Networks: TCP and UDP

Transport Layer

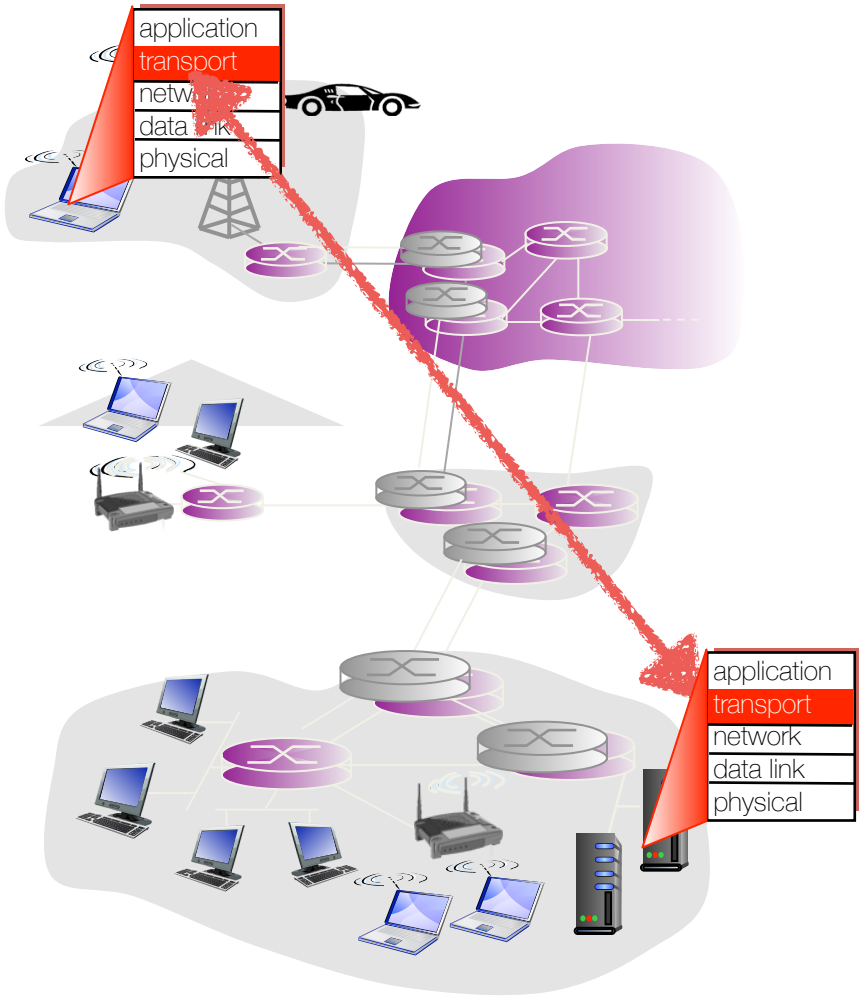
Dr Mirco Musolesi
Dr Ian Batten
Dr Eike Ritter

Transport vs. network layer

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

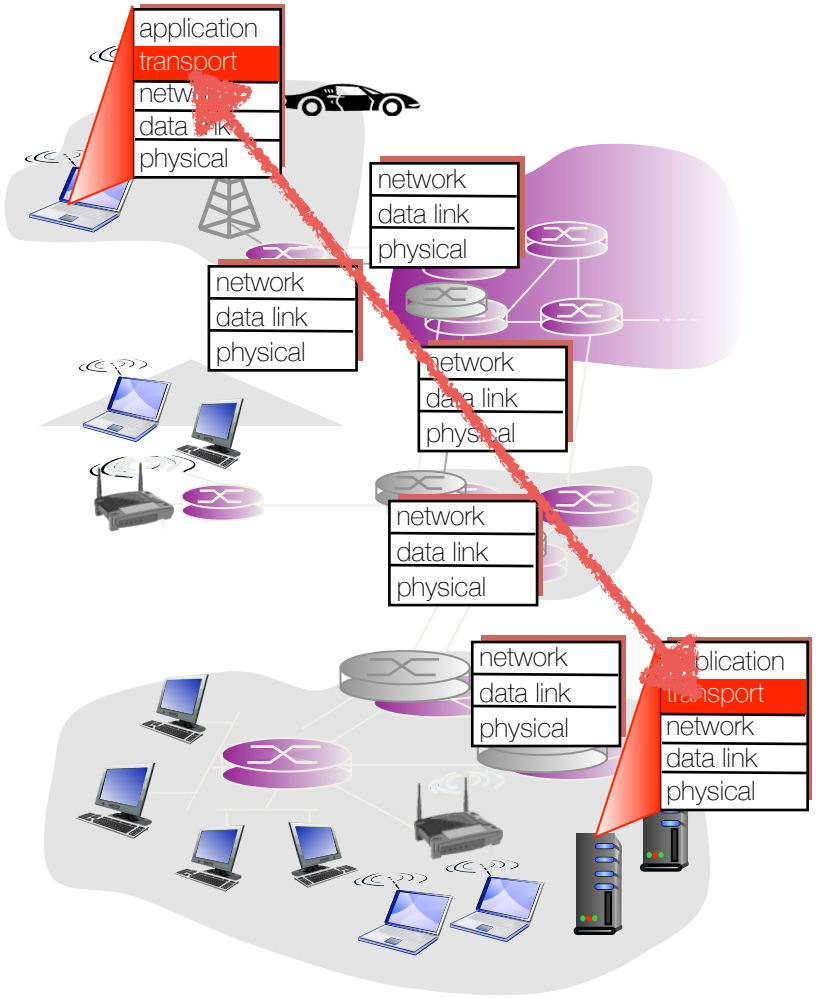
Transport services and protocols

- 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
- receive side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
- Internet: **TCP and UDP**



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



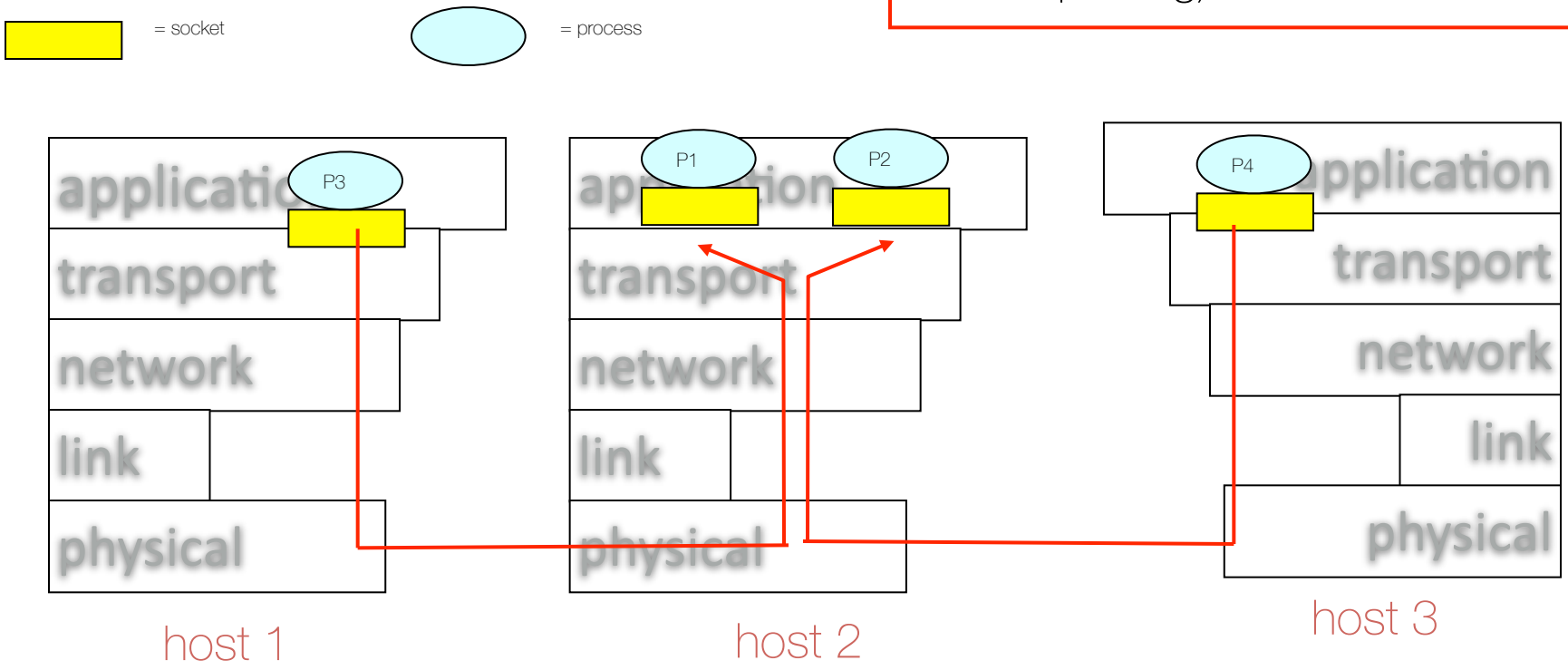
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

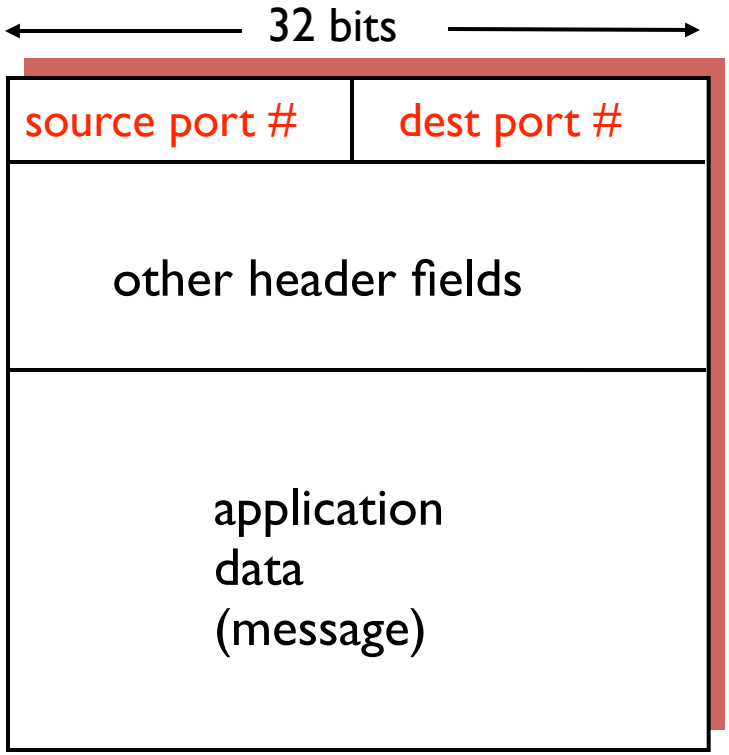
Multiplexing at send host:

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
- host uses IP addresses & port numbers to direct segment to appropriate socket

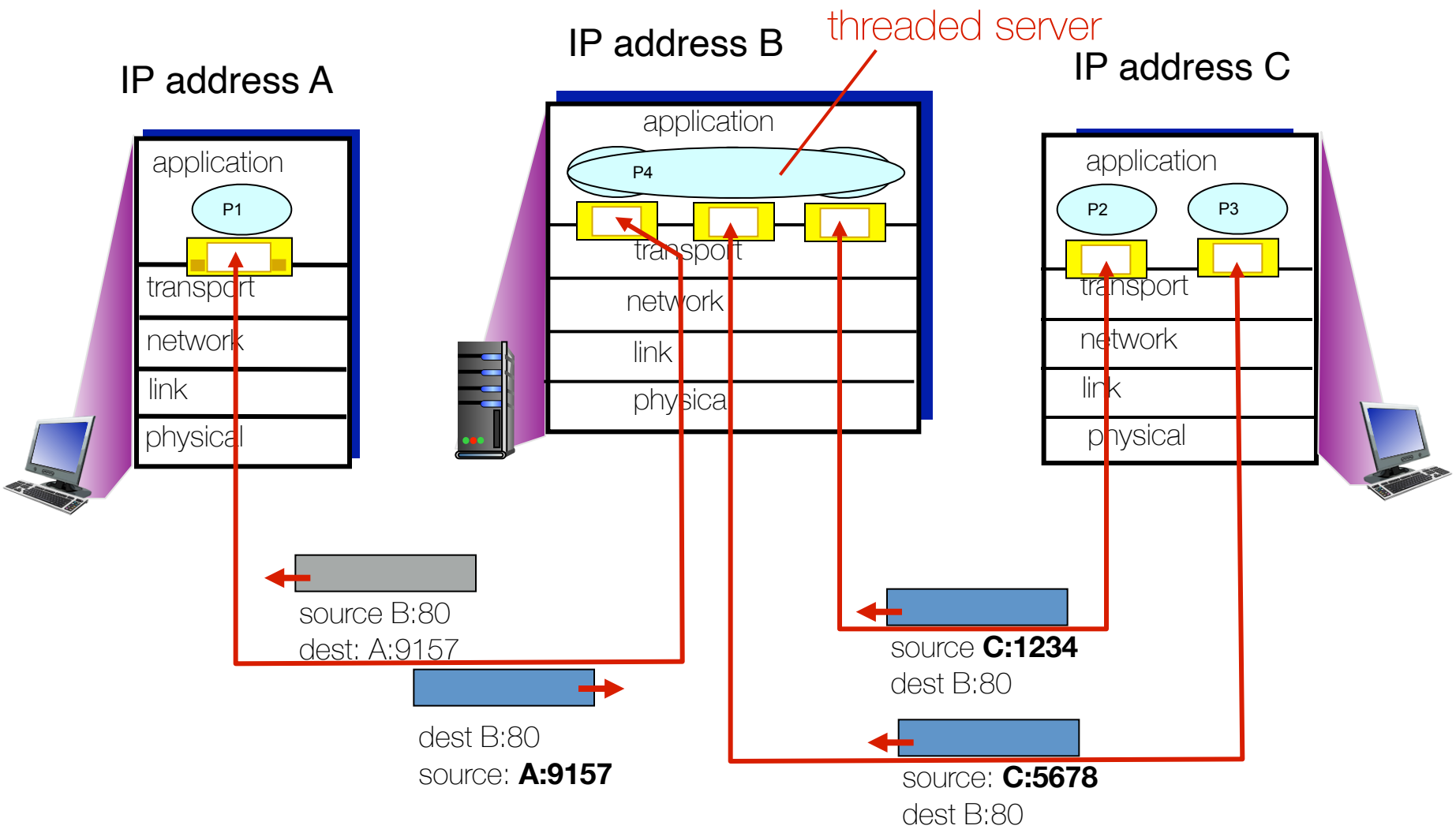


TCP/UDP segment format

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver 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: example



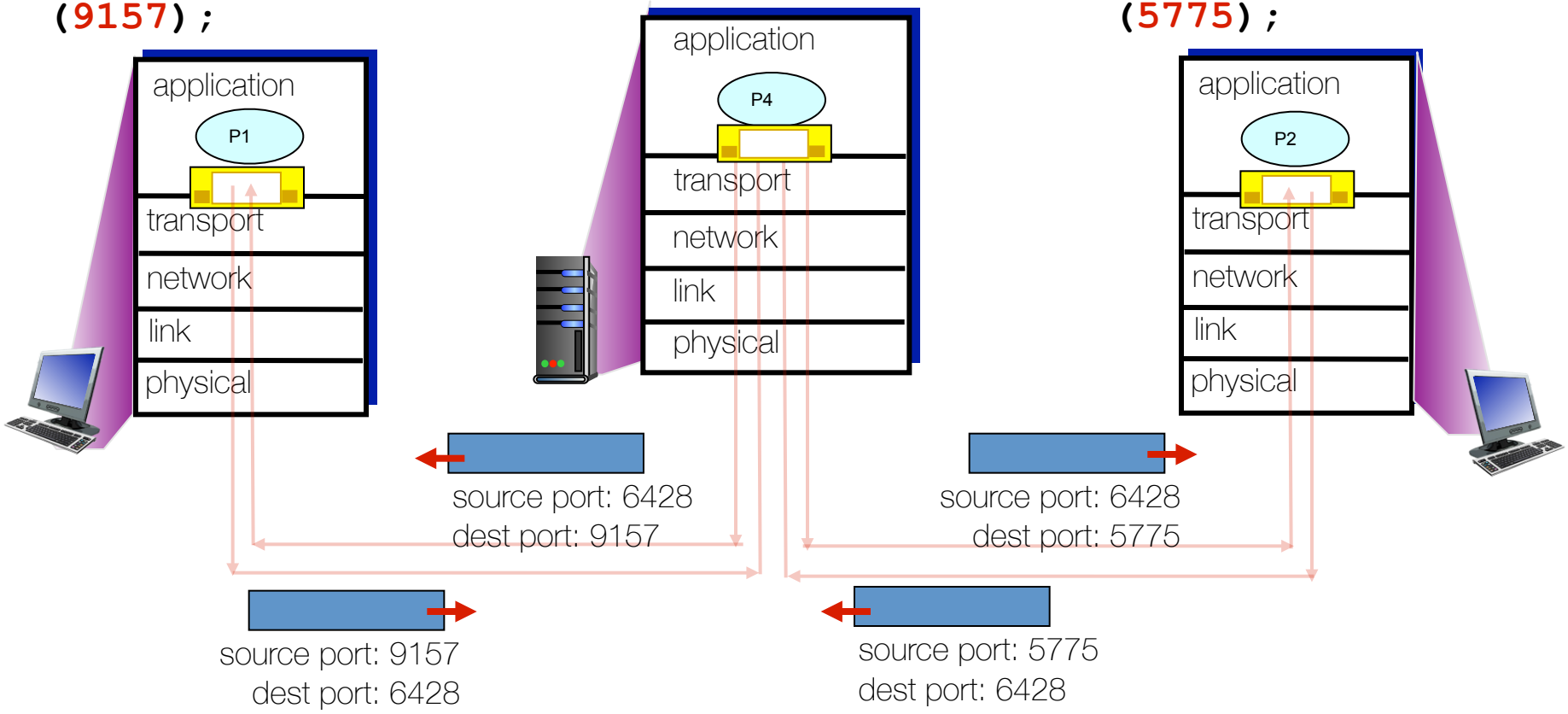
three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Connectionless demultiplexing

- Create sockets with port numbers:

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

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

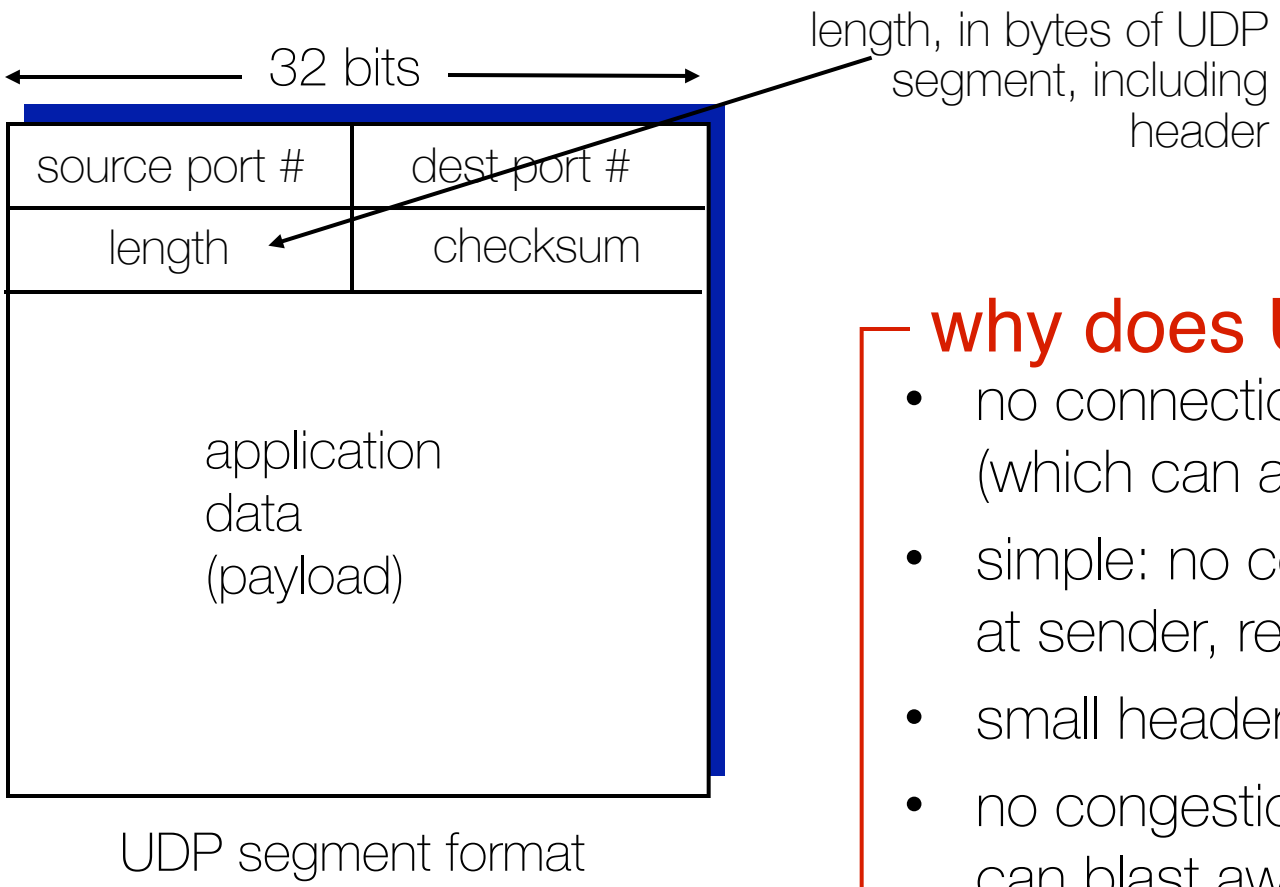
- When host receives UDP segment:
  - checks destination port number in segment
  - Directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers will be directed to same socket
- Delivery to sockets with duplicate port numbers not well defined

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

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

why does UDP exist?

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

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

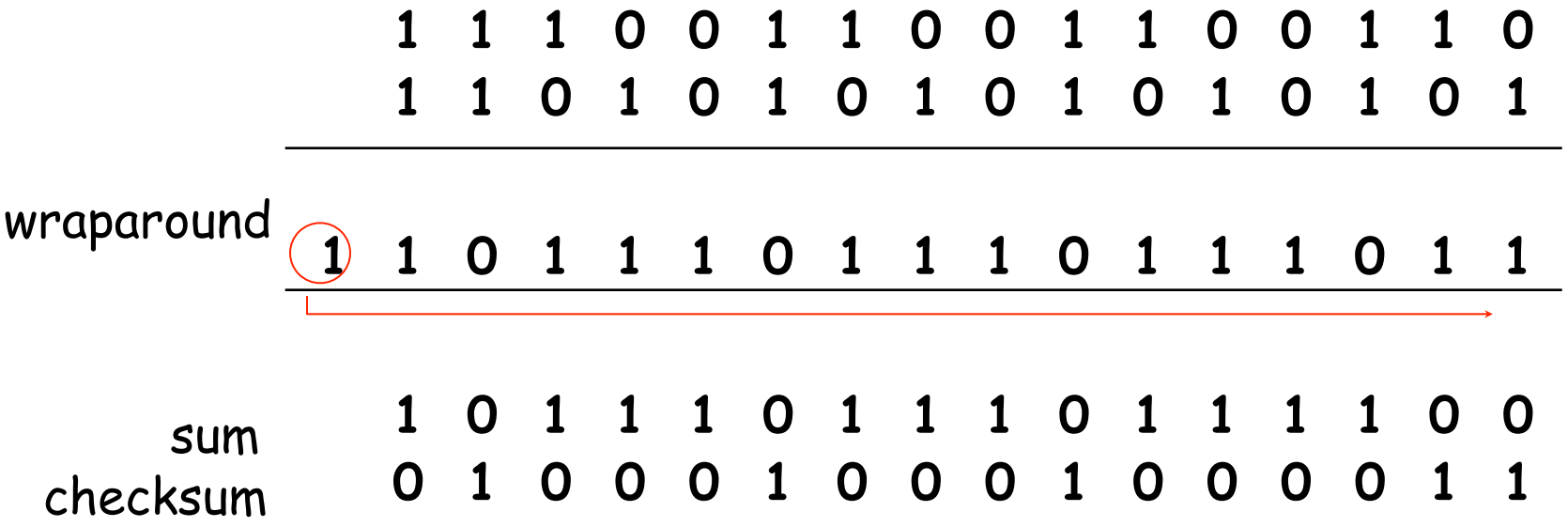
- treat segment contents, including header fields, as sequence of 16-bit integers, with checksum field set to zero
- "**Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets**." (RFC 768 – that's all it says!)
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:

Internet checksum: example

example: add two 16-bit integers



Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Implementation Issues

- The design is very 16-bit: IEN 45 in 1978 is all about pdp11s, and the non-16-bit machines are actually 36 bit (pdp10s).
  - 32 bit add (or 64 bit add) is expensive on such boxes
- You can get the effect of the end wraparound by summing 16 bit quantities in a 32 bit word and then adding the top and bottom halves together repeatedly until the top half is zero.
- RFC1071 has the gory details: it took until 1988 to publish
  - And is itself full of ancient history, like Cray assember
- Allegedly, at the first interworking "bake off" **no** implementations actually interworked…

Quick Quiz #1

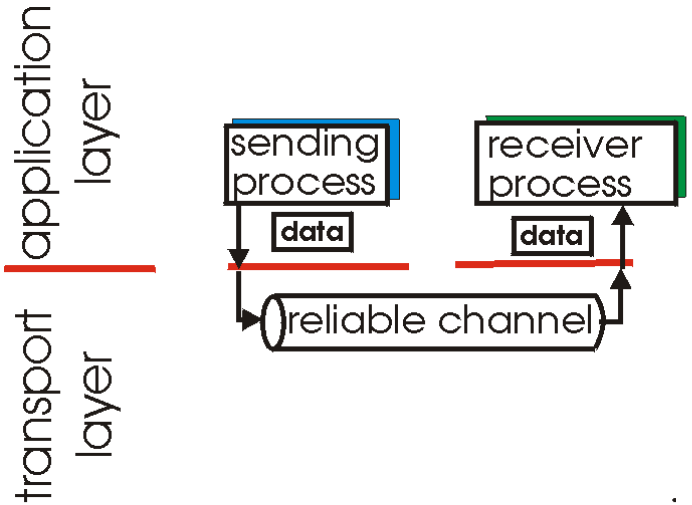
- UDP packets are retransmitted if they are lost: true or false?
- TCP offers reliable, sequenced delivery: true or false?
- UDP packets addressed to the same address/port number pair, but coming from two different computers, will be delivered to separate sockets: true or false?
- TCP packets addressed to the same address/port number pair, but coming from two different computers, will be delivered to separate sockets: true or false?

Answers to Quick Quiz #1

- False: lost UDP packets are just lost. It is up to the application layer to retransmit them (or not)
- True: TCP offers reliable, sequenced delivery
- False: all packets addressed to 1.2.3.4:53 will be delivered to the same socket, no matter where they came from
- True: TCP streams are distinguished by both the source and destination addresses and port numbers.

Principles of reliable data transfer

- important in application, transport, link layers



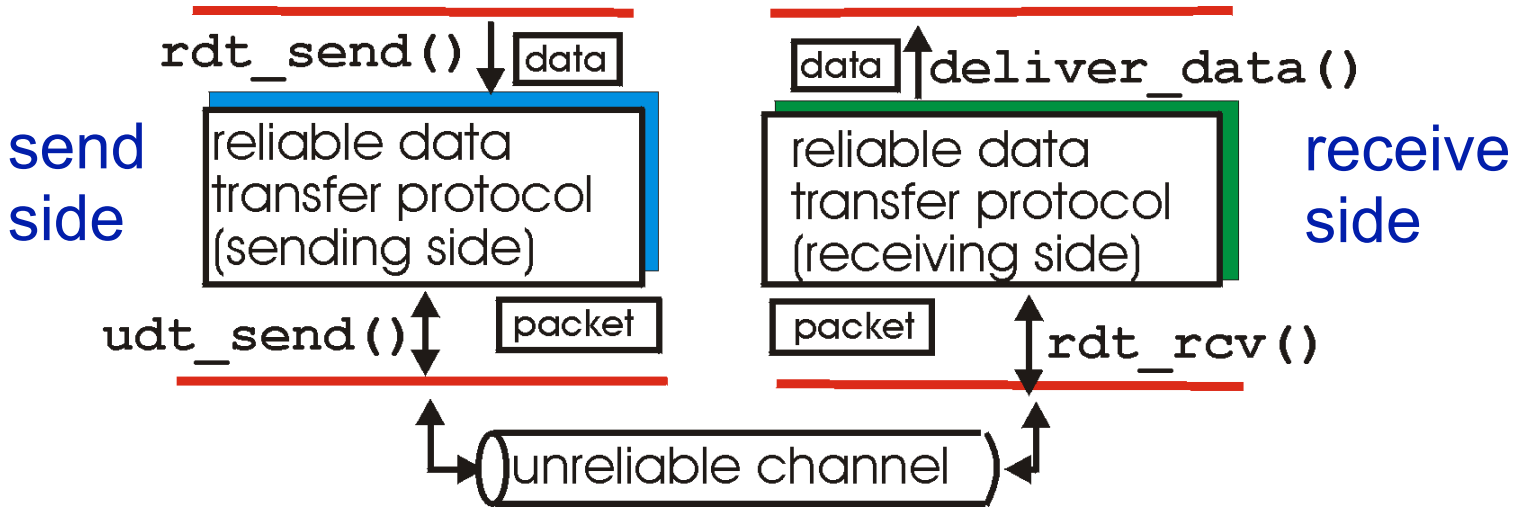
(a) provided service

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

Reliable data transfer: getting started

`rdt_send()`: called from above, (e.g., by application). Passed data to deliver to receiver upper layer

`deliver_data()`: called by **rdt** to deliver data to upper



`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

`rdt_rcv()`: called when packet arrives on rcv-side of channel

In a perfect world…

- Channels are perfectly reliable:
  - No errors (i.e., flipped bits)
  - No packet loss
  - Packets received by the receiver in order
- If a channel is perfectly reliable it is sufficient to send and receive data: you don't need any additional mechanisms such as acknowledgements
- But the world is not perfect and errors happen…
- …and you need to recover from errors!

Acknowledgement mechanisms

- Let us assume first that the channel might introduce errors (i.e., corrupted packets) but packets are not lost
- Underlying channel may flip bits in packet
  - Checksum is used to detect bit in errors
- The key problem is how to recover from errors
- Two mechanisms
  - acknowledgements (ACKs): receiver explicitly tells sender that the packet has received correctly
    - Sender retransmits on timeout

    OR
  - negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
    - Sender retransmits packet on receipt of NAK

What happens if ACKs/NACKs get corrupted?

- You can have errors in packet (data) transmission but also in ACK/NACK transmission
- And so what happens if ACKs/NACKs get corrupted?
  - Essentially, the sender does not know what happens at the receiver
- One solution is to just retransmit, **but** you can't simply do that, since you might have **duplicates** at the receiver
- Problem: how to handle duplicates

Note the checksums are weak

- The IP checksum is weak, and won't reliably detect multiple bit flips, depending on where they happen.
- Usually a channel that unreliable won't pass traffic successfully, and the TCP protocol itself will collapse
- But If you are moving data whose contents matter bit-for-bit, you need to use a stronger checksum over the whole thing (SHA384 is good, as is SHA512/256 and SHA3 in its various forms)

Handling Duplicates

- In order to handle duplicates, every packet should have a **sequence number** that identifies that packet uniquely
- The receiver discards (i.e., does not deliver the packet to the layer above) the duplicate packets

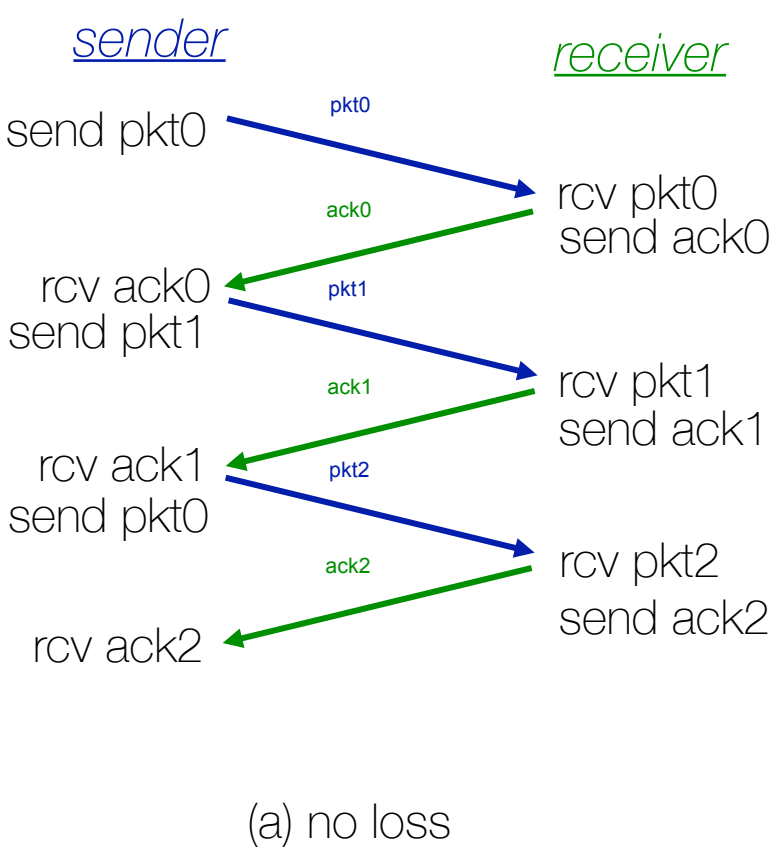
And now let's consider the general case:

channels **do** lose packets…(where "lose" includes "mangles to the point you can't tell what is was meant to be")

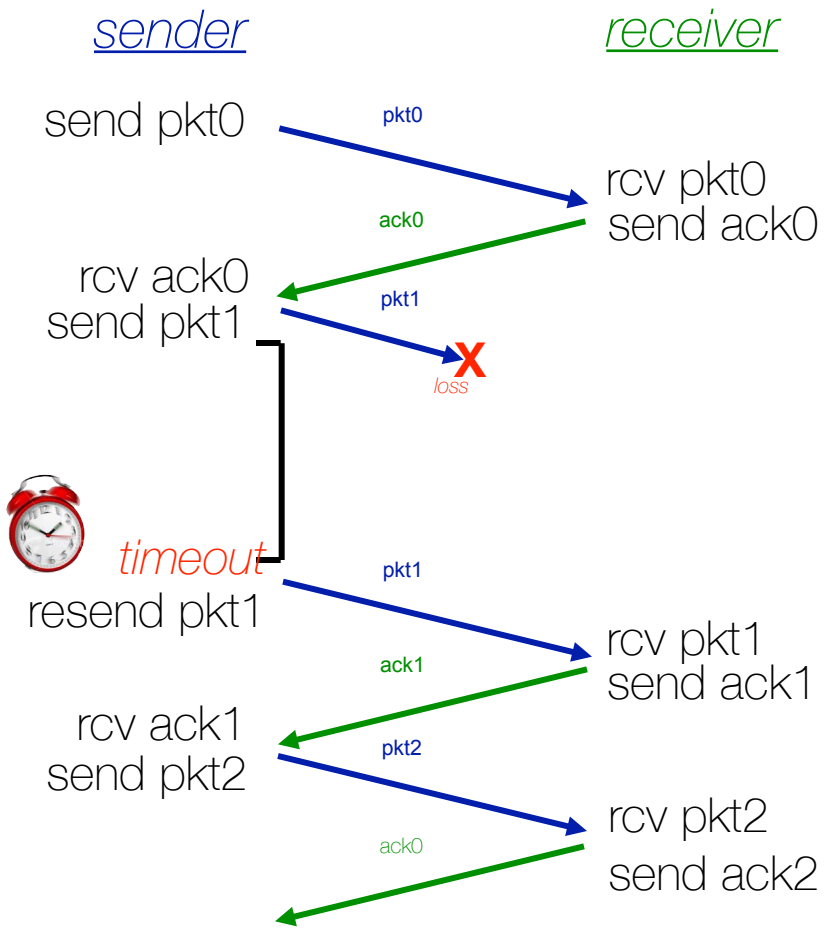
And a channel can also lose packets…

- Underlying channel can also lose packets (data, ACKs)
  - checksum, sequence numbers, ACKs, retransmissions need to work together.
- 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 detected by sequence number
    - receiver must specify sequence number of packets being ACKed
- This mechanism requires countdown timer

ACKs and time-outs in action

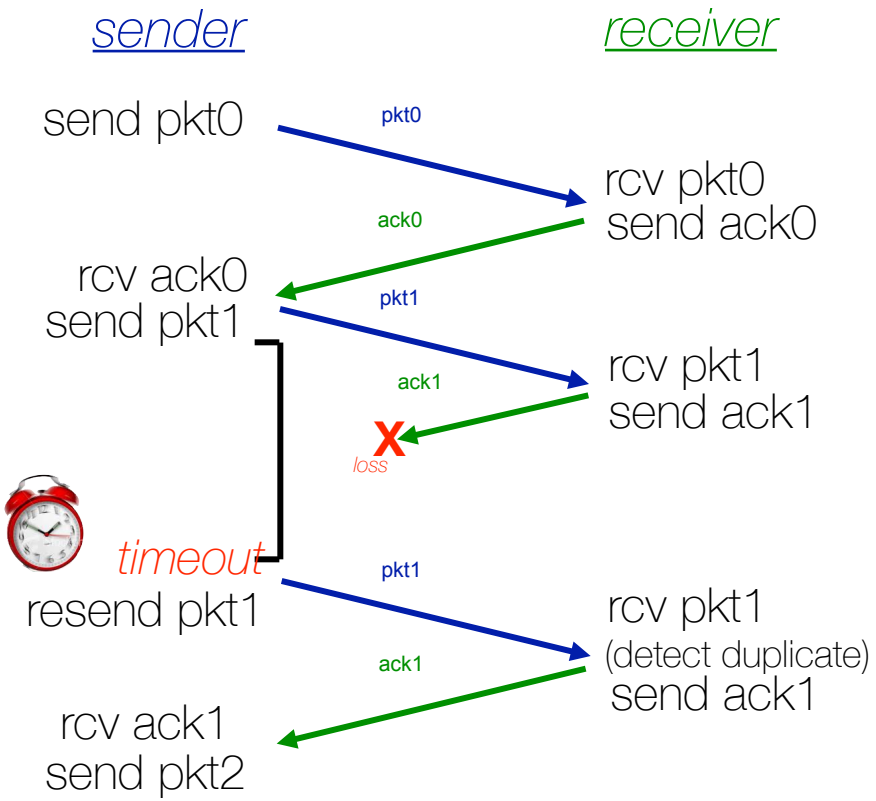


(a) no loss

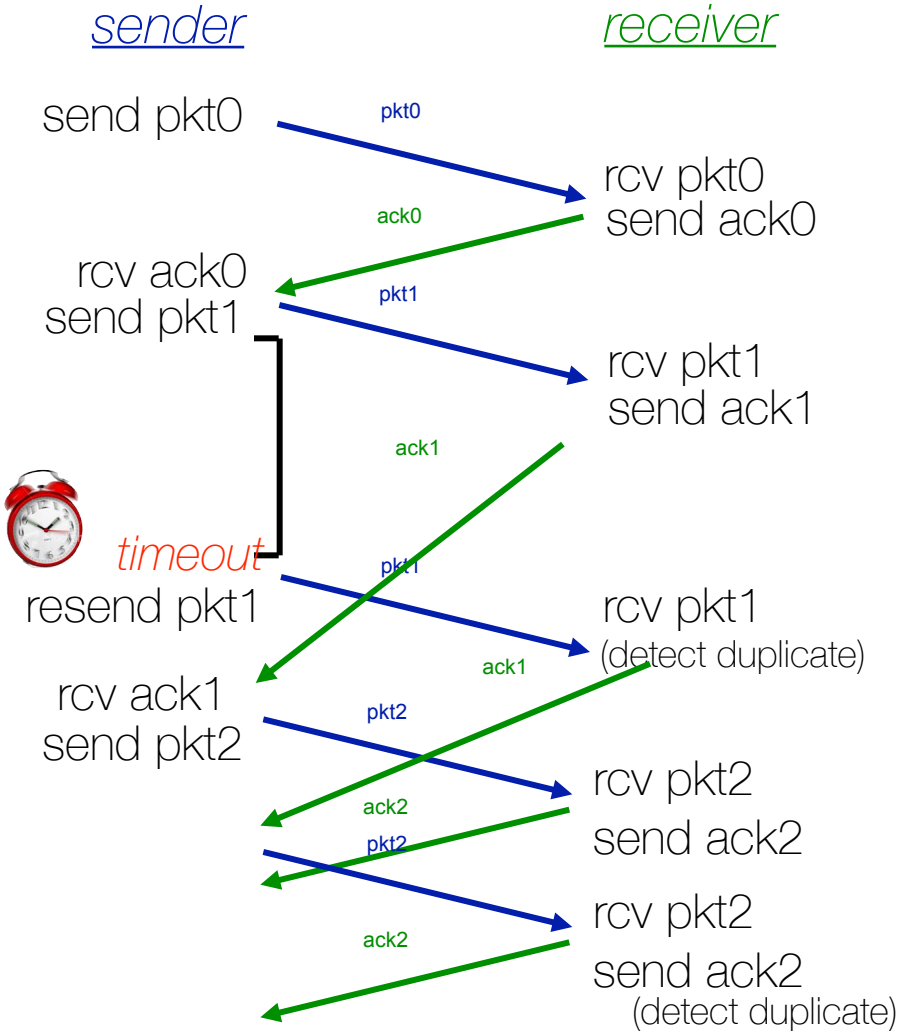


(b) packet loss

ACKs and time-outs in action



(c) ACK loss



(d) premature timeout/ delayed ACK

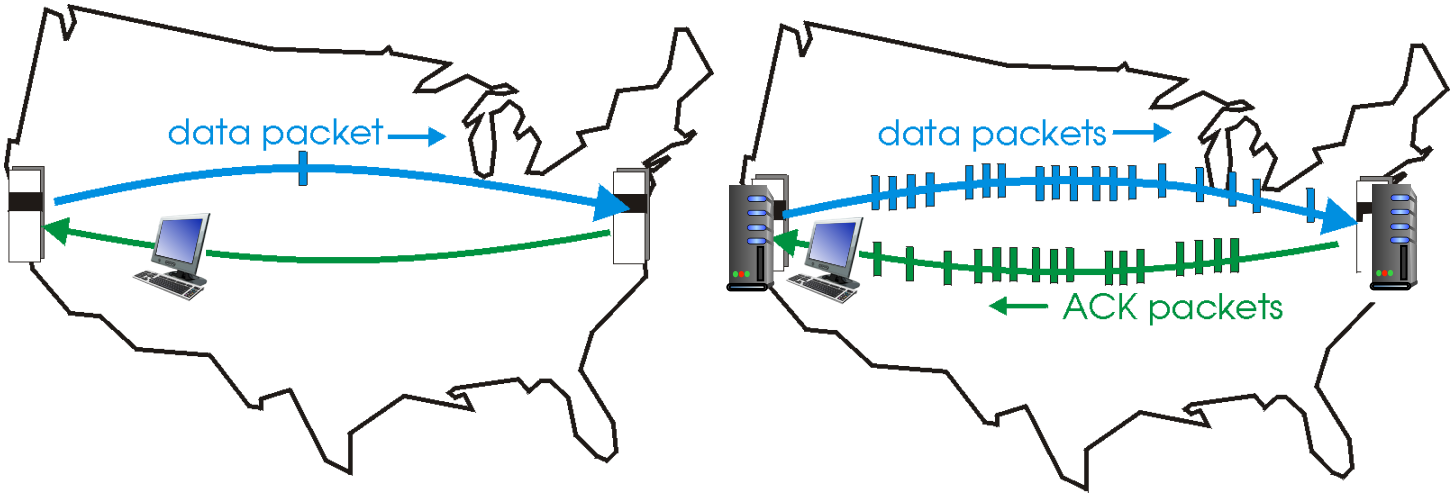
Latency kills this method

- Suppose we only allow one packet outstanding at a time
- Speed of light is $3x10^8 ms^{-1}$.
- Plenty of paths are $3x10^6 m$ (short transatlantic path to east coast).
- So 20ms round trip time, assuming infinitely fast routers and computers.
- 50 packets per second @1500bytes/packet = 75KB/sec ~ **600Kb/sec** maximum.

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

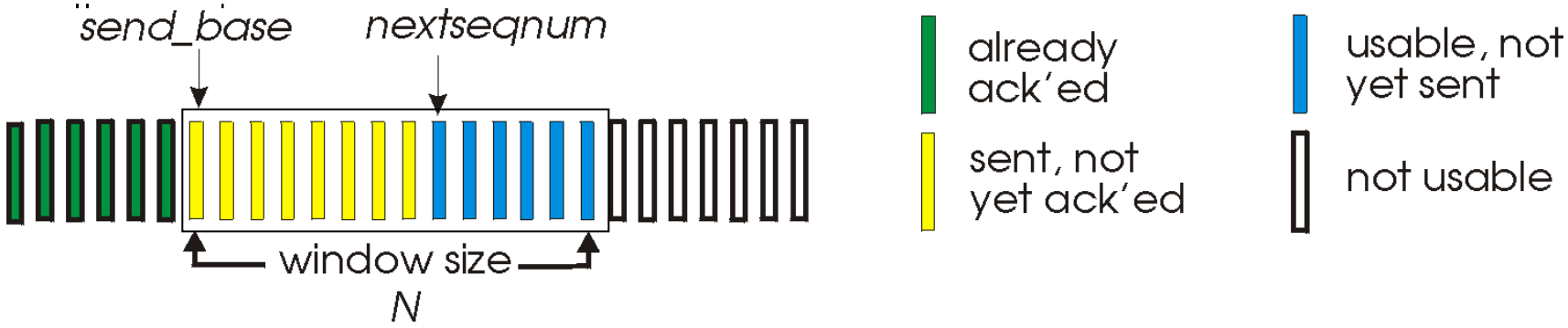
(b) a pipelined protocol in operation

Pipelined protocol: overview

- sender can have up to N un-acked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit all unacked packets

Pipelined Sender

- k-bit sequence number in packet header
- "send window" of up to N, consecutive unacked packets



- ACK(n): ACKs all pkts up to, including sequence number n - "*cumulative ACK*"
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher sequence number packets in window

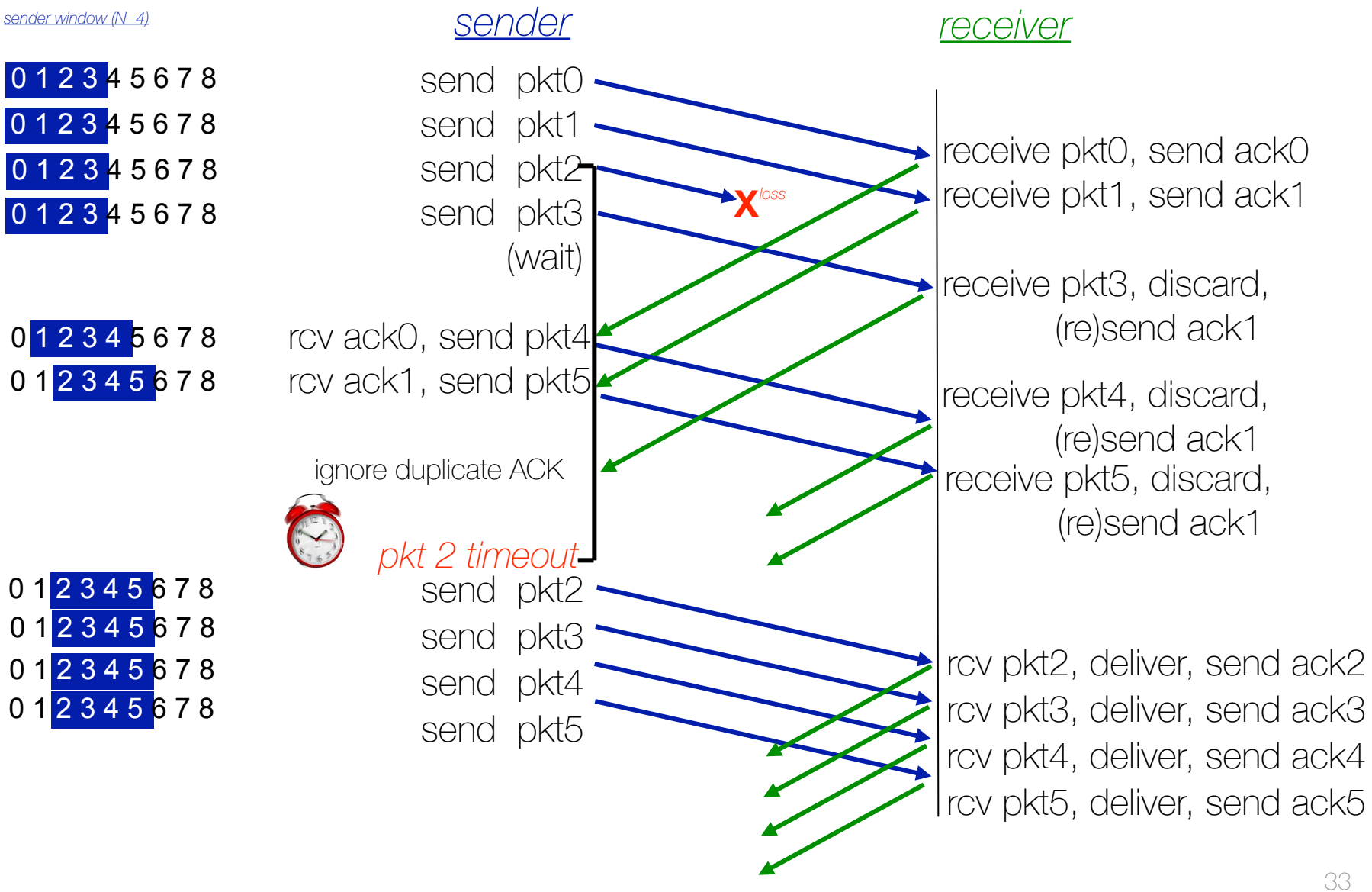
Pipelined Receiver

ACK-only: always send ACK for correctly-received packet with highest *in-order* sequence number

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

- out-of-order packets:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK packets with highest in-order sequence number

Pipelining in Action



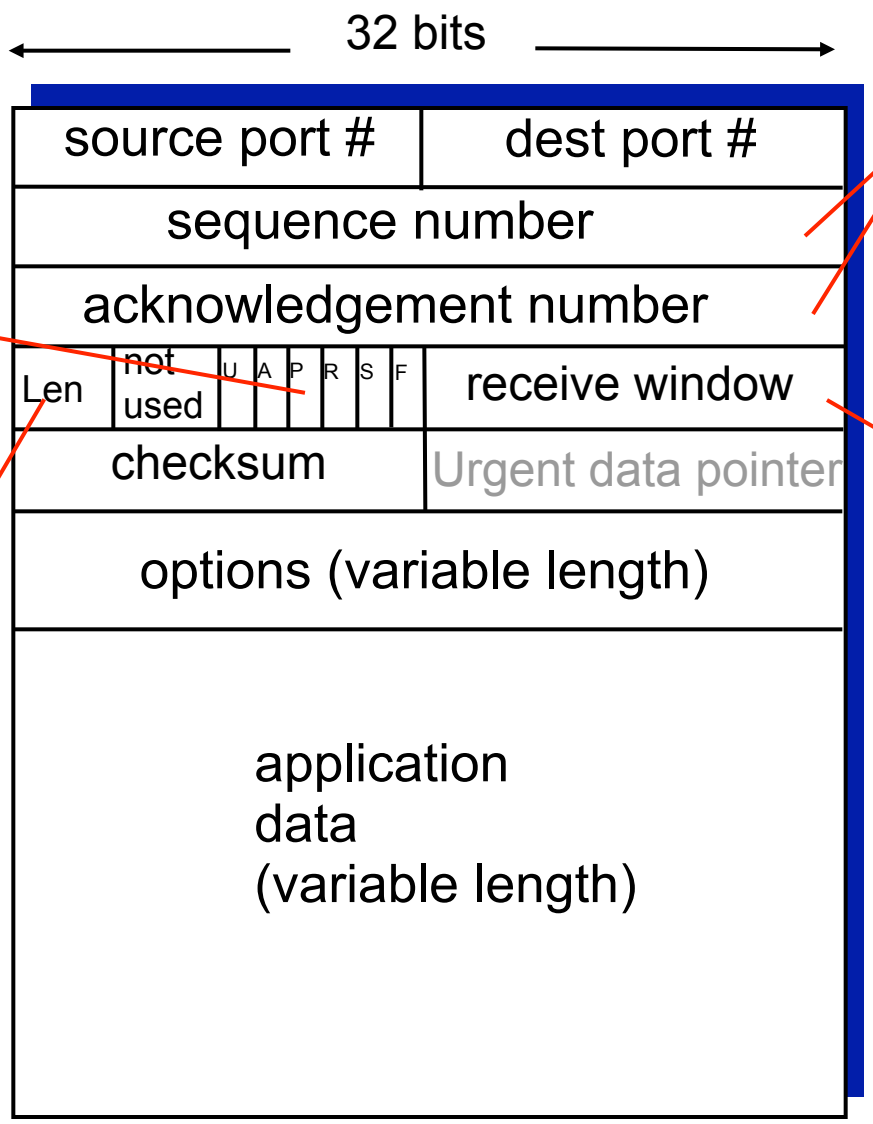
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no “message boundaries”
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

URGent
ACKnowledgement
PuSH
ReSeT
SYNchronise
FINish

Header Length in 32 bit words, to allow space for options. Payload is IP length - headers



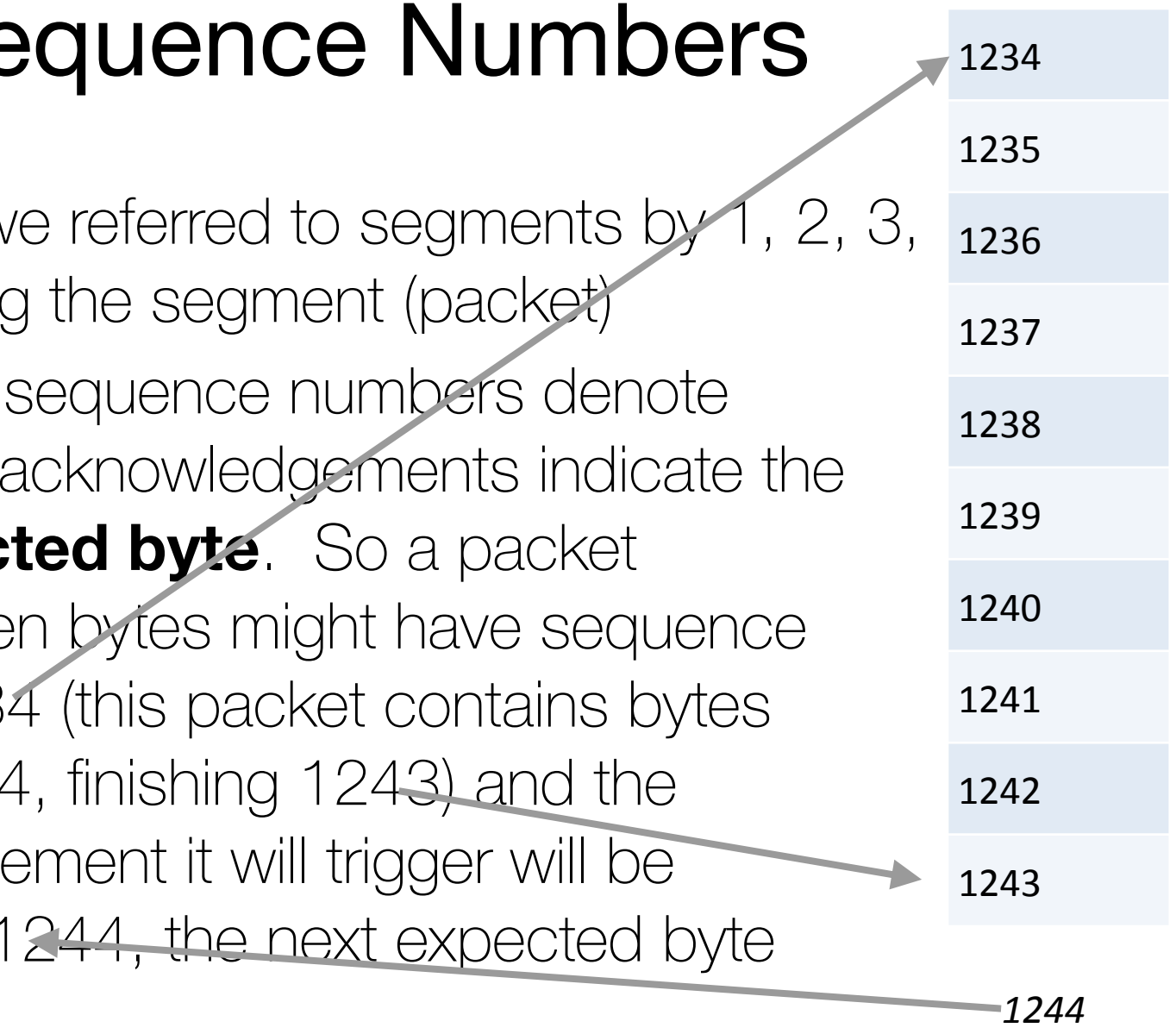
counting by bytes of data (not segments!)

bytes rcvr willing to accept

Urgent data pointer is not used for anything useful today. Which is a shame, because the receive window could very usefully be 32 bits.

Sequence Numbers

- Previously, we referred to segments by 1, 2, 3, 4, numbering the segment (packet)
- In fact, TCP sequence numbers denote **bytes**, and acknowledgements indicate the **next expected byte**. So a packet containing ten bytes might have sequence number 1234 (this packet contains bytes starting 1234, finishing 1243) and the acknowledgement it will trigger will be 1234+10 =1244, the next expected byte



TCP seq. numbers, ACKs

sequence numbers:

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

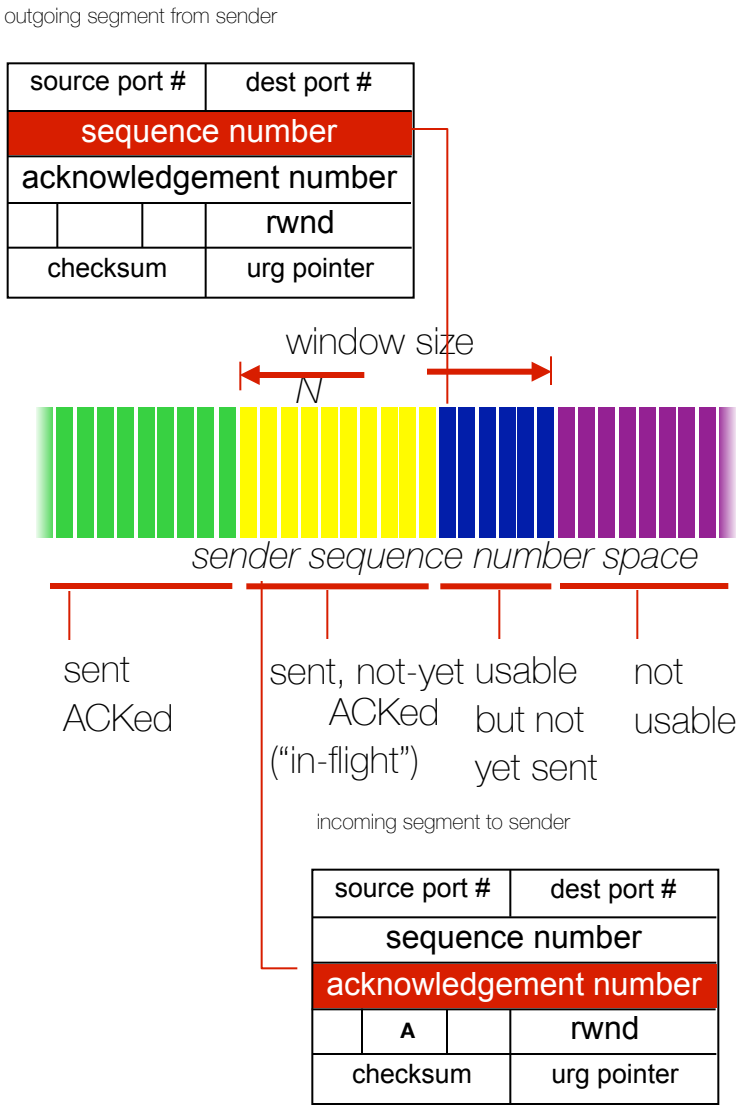
acknowledgements:

– seq # of next byte expected from other side

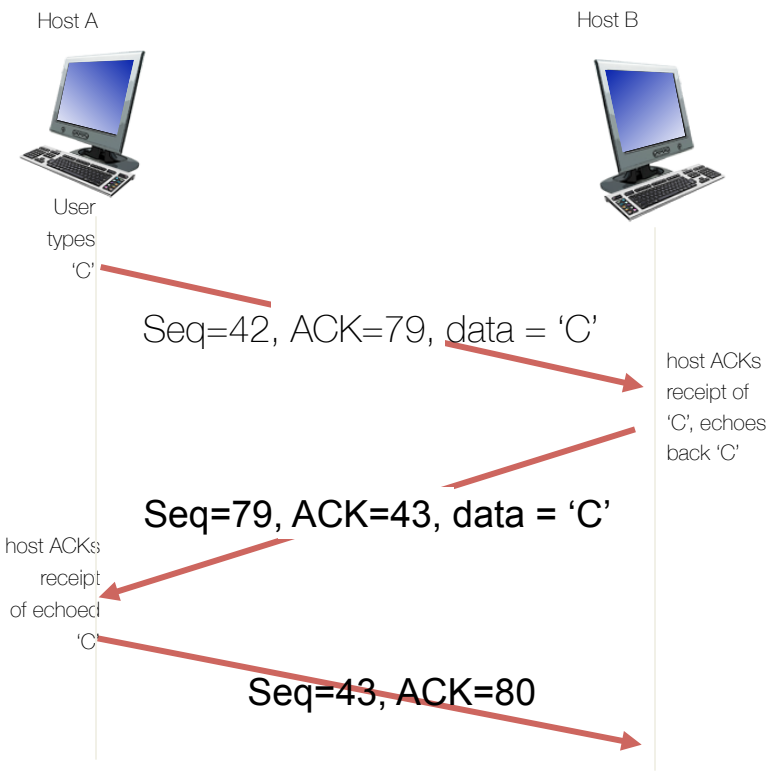
– cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs



simple telnet scenario

TCP round trip time, 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

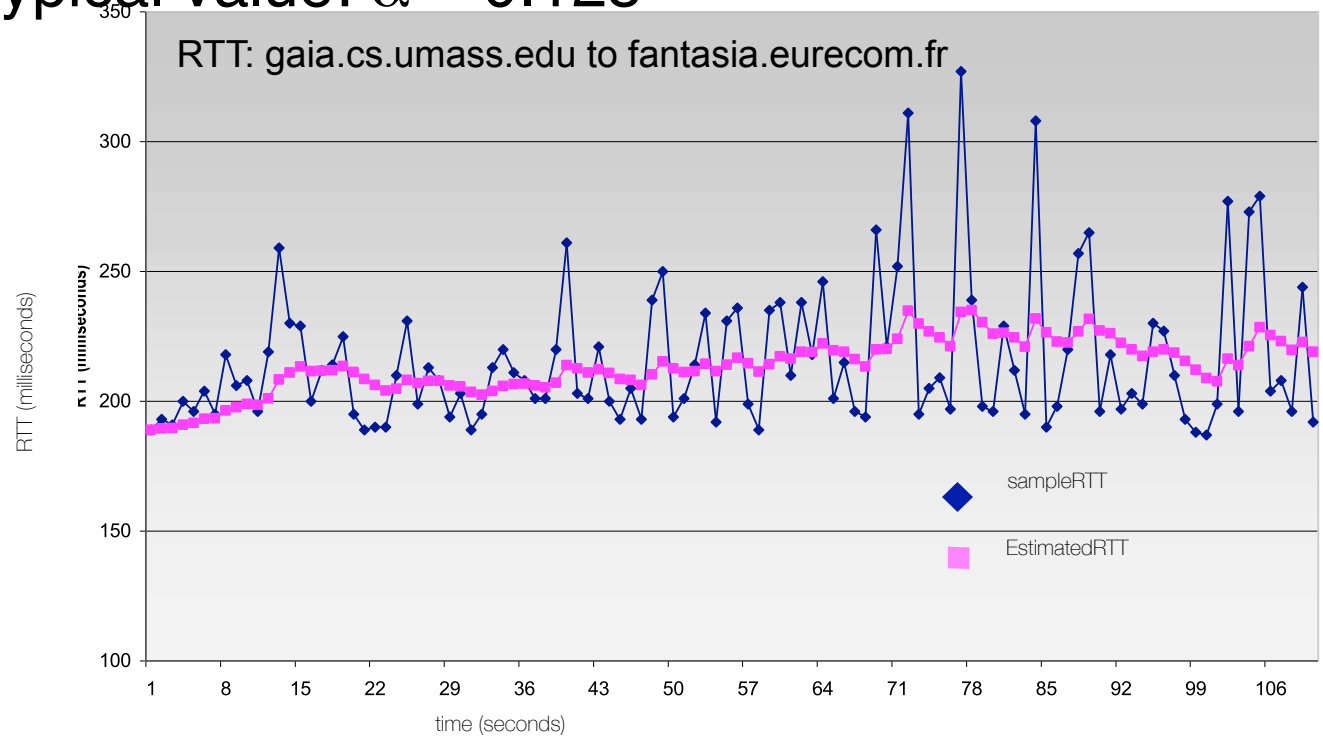
Q: how to estimate RTT?

- 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, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP reliable data transfer

- TCP creates **reliable data service** on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

- If waiting for an acknowledgement times out, either the segment was lost or the acknowledgement was lost: retransmit the segment
- If you receive two acknowledgements for segment X, the most likely reason is that segment X+1 has been lost but X+2 has been delivered, provoking the duplicate acknowledgement: retransmit segment X+1

TCP sender events:

Data received from app:

- create segment(s) each with seq number (seq number is byte-stream number of first data byte in segment)
- Send the segments(s) (assuming space in receive window, see "flow control" slide to come)
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

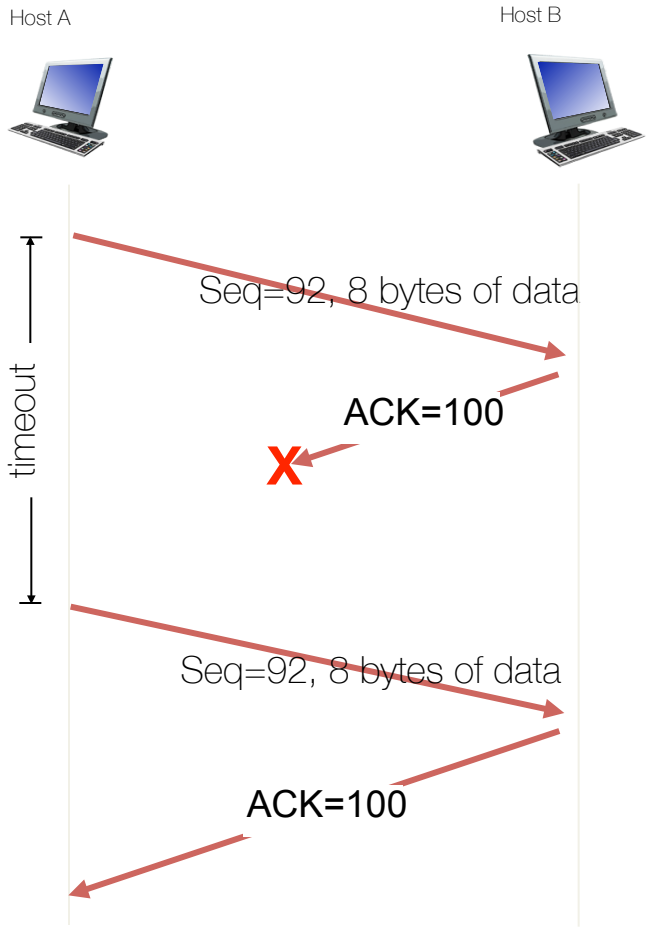
timeout:

- retransmit segment that caused timeout
- restart timer

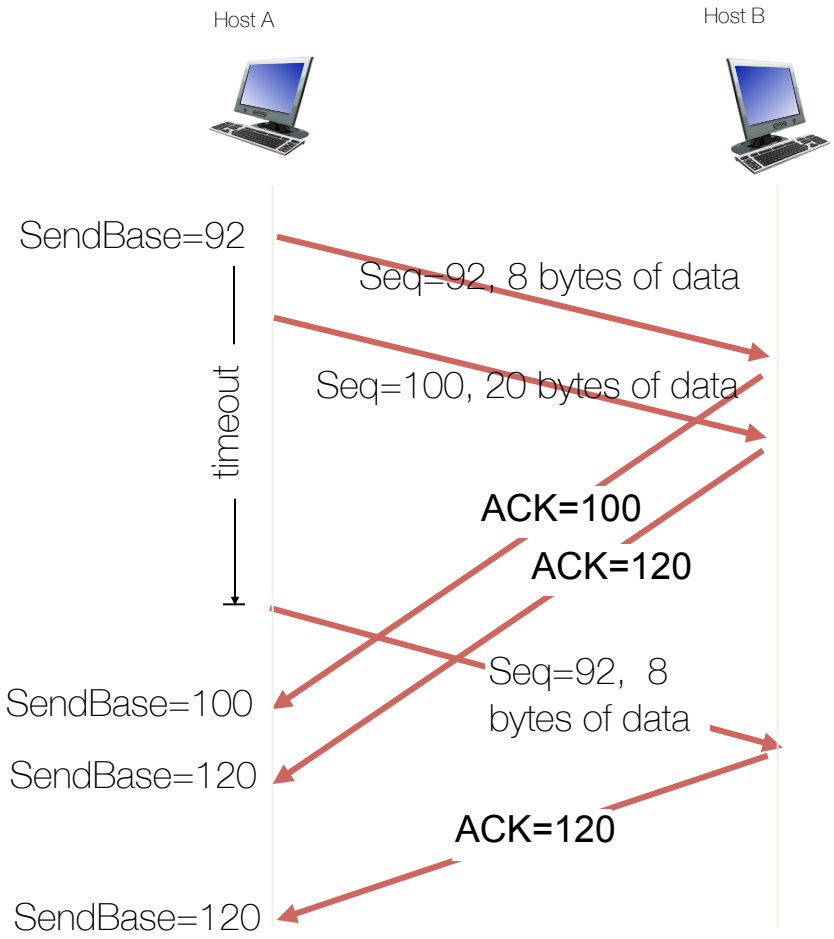
ack received:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

TCP: retransmission scenarios

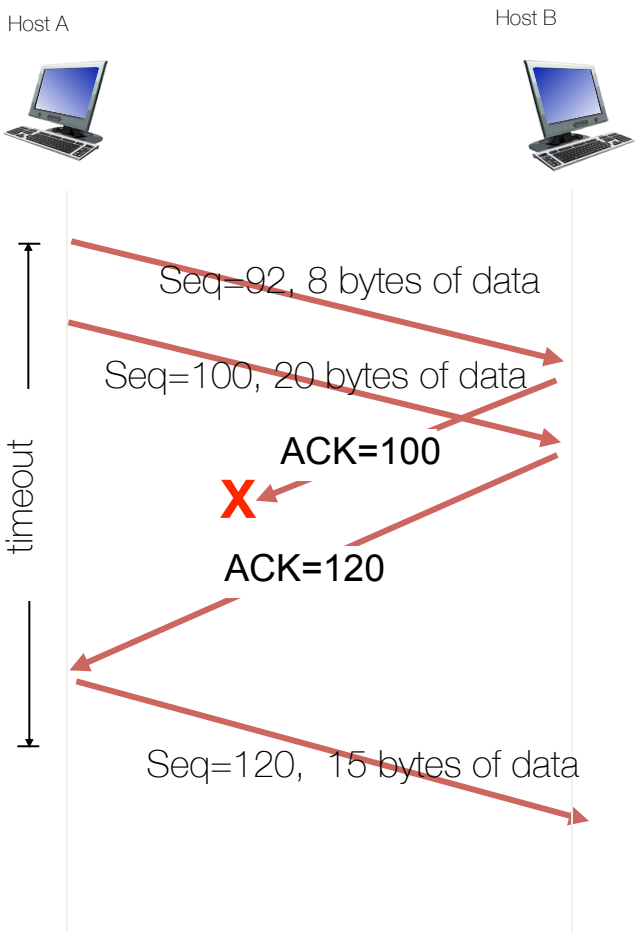


lost ACK scenario



premature timeout

TCP: retransmission scenarios

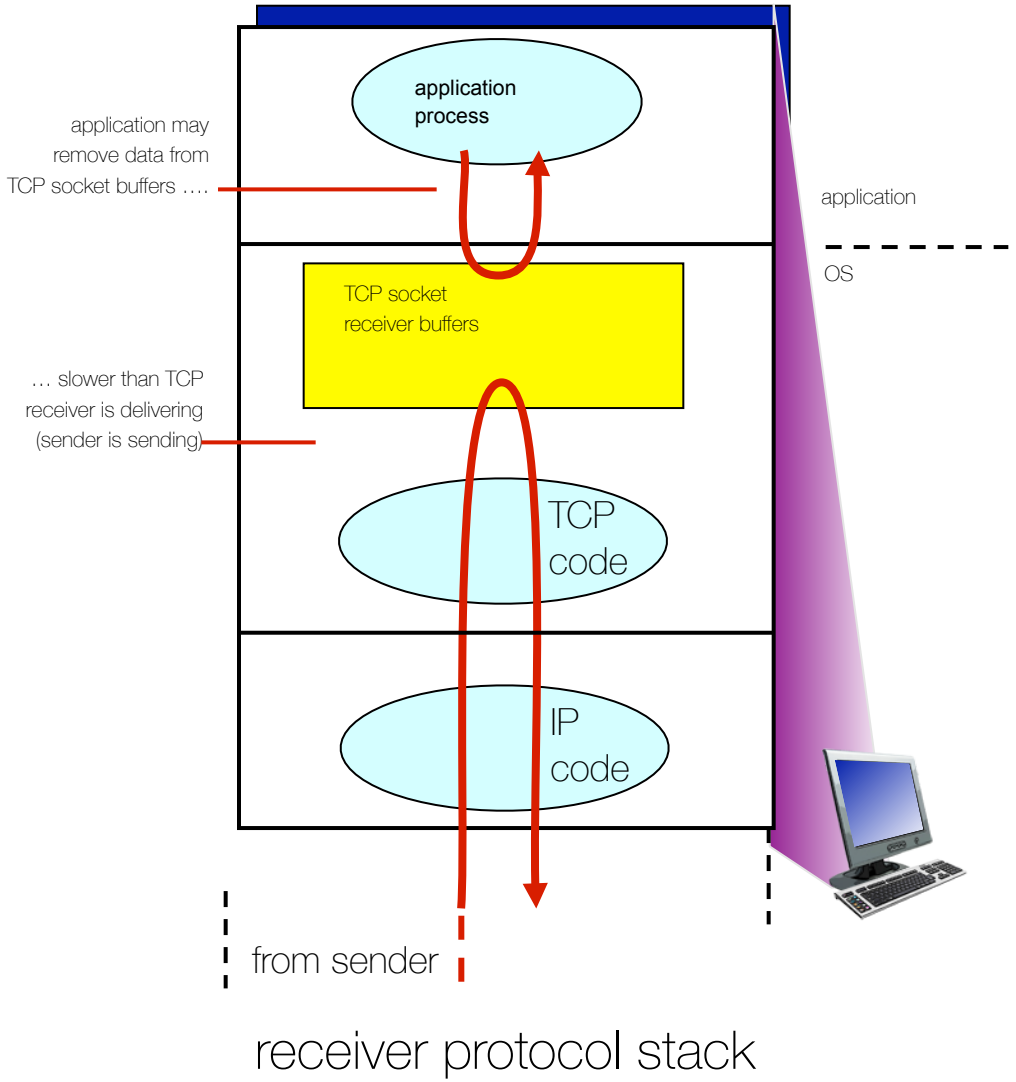


cumulative ACK

TCP ACK generation

Event at Receiver	Response
Arrival of in-order segment with expected sequence number, all data up to this point ack'd	Delated ack: wait up to 500,ms for next segment, then send ack
Arrival of in-order segment with expected sequence number, one other segment not ack'd	Immediately send one ack, acking everything up to the end of in-order data
Arrival of out-of-order segment with higher than expected sequence number: gap detected	Immediately send **duplicate ack**, indicating the next byte we were expecting
Arrival of segment that partially or completely fills a gap	Immediately send ack, provided segment starts at the beginning of the gap

TCP flow control



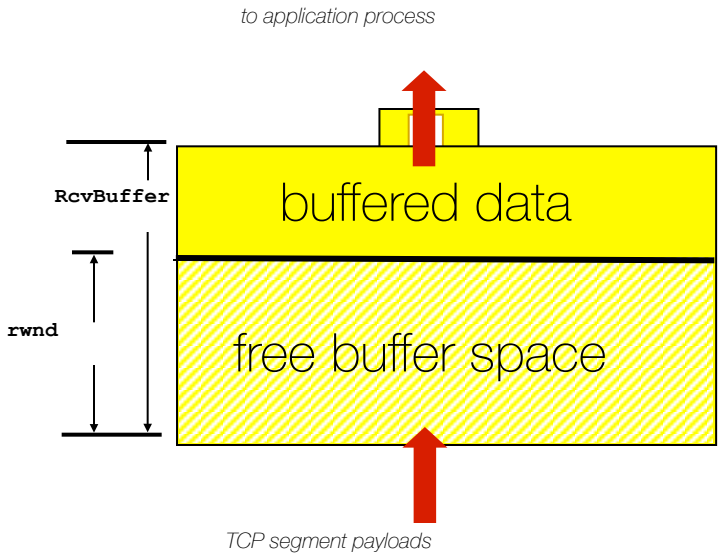
receiver protocol stack

flow control

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

TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 128k bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow

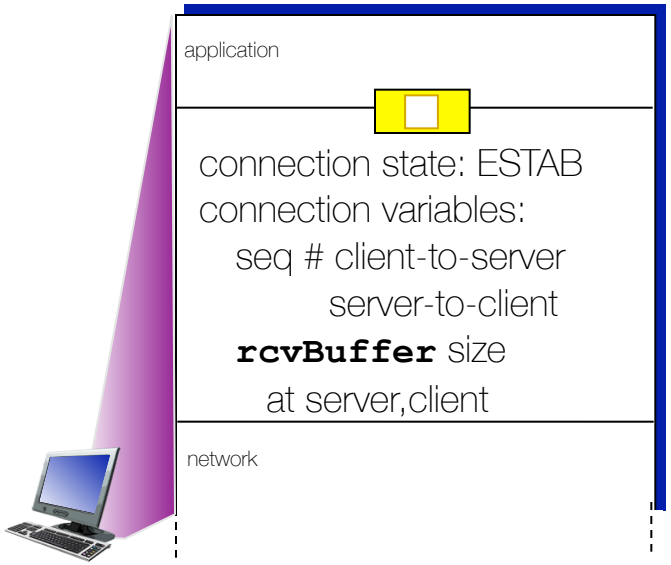


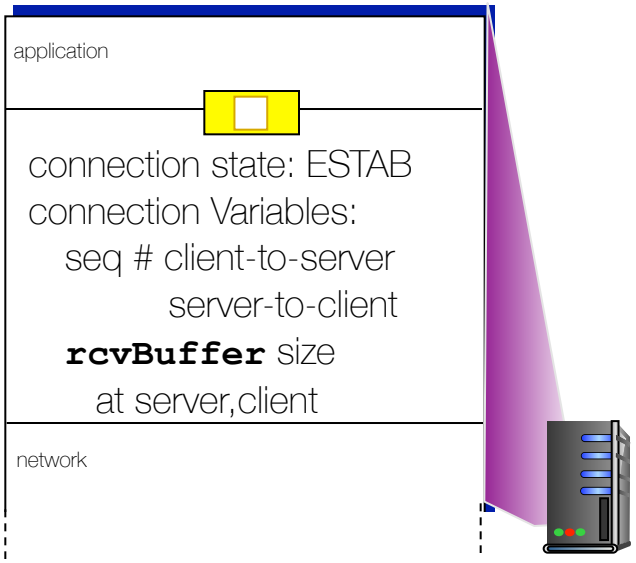
receiver-side buffering

Connection Management

before exchanging data, sender/receiver “handshake”:

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

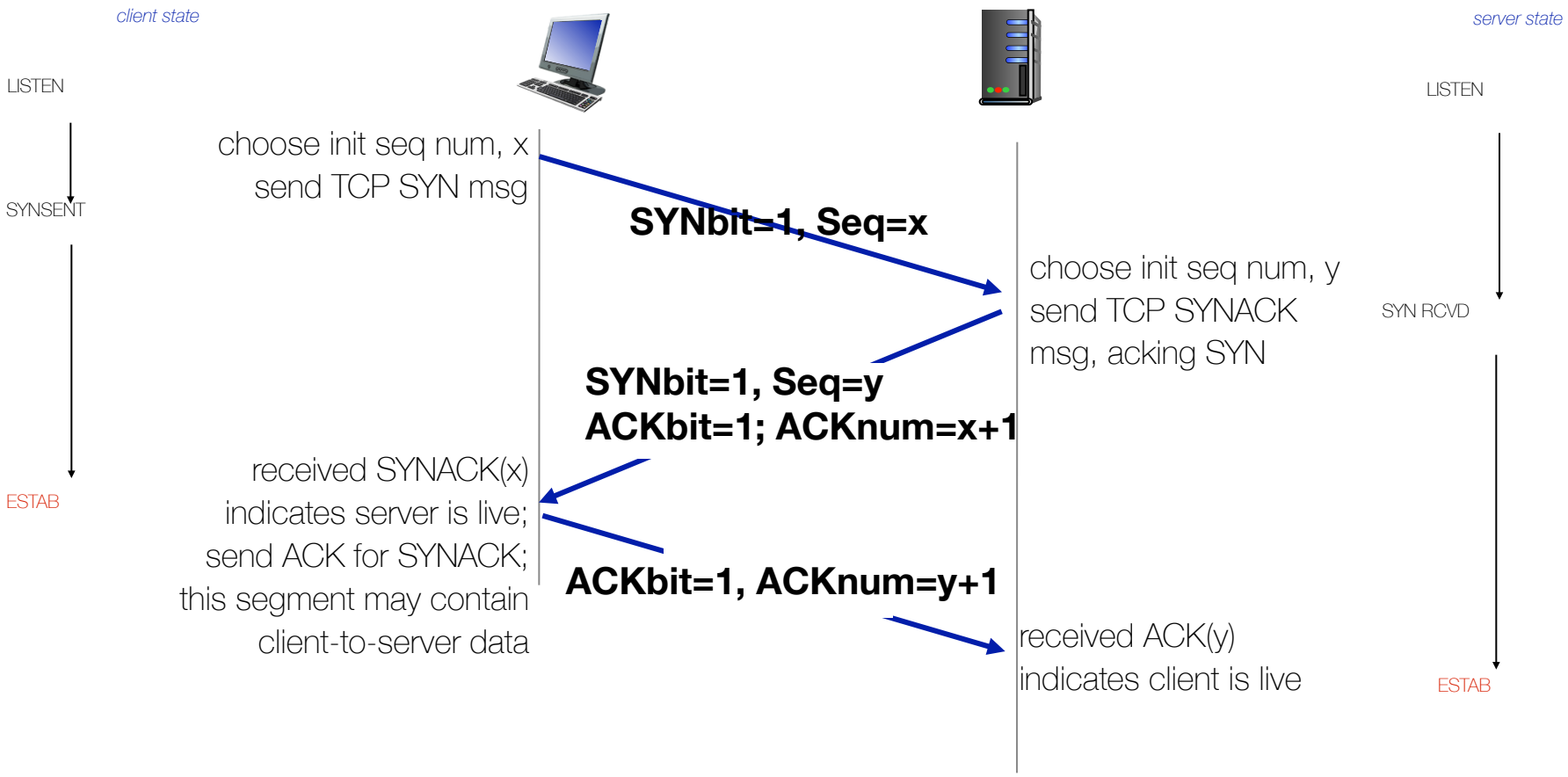





```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

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

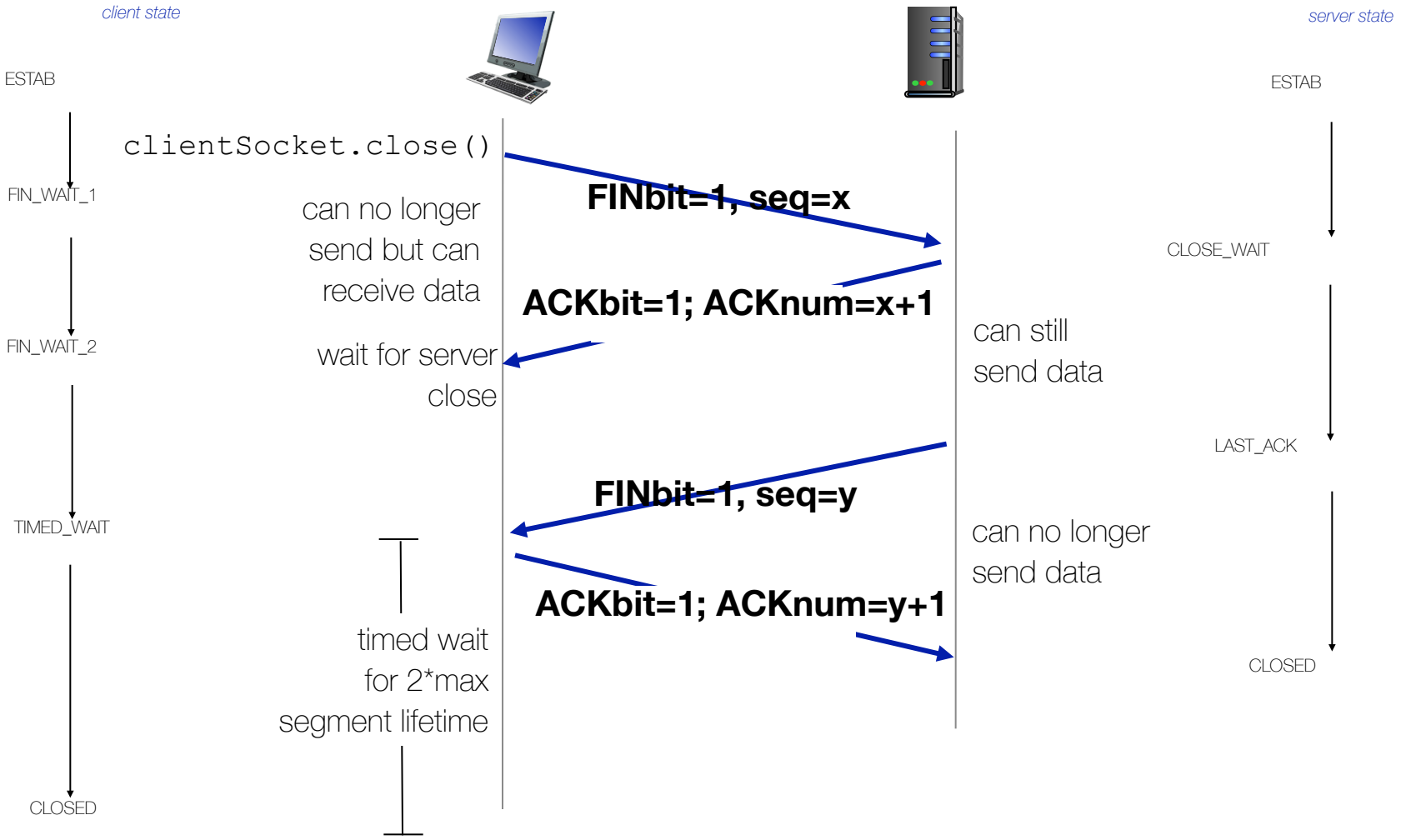
TCP 3-way handshake



TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection



Summary

- Creation
  - SYN/SYN-ACK/ACK
  - Both sides now have a sequence number.
- Shut down:
  - FIN/ACK/FIN/ACK
  - Might be condensed to FIN/FIN-ACK/ACK if acks delayed and/or fast machines

- Bulk transfer
  - Receiver advertises receive window (rwnd)
  - Sender transmits until rwnd bytes unack'd
  - Receiver acks contiguous segments, plus duplicate ack for out-of-order segments
  - Sender resends all unack'd data on timeout, or receipt of duplicate ack

Quick Quiz #2

- What acknowledgement is sent when TCP receives a segment after a gap (ie receives 1, 2, does not receive 3, receives 4)?
- Describe two ways for a TCP sender to know a transmitted segment has been lost?
- How many packets are there in a minimal TCP connection that passes no data and is then closed by the initiator, assuming standard Unix APIs?
- What is the URG bit used for in the header? The

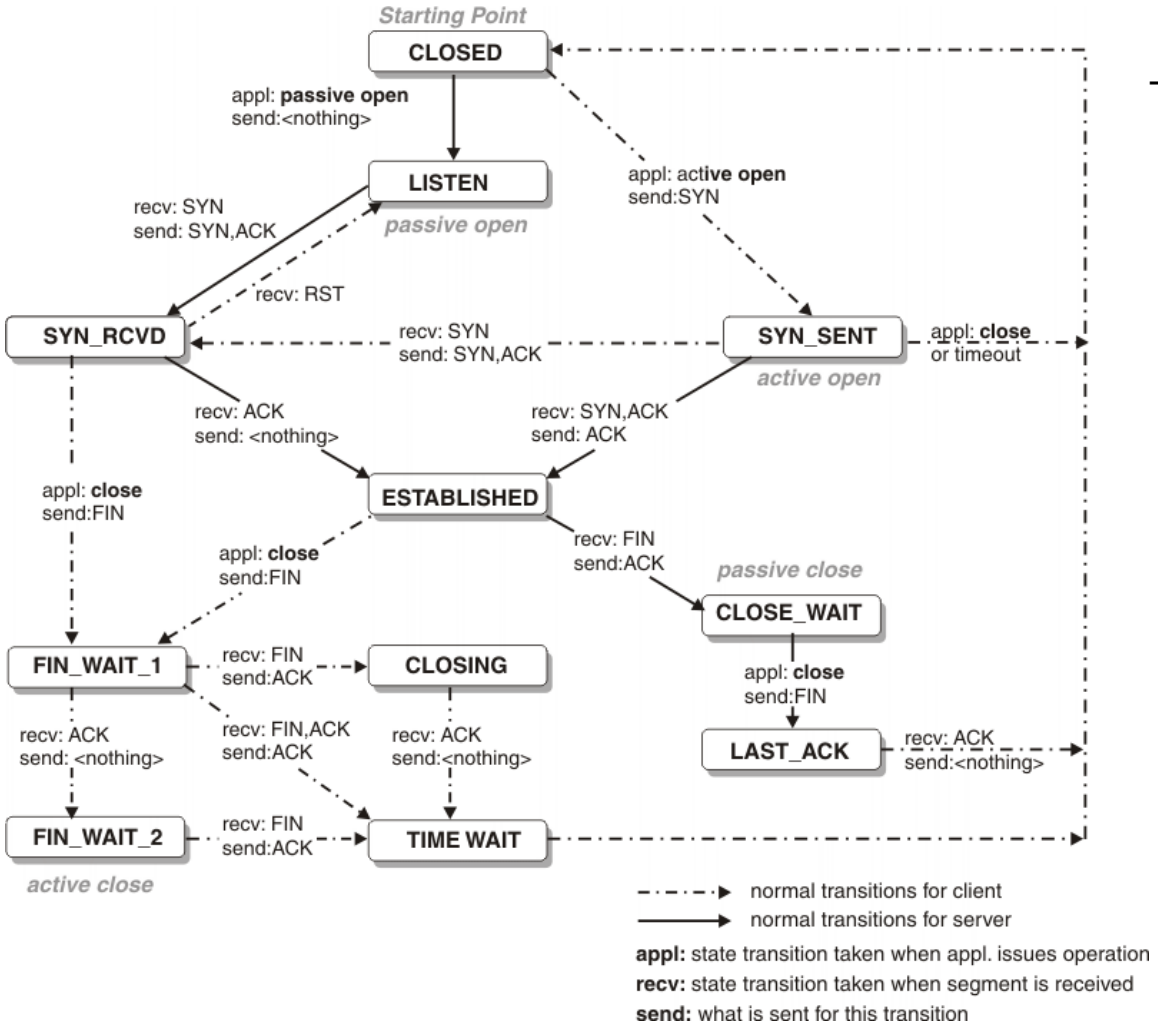
Quick Quiz #2 Answers

- The TCP receiver will send a duplicate ACK of segment 2 when segment 4 arrives. The ACK is **always** of the last **contiguous** packet.
- Firstly, there will be a timeout waiting for an acknowledgement. Secondly, the receipt of a duplicate acknowledgement is a strong clue the following packet has been lost (see q1).
- SYN, SYN+ACK, ACK, FIN, ACK, FIN, ACK = 7.
- Sometimes …FIN, FIN+ACK, ACK = 6
- Hypothetically SYN, SYN+ACK, ACK+FIN, FIN+ACK, ACK = 5, but you would need an API which supports "open a connection and immediately close it" to send the third packet.
- Both are redundant. URG is obsolete, PSH means "deliver now" and is usually set on all packets and ignored on receipt.

TCP Summary

- A connection established by exchange of initial sequence numbers, and acknowledging them (SYN, SYN+ACK, ACK).
- Each side advertises a current receive window in every segment it sends.
- Senders can send up to the lower of their willingness to buffer unacknowledged data and the other end's current receive window.
- Acknowledgements acknowledge the last contiguous byte plus one (the first expected byte)
- Timeout triggers retransmission of some or all outstanding data
- Receiver can optionally buffer out-of-order packets.
- Connection is closed in an orderly fashion by the exchange of FINs and acknowledging them (FIN, ACK, FIN, ACK or FIN, FIN+ACK, ACK). All data has been successfully acknowledged after this completes.
- Connections can be terminated at any time with an RST reset. There are no guarantees of how much data has been successfully received after this happens.

TCP State Diagram



This diagram, or similar, is very commonly seen is descriptions of TCP. I include it for information, but I'm never sure how useful it is: the complexities and performance issues are all in the bulk transfer ("ESTABLISHED") rather than the minutiae of how the connections are established and torn-down.

TCP Options

- I quickly glossed over the header length field, which is usually 5. It is measured in 32 bit words (grandad), so 5 covers the usual 20-byte TCP header.
- If it is greater than 5, options can be inserted after the main header, usually at SYN-time in order to specify particular optional features and values.

More TCP Issues

- Options
  - SACK enables selective acknowledgements
  - Window Scaling deals with bandwidth*delay > 64K
  - Timestamping for PAWS (protection against wrapped sequence numbers) and RTTM (round trip time measurement)
- Algorithms
  - Slow Start to fill the pipeline smoothly
  - Fast Retransmission of dropped packets
  - Rapid Recovery to avoid Slow Start when it is not necessary.

Selective Acknowledgement

- Instead of sending a cumulative ACK, instead send a set of “upto…starting from…” pairs.
- Allows the acknowledgement to indicate that parts of the sequence space have been received.
- Other algorithms (especially fast retransmission) have made it less useful, but it is widely supported anyway.
- Usual wisdom is that it’s not well tuned, and the benefits of a well-tuned “ordinary” TCP flow outweigh the benefits of SA.

Window Scaling

- TCP “Receive Window” is advertised in every packet, and indicates how much data the receiver is willing to accept right now.
- It is a 16 bit quantity, so limited to 64k.
- At 100MBps (GigE), this means any network with RTT>655 µs, which is <100km point to point, cannot fill the pipeline (ie, will have to stop and wait for ack)
- 40GigE: 2.5km
- In reality, RTT includes time taken to generate an ack, so limit is far lower.

Window Scaling

- Solution is to negotiate a scaling factor for the receive window
- Increases effective window size to 2^30 (shift window left by up to 14 bits)
- Allows bandwidth*delay up to 1GB.
- Hence GigE up to 10s RTT, or 40GigE up to 250ms RTT.

PAWS

- Protection against wrapped sequence numbers
- If you have a receive window that is approaching 1GB then with small amounts of reordering and delay, you can have duplicate sequence numbers on packets which are 4GB apart in the sequence space.
- “Time”stamp in header is monotonically increasing, every 1ms or less.
- This allows you to distinguish between “new” and “old” packets with same sequence no.
- 32 bit counter running at 1000 Hz takes 50 days to wrap around!

RTTM

- Round Trip Time Measurement, also known generically as “Timestamping”
- Measuring RTT can be difficult in the face of retransmissions
- Instead place timestamp in header of data segments (also used for PAWS) and reflect that timestamp back in an ACK.
- Subtract the ACK timestamp from current value of counter to get RTT. Simples!

Slow Start

- In many environments, the local network is faster than the wide area network.
  - Untrue in 2021 for common “Wifi plus gigabit” scenarios, but true for most of the last 40 years.
- So a large advertised receive window (possibly scaled) will result in the sender potentially sending large amounts of data to the local router as soon as the connection starts, which it then has to clock it out at a slower speed.
- Potential packet loss (limited router memory), does not make effective use of the whole path.
- Pathological case: dump a lot of data into local router, stuff is dropped, retransmit it all at the same rate, permanent loss.

Slow Start

- Once connection is established, do not initially send complete receive window.
- Instead, only send one packet, and increase the number of packets sent by one for every acknowledgement received, until the window is full (or double the number of packets, or some other algorithm).
- Conceptually, this spreads out the packets in the network, and reduces congestion at intervening routers.

Slow Start

- Many different algorithms (increase by one, double each time, etc)
- All maintain a local “congestion window” which is less than or equal to the receive window advertised by the other end.
- In general, modern implementations ramp up **much** faster than older ones, to the point the effect may be almost invisible.
- Compulsory, but much less important today.

Rapid Recovery / Fast Retransmit

- Various algorithms are implemented in various versions of various operating systems, with names like “Tahoe” and “Reno”.
- Uses receipt of duplicate acks to infer that subsequent packet(s) have been lost, and retransmits them right now rather than waiting for timeout.
- Avoids going into slow start, so that window is kept full.
- Often just retransmits one segment on the assumption that was lost and the other end has buffered the following segments.

Silly Window Syndrome

- Window is zero (receiver's buffers are full)
- Receiving application is reading one byte at a time
- Receive window is now 1 byte, so ACK sent opening window by 1
- Sender sends one more byte
- We are now using two complete TCP frames every time one byte is read

Two solutions

- Receiver: do not advertise small increases in window, but only advertise when either we can accept one more segment, or half our buffer.
- Sender ("Nagle's algorithm"): do not send small packets when there is outstanding unack'd data (more complex in reality: RFC896).

Quick Quiz #3

- Why is window-scaling necessary on fast networks?
- Why is slow start less important today than it was in 2010?
- What does rapid retransmission avoid?
- What two purposes do timestamps in TCP packets serve?

Quick Quiz #3 Answers

- Window scaling is necessary because the 16 bit receive window field is not large enough for long, fast networks.
- Slow start helps routers with limited memory located between fast and slow network segments: WANs now approach or exceed practical LAN speeds.
- Rapid retransmission keeps connections out of slow start, and avoids retransmitting all outstanding packets (which may be huge with a large window)
- They help measure round trip times more accurately, and they disambiguate sequence numbers on long fast connections.

Concrete TCP

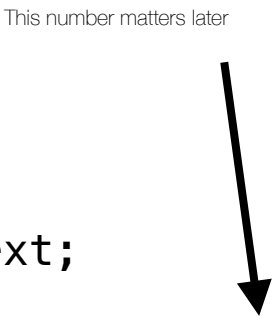
- I find it easier to understand things from real examples.
- I hope this helps.

Simple Daemon

```
my $listen = new IO::Socket::INET (LocalPort => $port,
  Proto => "tcp", Listen => 5,
  ReuseAddr => 1) or die "cannot listen $!";
warn "listening on $port";


while (my $s = $listen->accept ()) {
  my $pid = fork ();
  die "fork $!" unless defined ($pid);
  if ($pid == 0) {
     warn "connection accepted";
     $listen->close ();
     sendfiles ($s, @ARGV);
     exit (0);
  } else {
     $s->close ();
  }
}
```

sendfiles




```
sub sendfiles {
  my $s = shift;
  while (my $f = shift) {
     my $fh = new IO::File ("<$f") or next;
     my $buffer;
     while (my $bytes = $fh->sysread ($buffer, 8192)) {
        $s->syswrite ($buffer, $bytes) or die "cannot write $!";
     }
  }
}
```

Connection Refused

```
igb@pi-two:~$ telnet pi-one 5758
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
telnet: Unable to connect to remote host: Connection refused
igb@pi-two:~$
```

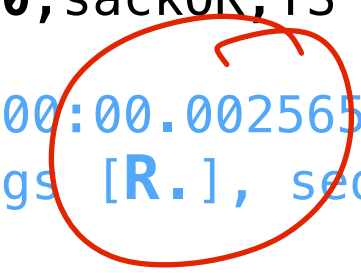
Connection Rejected

```
00:00:00.000000 IP6
2001:8b0:129f:a90f:ba27:ebff:fec9:acae.41529 >
2001:8b0:129f:a90f:ba27:ebff:fe00:efe7.5758: Flags [S], seq
1698998887, win 28000, options [mss 1400,sackOK,TS val
19928636 ecr 0,nop,wscale 6], length 0

00:00:00.000358 IP6
2001:8b0:129f:a90f:ba27:ebff:fe00:efe7.5758 >
2001:8b0:129f:a90f:ba27:ebff:fec9:acae.41529: Flags [R.], seq
0, ack 1698998888, win 0, length 0

00:00:00.002250 IP 10.92.213.238.44781 > 10.92.213.231.5758:
Flags [S], seq 1260504038, win 29200, options [mss
1460,sackOK,TS val 19928636 ecr 0,nop,wscale 6], length 0

00:00:00.002565 IP 10.92.213.231.5758 > 10.92.213.238.44781:
Flags [R.], seq 0, ack 1260504039, win 0, length 0
```



FLAG ● = ACK

Connection Immediately Closed

```
igb@pi-two:~$ telnet pi-one 5757
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
Connected to pi-one.
Escape character is '^]'.
Connection closed by foreign host.
igb@pi-two:~$
```

Immediate Close

```
00:00:00.002258 IP 10.92.213.238.49560 > 10.92.213.231.5757: Flags [S],
seq 2044006020, win 29200, options [mss 1460,sackOK,TS val 19986406 ecr
0,nop,wscale 6], length 0
00:00:00.002595 IP 10.92.213.231.5757 > 10.92.213.238.49560: Flags [S.],
seq 894665383, ack 2044006021, win 28960, options [mss 1460,sackOK,TS val
19986382 ecr 19986406,nop,wscale 6], length 0
00:00:00.003596 IP 10.92.213.238.49560 > 10.92.213.231.5757: Flags [.],
ack 1, win 457, options [nop,nop,TS val 19986406 ecr 19986382], length 0
00:00:00.018316 IP 10.92.213.231.5757 > 10.92.213.238.49560: Flags [F.],
seq 1, ack 1, win 453, options [nop,nop,TS val 19986384 ecr 19986406],
length 0
00:00:00.019698 IP 10.92.213.238.49560 > 10.92.213.231.5757: Flags [F.],
seq 1, ack 2, win 457, options [nop,nop,TS val 19986408 ecr 19986384],
length 0
00:00:00.021259 IP 10.92.213.231.5757 > 10.92.213.238.49560: Flags [.],
ack 2, win 453, options [nop,nop,TS val 19986384 ecr 19986408], length 0
```

Question

- I said in previous lecture minimum TCP session SEVEN packets (SYN, SYN/ACK, ACK, FIN, ACK, FIN, ACK).
- Here there are only SIX
- Why?

Delayed Acks at Work

- After receipt of the first FIN, the caller has nothing to say, and no outstanding un-ack'd data
- So ACK is sent only when there is traffic to pass, in this case the answering FIN.
- Hence the close is FIN, FIN/ACK, ACK.
- Requires client to call close() within 500ms of receipt of FIN, with no other traffic outstanding.

Slightly Delayed Close

```
#!/usr/bin/perl -w

use warnings;
use strict;
use IO::Socket::INET;

my $s = new IO::Socket::INET (PeerAddr => "10.92.213.231:5757",
Proto => "tcp") or die "$!";

sleep (2);

exit (0);
```

Typical Seven Packets

```
00:00:00.000000 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [S], seq
1156157459, win 29200, options [mss 1460,sackOK,TS val 20078528 ecr 0,nop,wscale
6], length 0
00:00:00.000413 IP 10.92.213.231.5757 > 10.92.213.238.49583: Flags [S.], seq
1726094991, ack 1156157460, win 28960, options [mss 1460,sackOK,TS val 20078505 ecr
20078528,nop,wscale 6], length 0
00:00:00.001393 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [.], ack 1, win
457, options [nop,nop,TS val 20078528 ecr 20078505], length 0
// 457*2^6 = 29248
00:00:00.014287 IP 10.92.213.231.5757 > 10.92.213.238.49583: Flags [F.], seq 1, ack
1, win 453, options [nop,nop,TS val 20078506 ecr 20078528], length 0
00:00:00.023675 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [.], ack 2, win
457, options [nop,nop,TS val 20078531 ecr 20078506], length 0
// TWO SECOND DELAY
00:00:02.002245 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [F.], seq 1, ack
2, win 457, options [nop,nop,TS val 20078728 ecr 20078506], length 0
00:00:02.002556 IP 10.92.213.231.5757 > 10.92.213.238.49583: Flags [.], ack 2, win
453, options [nop,nop,TS val 20078705 ecr 20078728], length 0
```

Window Scaling

- Note Window Scale has been set to 6

```
options [mss 1460,sackOK,TS val 19986406 ecr 0,nop,wscale 6]
```

- Both implementations are the same (Linux, recent kernel)
- Win is being sent as ~500, 500 * 2^6 = ~32000, which is a typical value for a Receive Window

Wireshark usually figures out the scale, but tcpdump is packet-by-packet.

Other Systems Are Different

- Solaris requires more provocation to select Window Scaling, especially on local area network
- Can be Asymmetric

```
00:07:51.826744 IP 10.92.213.241.44205 > 10.92.213.231.5757:
Flags [S], seq 87151905, win 64240, options [mss 1460,sackOK,TS
val 19039311 ecr 0,nop,wscale 1], length 0
00:07:51.827087 IP 10.92.213.231.5757 > 10.92.213.241.44205:
Flags [S.], seq 3263477770, ack 87151906, win 28960, options [mss
1460,sackOK,TS val 20196188 ecr 19039311,nop,wscale 6], length 0
```

Bulk Data

```
igb@pi-two:~$ telnet pi-one 5757
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
Connected to pi-one.
Escape character is '^]'.
pi-one.home.batten.eu.org, DNS, DHCP, NTP (GPS), heyu, odds and ends.

*** This is a private machine.  If you are not personally authorised by
*** igb@batten.eu.org then your access is illegal.

Connection closed by foreign host.
igb@pi-two:~$
```

Sending small amounts of data

```
SYN SYN/ACK ACK as usual, options deleted for clarity

Data Packet:

00:03:11.671436 IP 10.92.213.231.5757 > 10.92.213.238.49591:
Flags [P.], seq 1:196, ack 1, win 453, length 195

Ack:

00:03:11.672488 IP 10.92.213.238.49591 > 10.92.213.231.5757:
Flags [.], ack 196, win 473, length 0

Start of shut down:

00:03:11.675722 IP 10.92.213.231.5757 > 10.92.213.238.49591:
Flags [F.], seq 196, ack 1, win 453, length 0
```

Slightly more data

- Notice mss is 1460, so anything more than 1400 bytes will require more than one packet.
- Let's try 2KB

```
igb@pi-one:~$ i=1;while [[ $i -lt 2048 ]]; do echo -n 0; let
i=i+1; done > 2kfile
igb@pi-one:~$
```

```
igb@pi-two:~$ telnet pi-one 5757
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
Connected to pi-one.
Escape character is '^]'.
00000000…0Connection closed by foreign host.
igb@pi-two:~$
```

```
// Note caller is now Solaris, wscale 1, options deleted
00:00:00.002792 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 1, win 64436, length 0
00:00:00.021531 IP 10.92.213.231.5757 > 10.92.213.241.63887:
Flags [.], seq 1:1449, ack 1, win 453, length 1448
00:00:00.022776 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 1449, win 64436, length 0
00:00:00.024380 IP 10.92.213.231.5757 > 10.92.213.241.63887:
Flags [P.], seq 1449:2048, ack 1, win 453, length 599
00:00:00.025245 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 2048, win 64436, length 0
00:00:00.027568 IP 10.92.213.231.5757 > 10.92.213.241.63887:
Flags [F.], seq 2048, ack 1, win 453, length 0
00:00:00.028577 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 2049, win 64436, length 0
00:00:00.028884 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [F.], seq 1, ack 2049, win 64436, length 0
```

Slightly more data

```
igb@pi-one:~$ i=0;while [[ $i -lt 2048 ]]; do echo
0123456789ABCDE; let i=i+1; done > 32kfile
igb@pi-one:~$ ls -lt 32kfile
-rw-r--r-- 1 igb igb 32768 Feb 12 06:54 32kfile
igb@pi-one:~$
```

```
igb@zone-host:~$ more /tmp/32k
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
telnet: connect to address
2001:8b0:129f:a90f:ba27:ebff:fe00:efe7: Connection refused
Trying 10.92.213.231...
Connected to pi-one.home.batten.eu.org.
Escape character is '^]'.
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
…
0123456789ABCDE
0123456789ABCDE
Connection to pi-one.home.batten.eu.org closed by foreign h
igb@zone-host:~$
```

```
00:00:00.031462 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.032635 IP 10.92.213.241.54141 > 10.92.213.231.57
0
00:00:00.034243 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.036050 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.036384 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.036678 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.037042 IP 10.92.213.231.5757 > 10.92.213.241.541
952
00:00:00.037689 IP 10.92.213.241.54141 > 10.92.213.231.57
0
00:00:00.038255 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.038591 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.038871 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.039169 IP 10.92.213.231.5757 > 10.92.213.241.541
1448
00:00:00.039455 IP 10.92.213.231.5757 > 10.92.213.241.541
```



Slow Start

- Note behaviour of slow start
- Sender initially only has small amounts of unack'd data, but progressively speeds up by sending more and more packets "back to back"

What's going on?

- What are those 952 byte packets? 952? Where does that come from?
- 1448 * 5 + 952 = 8192, the blocks I'm calling the API with.
- When in slow start, there's enough delay on proceedings that the blockings matter.
- Once the transfer really gets going, they're aggregated.
- 1448 * 11 + 456 = 8192 * 2: two blocks

Edit daemon…

- Make the block size I'm sending from userland be 64k instead (ie, all of 32k file sent in one write())

```
1448
00:00:00.032257 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.032536 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.032837 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.033189 IP 10.92.213.241.57935 > 10.92.213.231.57
0
00:00:00.033575 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.033799 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.034122 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.034340 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.034686 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.034897 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.035255 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.035473 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
00:00:00.035805 IP 10.92.213.231.5757 > 10.92.213.241.579
1448
```



Programming Tip

- For maximum performance, send the largest possible blocks into the kernel
- With 8192 byte writes, 30ms to send 32768 bytes
- With 65536 byte writes, ie “file at a file”, 23ms
- 23% performance improvement by changing one constant!
- If available, use “sendfile()” system call

1M file, steady state

```
00:00:00.300575 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.300854 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.301345 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.301584 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.301843 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.302088 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.302330 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.302560 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.302797 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.303115 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.303477 IP 10.92.213.241.53681 > 10.92.213.231.5757: tcp
0
00:00:00.303758 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
00:00:00.304058 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp
1448
```

Acks are trailing data

```
00:00:00.624882 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [.], seq 988985:990433, ack 1, win 453, length 1448
00:00:00.625125 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [P.], seq 990433:991881, ack 1, win 453, length 1448
00:00:00.625469 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [.], seq 991881:993329, ack 1, win 453, length 1448
00:00:00.625747 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [.], seq 993329:994777, ack 1, win 453, length 1448
00:00:00.626078 IP 10.92.213.241.34058 > 10.92.213.231.5757:
Flags [.], ack 991881, win 64436, length 0
00:00:00.626310 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [.], seq 994777:996225, ack 1, win 453, length 1448
00:00:00.626575 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [.], seq 996225:997673, ack 1, win 453, length 1448
00:00:00.627065 IP 10.92.213.231.5757 > 10.92.213.241.34058:
Flags [.], seq 997673:999121, ack 1, win 453, length 1448
```

What about broken networks?

technique.

```
# use a very, very rubbish transmission protocol

sub sendfiles {
  my $s = shift;
  while (my $f = shift) {
     my $fh = new IO::File ("<$f") or next;
     my $buffer;
     while (my $bytes = $fh->sysread ($buffer, 8)) {
        $s->syswrite ($buffer, $bytes) or die "cannot write $!";
   sleep (1);
     }
  }
}
```

00:00:**00.020080** IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq 1:9, ack 1, win 453, options [nop,nop,TS val 21020214 ecr 21020236], length 8
00:00:00.021062 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack 9, win 457, options [nop,nop,TS val 21020238 ecr 21020214], length 0
00:00:**01.023658** IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq 9:17, ack 1, win 453, options [nop,nop,TS val 21020314 ecr 21020238], **length 8**
00:00:01.024786 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack 17, win 457, options [nop,nop,TS val 21020338 ecr 21020314], length 0
00:00:**02.027197** IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq 17:25, ack 1, win 453, options [nop,nop,TS val 21020415 ecr 21020338], **length 8**
00:00:02.028213 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack 25, win 457, options [nop,nop,TS val 21020439 ecr 21020415], length 0
00:00:**03.030555** IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val 21020515 ecr 21020439], **length 8**
00:00:03.031564 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack 33, win 457, options [nop,nop,TS val 21020539 ecr 21020515], length 0

We can cause Linux to lose packets

```
igb@pi-one:~$ sudo iptables -I OUTPUT -p tcp -m tcp --sport
5757 -j DROP
```

```
igb@pi-one:~$ sudo iptables --delete OUTPUT 1
```

Note that the dropped packets don't appear in tcpdump output when tcpdump is run on the local machine; if you need that, you need to monitor at the other end.

Flags [P.], seq 17:25, ack 1, win 453, options [nop,nop,TS val
21041889 ecr 21041813], length 8
00:00:02.026411 IP 10.92.213.238.49784 > 10.92.213.231.5757:
Flags [.], ack 25, win 457, options [nop,nop,TS val 21041913 ecr
21041889], length 0
00:00:03.028760 IP 10.92.213.231.5757 > 10.92.213.238.49784:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21041989 ecr 21041913], length 8
00:00:03.029824 IP 10.92.213.238.49784 > 10.92.213.231.5757:
Flags [.], ack 33, win 457, options [nop,nop,TS val 21042013 ecr
21041989], length 0
00:00:04.032088 IP 10.92.213.231.5757 > 10.92.213.238.49784:
Flags [P.], seq 33:41, ack 1, win 453, options [nop,nop,TS val
21042090 ecr 21042013], length 8
00:00:**04.033088** IP 10.92.213.238.49784 > 10.92.213.231.5757: Flags
[.], ack 41, win 457, options [nop,nop,TS val 21042114 ecr 21042090],
length 0
00:00:**22.039894** IP 10.92.213.231.5757 > 10.92.213.238.49784: Flags
[P.], seq 41:185, ack 1, win 453, options [nop,nop,TS val 21043891
ecr 21042114], length **144**
00:00:22.040918 IP 10.92.213.238.49784 > 10.92.213.231.5757:
Flags [.], ack 185, win 473, options [nop,nop,TS val 21043914 ecr
21043891], length 0
00:00:23.040658 IP 10.92.213.231.5757 > 10.92.213.238.49784:
Flags [P.], seq 185:193, ack 1, win 453, options [nop,nop,TS val
21043991 ecr 21043914], length 8

What happened?

Note

- Packets aren't retransmitted, data is retransmitted
- So the dropped packets' data is all re-sent in the minimum number of packets it will fit in

Packets Dropped

```
igb@pi-one:~$ sudo iptables -I OUTPUT -p tcp -m tcp --sport 5757
-j DROP; sleep 30; sudo iptables --list -v; sudo iptables --
delete OUTPUT 1
Chain INPUT (policy ACCEPT 53 packets, 3108 bytes)
 pkts bytes target     prot opt in     out     source
destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source
destination

Chain OUTPUT (policy ACCEPT 54 packets, 10767 bytes)
 pkts bytes target     prot opt in     out     source
destination
   30  4153 DROP       tcp  --  any    any     anywhere
anywhere             tcp spt:5757
igb@pi-one:~$
```

Drop Packets on Other Side

- If we drop packets at the INPUT of the receiver, we can see the retransmissions

```
sudo iptables -I INPUT -p tcp -m tcp --sport 5757 -j DROP; \
   sleep 10; sudo iptables -D INPUT 1
```

21158223], length 0
00:02:**57.215914** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158323 ecr 21158247], length 8
00:02:**57.418894** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158344 ecr 21158247], length 8
00:02:**57.628899** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158365 ecr 21158247], length 8
00:02:**58.048900** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158407 ecr 21158247], length 8
00:02:**58.888921** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158491 ecr 21158247], length 8
00:03:**00.568905** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158659 ecr 21158247], length 8
00:03:**03.938934** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21158996 ecr 21158247], length 8
00:03:**10.678934** IP 10.92.213.231.5757 > 10.92.213.238.49817:
Flags [P.], seq 25:33, ack 1, win 453, options [nop,nop,TS val
21159670 ecr 21158247], length 8
00:03:10.679897 IP 10.92.213.238.49817 > 10.92.213.231.5757:

Why don’t the packets get bigger?

- Implementation decision: could be different on other operating systems (although appears same on Solaris, as it happens)
- Makes sense on slow, lossy network: once you suspect packets are going missing, more sensible to send small packets to find out if it is working than large ones.