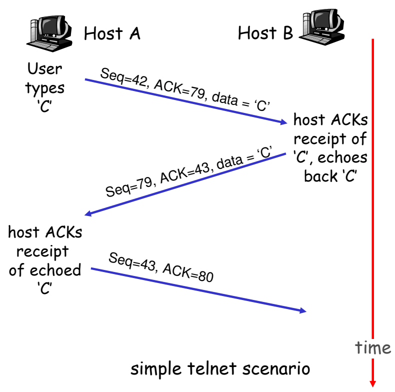
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

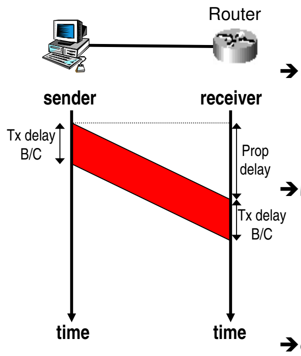
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



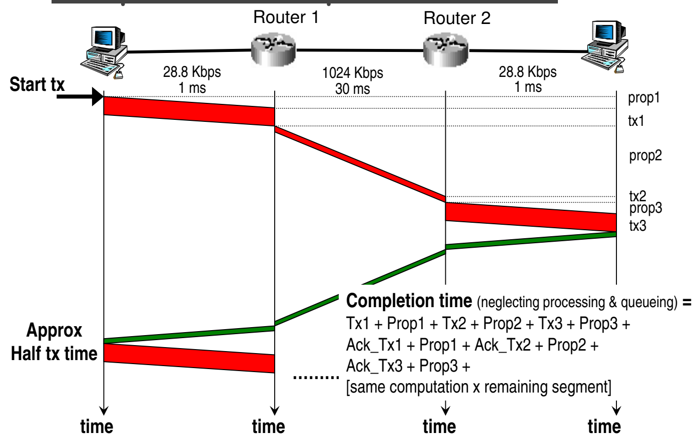
Performance issues with/without pipelining

Link delay computation

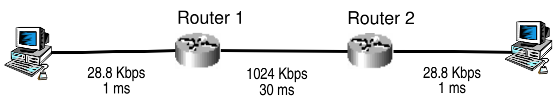


- ➔Transmission delay:
  - ➔C [bit/s] = link rate
  - ➔B [bit] = packet size
  - ➔transmission delay = B/C [sec]
- ➔Example:
  - ➔512 bytes packet
  - ➔64 kbps link
  - ➔transmission delay = 512*8/64000 = 64ms
- ➔Propagation delay – constant depending on
  - ➔Link length
  - ➔Electromagnetig waves propagation speed in considered media
    - ➔200 km/s for copper links
    - ➔300 km/s in air
- ➔other delays neglected
  - ➔Queueing
  - ➔processing

Stop-and-wait performance



Stop-and-wait performance
Numerical example



- Message:
  - 1024 bytes;
  - 2 segments: 536+488 bytes
  - Overhead: 20 bytes TCP + 20 bytes IP
  - ACK = 40 bytes (header only)

➔ Segment 1:
- ⇨ Tx1 = 576*8/28,8 = 160ms
- ⇨ Tx3 = Tx1
- ⇨ Tx2 = 576*8/1024 = 4,5 ms

➔ Segment 2:
- ⇨ Tx1 = 528*8/28,8 = 146,7ms
- ⇨ Tx3 = Tx1
- ⇨ Tx2 = 528*8/1024 = 4,1 ms

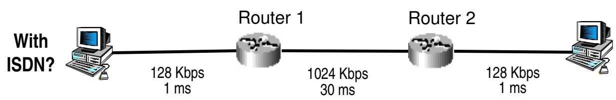
➔ Acks:
- ⇨ Tx1 = Tx3 = 40*8/28,8 = 11,1ms
- ⇨ Tx2 = 40*8/1024 = 0,3 ms

RESULT:

D = 667 (tx total) + 2*RTT = = 795 ms

THR = 1024*8/795 = = 10,3 kbps

Stop-and-wait performance
Numerical example



➔ Segment 1:
- ⇨ Tx1 = Tx3 = 576*8/128 = 36ms
- ⇨ Tx2 = 576*8/1024 = 4,5 ms

➔ Segment 2:
- ⇨ Tx1 = Tx3 = 528*8/128 = 33 ms
- ⇨ Tx2 = 528*8/1024 = 4,1 ms

➔ Acks:
- ⇨ Tx1 = Tx3 = 40*8/128 = 2,5 ms
- ⇨ Tx2 = 40*8/1024 = 0,3 ms

RESULT:

D = 151,9 (tx total) + 2*RTT = = 279,9 ms

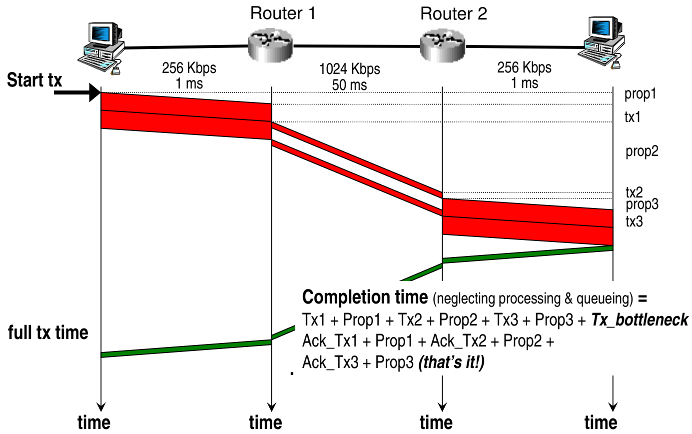
THR = 1024*8/279,9 = = 29,3 kbps

on Gbps fiber optics?

D = negligible + 2*RTT = = 128 ms

THR = 1024*8/128 = = 64 kbps

Pipelining performance



Pipelining performance
numerical example

On 28,8 kbps links

D = 347 (tx segm1+ack) + RTT + + 160 (segm2 bottleneck) = = 571 ms

THR = 1024*8/571 = = 14,3 kbps

On 128 kbps ISDN links

D = 81,8 (tx segm1+ack) + RTT + + 33 (segm2 bottleneck) = = 178,8 ms

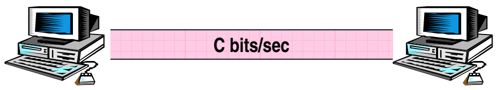
THR = 1024*8/178,8 = = 45,8 kbps

on Gbps fiber optics?

D = negligible + RTT = 64 ms

THR = 1024*8/64 = 128 kbps

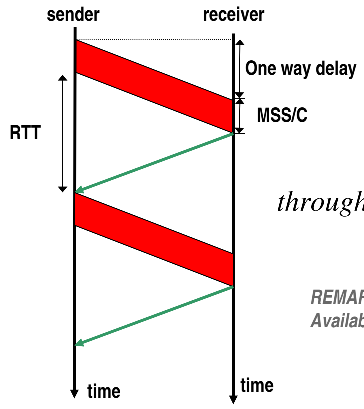
Simplified performance model



Approximate analysis, much simpler than multi-hop
Typically, C = bottleneck link rate

MSS = segment size (ev. ignore overhead)
MSIZE = message size
Ignore ACK transmission time
No loss of segments
W = number of outstanding segments
- W=1: stop-and-wait
- W>1: go-back-N (sliding window)
- *This is a highly dynamic parameter in TCP!!*
- *For now, consider W fixed*

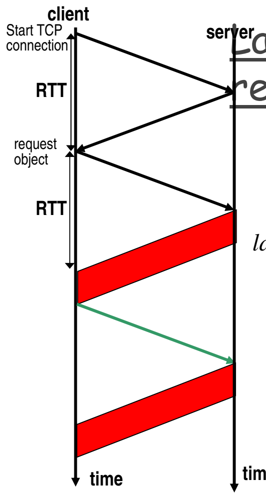
W=1 case (stop-and-wait)



$$throughput = \frac{MSS}{RTT + MSS/C}$$

REMARK: throughput always lower than Available link rate!

Latency in TCP retrieval model



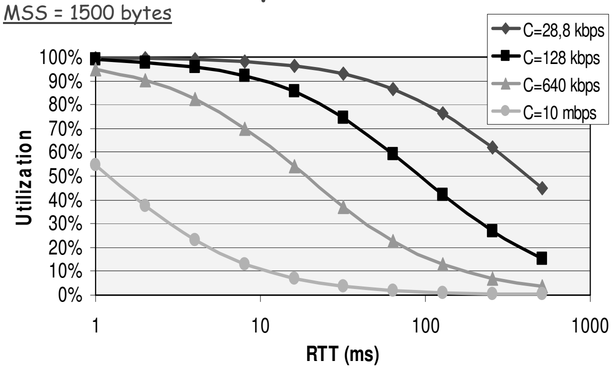
Latency: time elapsing between TCP connection Request, and last bit received at client

$$latency = 2RTT + \frac{MSIZE}{C} + \left\lceil \frac{MSIZE}{MSS} - 1 \right\rceil RTT$$

Number of segments In which message is split

W=1 case (stop-and-wait)

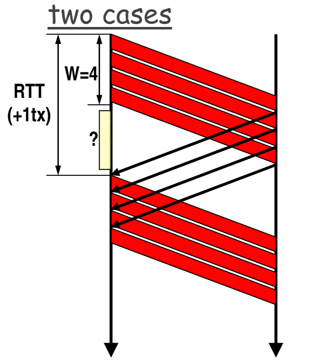
MSS = 1500 bytes



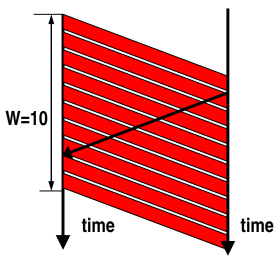
Under-utilization with: 1) high capacity links, 2) large RTT links

Pipelining (W>1) analysis

two cases



UNDER-SIZED WINDOW: THROUGHPUT INEFFICIENCY



WINDOW SIZING that allows CONTINUOUS TRANSMISSION

Continuous transmission

Condition in which link rate is fully utilized

$$\underbrace{W \cdot \frac{MSS}{C}}_{\text{Time to transmit W segments}} > \underbrace{RTT + \frac{MSS}{C}}_{\text{Time to receive Ack of first segment}}$$

We may elaborate:

$$W \cdot MSS > RTT \cdot C + MSS \approx RTT \cdot C$$

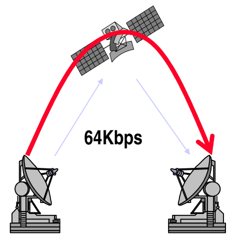
This means that full link utilization is possible when window size (in bits) is Greater than the bandwidth (C bit/s) delay (RTT s) product!

Bandwidth-delay product

➔Network: like a pipe

➔*C* [bit/s] x *D* [s]

- ⇨ number of bits "flying" in the network
- ⇨ number of bits injected in the network by the tx, before that the first bit is rxed



A 15360 (64000x0.240) bits "worm" in the air!!

bandwidth-delay product = no of bytes that saturate network pipe

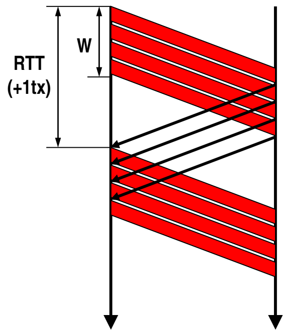
Long Fat Networks

LFNs (el-ef-an(t)s): large bandwidth-delay product

NETWORK	RTT (ms)	rate (kbps)	BxD (bytes)
Ethernet	3	10.000	3.750
T1, transUS	60	1.544	11.580
T1 satellite	480	1.544	92.640
T3 transUS	60	45.000	337.500
Gigabit transUS	60	1.000.000	7.500.000

The 65535 (16 bit field in TCP header) maximum window size W may be a limiting factor!

Pipelining (W>1) analysis



$$thr = \min\left(C, \frac{W \cdot MSS}{RTT + MSS / C} \right)$$

Delay analysis (for TCP object retrieval) – Non continuous transmission

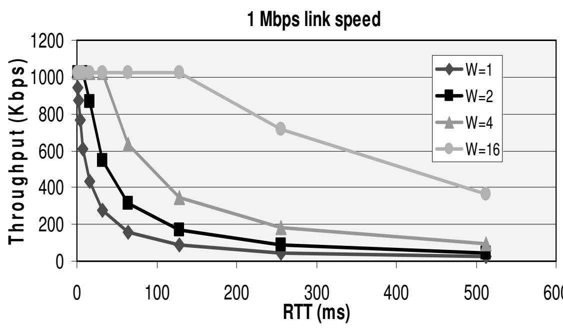
$$latency = 2RTT + \frac{MSIZE}{C} + \left\lceil \frac{MSIZE}{W \cdot MSS} - 1 \right\rceil \left(RTT - \frac{(W-1)MSS}{C} \right)$$

Continuous transmission case:

$$latency = 2RTT + \frac{MSIZE}{C}$$

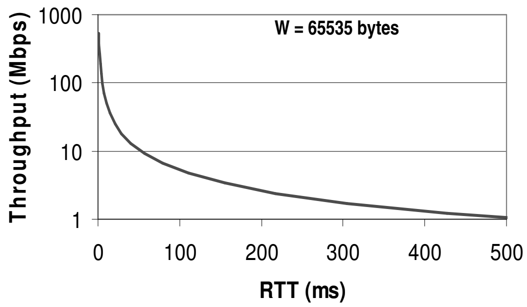
Throughput for pipelining

MSS = 1500 bytes



Maximum achievable throughput

(assuming infinite speed line…)

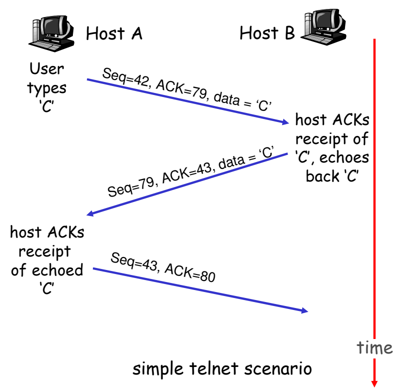


TCP seq. #'s and ACKs

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 implementor



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?

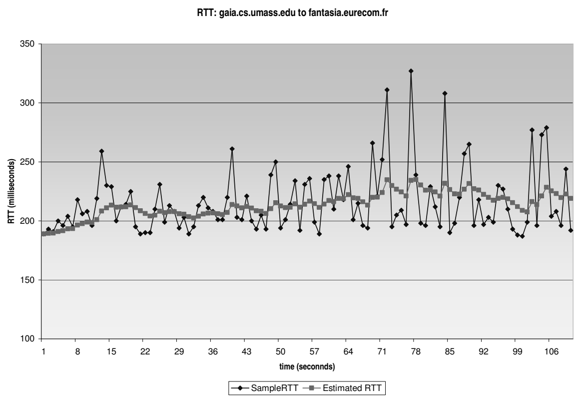
- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current `SampleRTT`

TCP Round Trip Time and Timeout

`EstimatedRTT = (1- α)*EstimatedRTT + α*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

- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> 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:

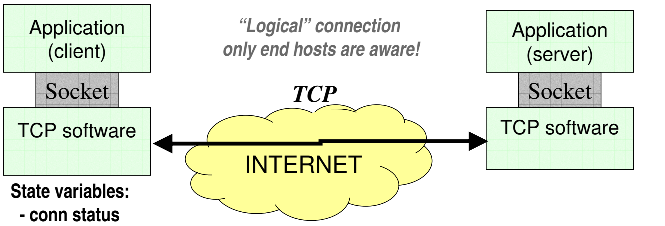
`TimeoutInterval = EstimatedRTT + 4*DevRTT`

Understanding TCP connection management

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 connection



State variables:
- **conn status**
- **MSS**
- **windows**
- **...**

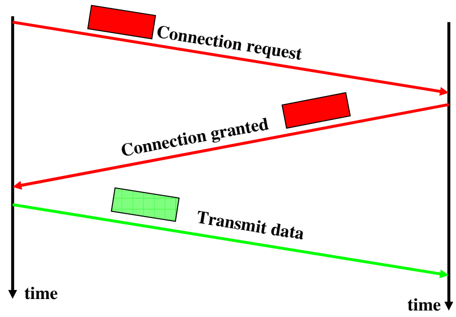
buffer space
normally 4 to 16 Kbytes
64+ Kbytes possible

Connection described by client&server status
Connection SET-UP duty:
1) initializes state variables
2) reserves buffer space

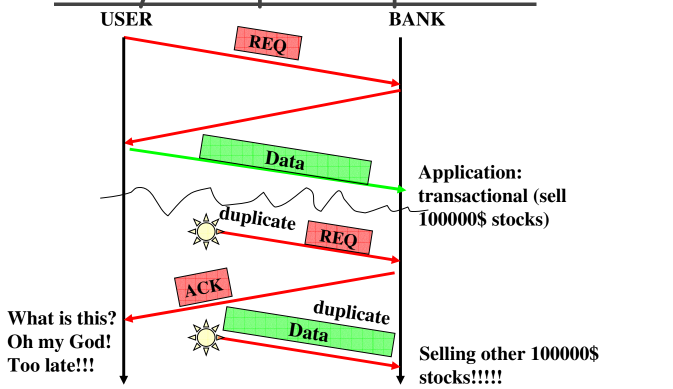
Transmission control block

Contains also info on: sockets, pointers to the users' send and receive buffers, to the retransmit queue and to the current segment

Connection establishment: simplest approach (non TCP)

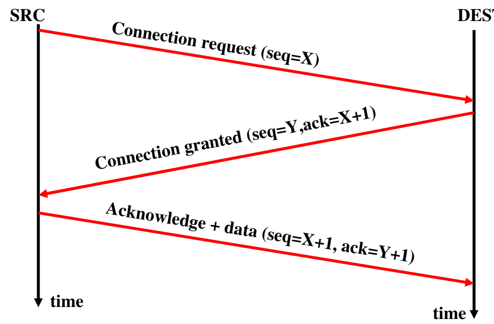


Delayed duplicate problem

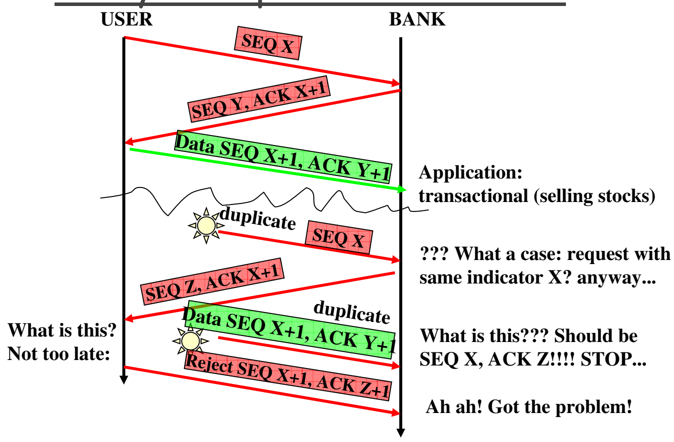


Solution: three way handshake

Tomlinson 1975



Delayed duplicate detection

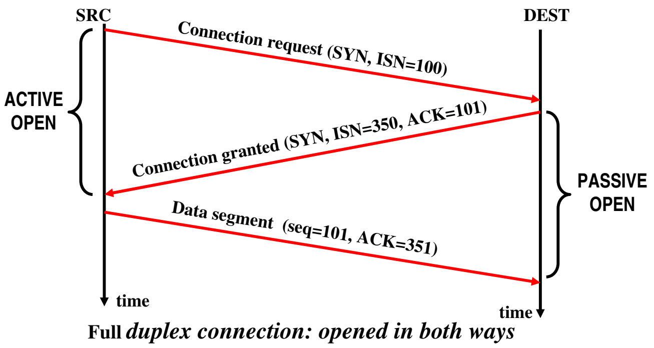


Source port								Destination port
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	URG	ACK	PSH	RST	SYN	FIN	Window size
checksum								Urgent pointer

- SYN (synchronize sequence numbers): used to open connection
  - SYN present: this host is setting up a connection
  - SEQ with SYN: means initial sequence number (ISN)
  - data bytes numbered from ISN+1.
- FIN: no more data to send
  - used to close connection

...more later about connection closing...

Three way handshake in TCP



Full *duplex connection: opened in both ways*

SRC: performs ACTIVE OPEN

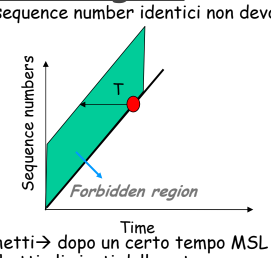
DEST: Performs PASSIVE OPEN

Initial Sequence Number

- Should change in time
  - RFC 793 (but not all implementations are conforming) suggests to generate ISN as a sample of a 32 bit counter incrementing at 4us rate
- transmitted whenever SYN (Synchronize sequence numbers) flag active
  - note that both src and dest transmit THEIR initial sequence number (remember: full duplex)
- Data Bytes numbered from ISN+1
  - necessary to allow SYN segment ack

Forbidden Region

- Obiettivo: due sequence number identici non devono trovarsi in rete allo stesso tempo

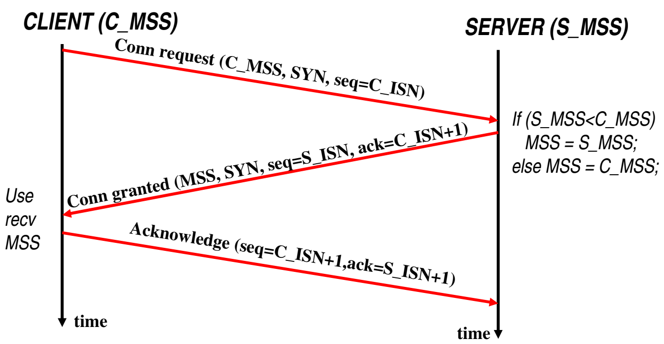


- Aging dei pacchetti→ dopo un certo tempo MSL (Maximum Segment Lifetime) i pacchetti eliminati dalla rete
- Sequence numbers basati sul clock
- Un ciclo del clock circa 4 ore; MSL circa 2 minuti.
- → Se non ci sono crash che fanno perdere il valore dell'ultimo sequence number usato NON ci sono problemi (si riusa lo stesso sequence number ogni 4 ore circa, quando il segmento precedentemente trasmesso con quel sequence number non è più in rete)
- → Cosa succede nel caso di crash? RFC suggerisce l'uso di un 'periodo di silenzio' in cui non vengono inviati segmenti dopo il riavvio pari all'MSL

Maximum Segment Size - MSS

- Announced at setup by both ends.
- Lower value selected.
- MSS sent in the Options header of the SYN segment
  - clearly cannot (=ignored if happens) send MSS in a non SYN segment, as connection has been already setup
  - when SYN has no MSS, default value 536 used
- goal: the larger the MSS, the better...
  - until fragmentation occurs
  - e.g. if host is on ethernet, sets MSS=1460
    - 1500 max ethernet size - 20 IP header - 20 TCP header

MSS advertise



Does not avoid fragmentation to occur WITHIN the network!!

TCP Connection Management

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

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- *client:* connection initiator
  ```
  Socket clientSocket = new Socket("hostname","port number");
  ```
- *server:* contacted by client
  ```
  Socket connectionSocket = welcomeSocket.accept();
  ```

Three way handshake:

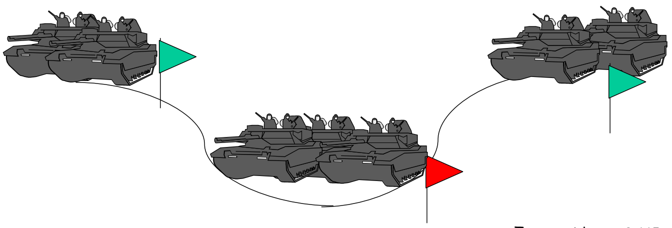
Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

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

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

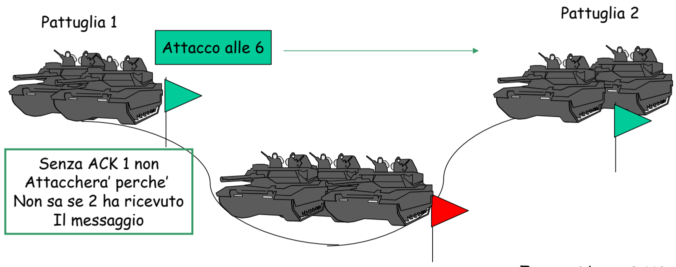
Problema dei due eserciti

- L'esercito rosso e' globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all'orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero' essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



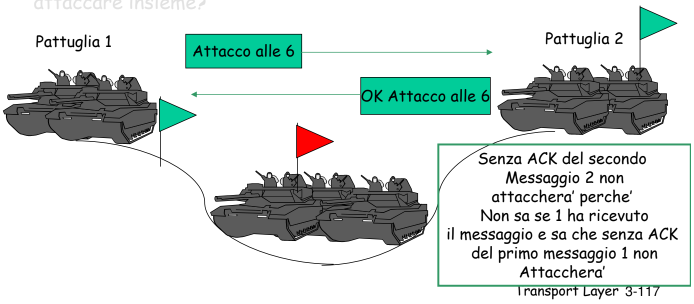
Problema dei due eserciti

- L'esercito rosso e' globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all'orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero' essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



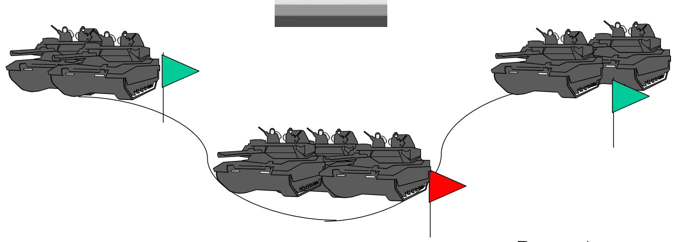
Problema dei due eserciti

- L'esercito rosso e' globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all'orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero' essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



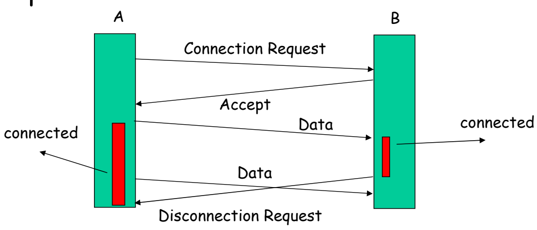
Problema dei due eserciti

- In generale: se N scambi di messaggi /Ack etc. necessari a raggiungere la certezza dell'accordo per attaccare allora cosa succede se l'ultimo mesaggio 'necessario' va perso?
- →E' impossibile raggiungere questa certezza. Le due pattuglie non attaccheranno mai!!



Problema dei due eserciti: cosa ha a che fare con le reti e TCP??

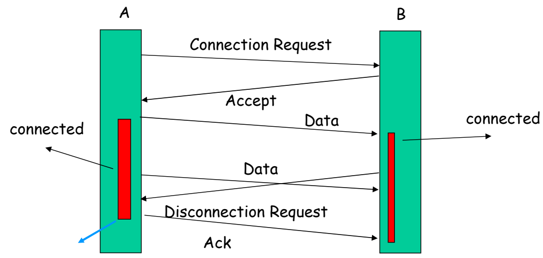
- Chiusura di una connessione. Vorremmo un accordo tra le due peer entity o rischiamo di perdere dati.



A pensa che il secondo pacchetto sia stato ricevuto. La connessione e' Stata chiusa a B prima che ciò avvenisse→ secondo pacchetto perso!!!

Quando si può dire che le due peer entity abbiano raggiunto un accordo???

- Problema dei due eserciti!!!



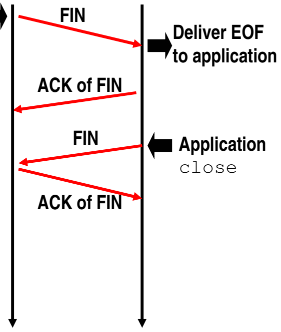
Ma se l'ACK va perso????

Soluzione: si e' disposti a correre piu' rischi quando si butta giu' una connessione di quando si attacca un esercito nemico. Possibili malfunzionamenti. Soluzioni per la recovery in questi casi

Connection closing in TCP

since it is impossible problem, use simples solution (two way handshake)

- Since connection full duplex, necessary two half-closes (each a two-way handshake) originating by both sides
- close notified with FIN flag on
- FIN segment ACK-ed as usual



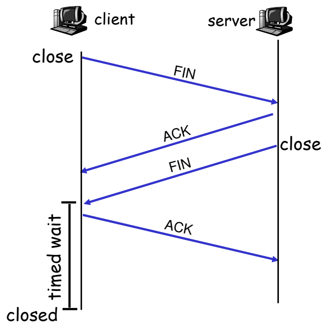
TCP Connection Management (cont.)

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.

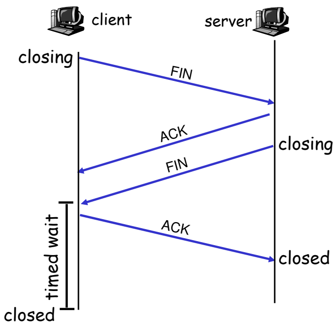


TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

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

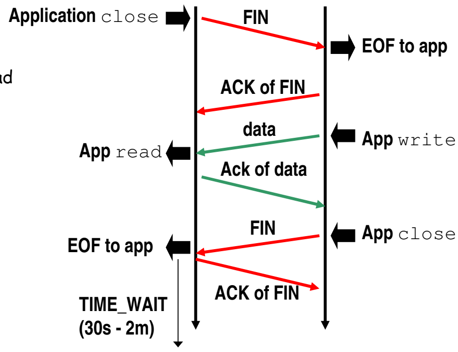
Step 4: server, receives ACK. Connection closed.



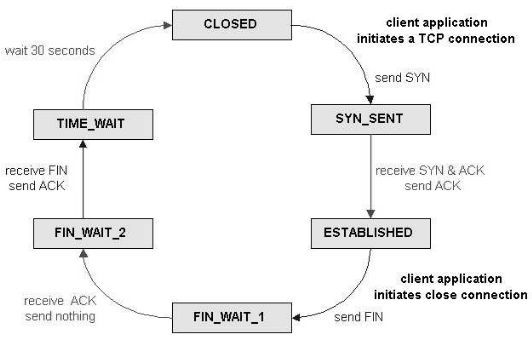
Half close

may close one direction only - seldomly used

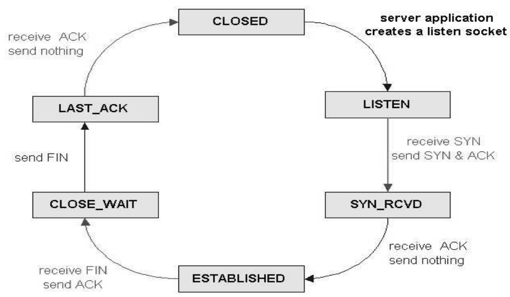
- Supported by system call **shutdown** instead of **close**



Connection states - Client



Connection States - Server



Why TIME_WAIT?

- MSL (Maximum Segment Lifetime): **maximum time a segment can live in the Internet**
  - no timers on IP packets! Only hop counter
  - RFC 793 specifies MSL=2min, but each implementation has its own value (from 30s to 2min)
- TIME_WAIT state: **2 x MSL**
  - **allows to "clean" the network of delayed packets belonging to the connection**
  - **2xMSL because a lost FIN_ACK implies a new FIN from server**
- during TIME_WAIT conn sock pair reserved
  - **many implementations even more restrictive (local port non reusable)**
  - **clearly this may be a serious problem when restarting server daemon (must pause from 1 to 4 minutes…)**

Source port			Destination port
32 bit Sequence number			
32 bit acknowledgement number			
Header length	6 bit Reserved	URG ACK PSH RST SYN FIN	Window size
checksum			Urgent pointer

- RST (Reset)
  - sent whenever a segment arrives and does not apparently belong to the connection
  - typical RST case: connection request arriving to port not in use
- Sending RST within an active connection:
  - allows ***aborting release*** of connection (versus ***orderly release***)
    - any queued data thrown away
    - receiver of RST can notify app that abort was performed at other end

Chapter 3 outline

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- 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 rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

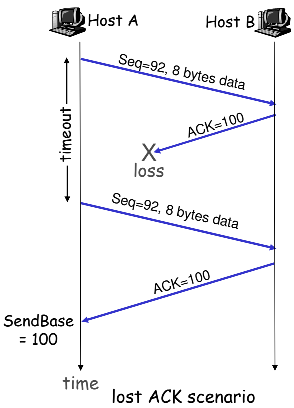
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

   event: data received from application above
       create TCP segment with sequence number NextSeqNum
       if (timer currently not running)
           start timer
       pass segment to IP
       NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
       retransmit not-yet-acknowledged segment with
            smallest sequence number
       start timer

    event: ACK received, with ACK field value of y
       if (y > SendBase) {
           SendBase = y
           if (there are currently not-yet-acknowledged segments)
               start timer
           }

 }  /* end of loop forever */
```

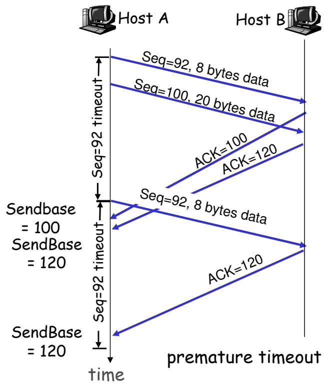
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

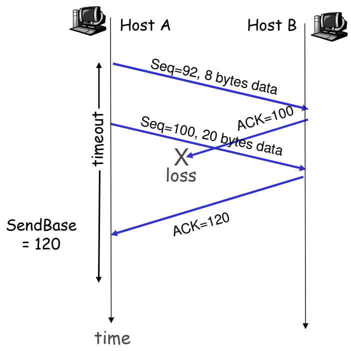
TCP: retransmission scenarios



lost ACK scenario



TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment startsat lower end of gap

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.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
          }
          else {
              increment count of dup ACKs received for y
              if (count of dup ACKs received for y = 3) {
                  resend segment with sequence number y
              }
```

a duplicate ACK for already ACKed segment

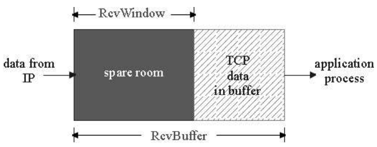
fast retransmit

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

- receive side of TCP connection has a receive buffer:

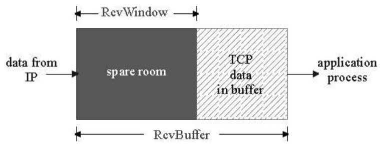


- app process may be slow at reading from buffer

flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow