6: Transport Layer Overview

Last Modified:
2/17/2003 2:18:41 PM

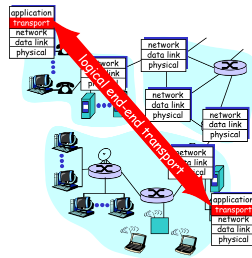
Transport Layer

Overview:

- transport layer services
- multiplexing/demultiplexing
- connectionless transport: UDP
- principles of reliable data transfer
- connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management
  - congestion control
- Instantiation and implementation in the Internet

Transport services and protocols

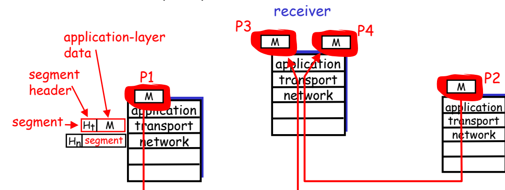
- provide *logical communication* between app' processes running on different hosts
- transport protocols run in end systems
- transport vs network layer services:
- *network layer:* data transfer between **end systems**
- *transport layer:* data transfer between **processes**
  - relies on, enhances, network layer services



Process-to-Process Message Delivery

Goal : Deliver application data to correct process (and more particularly to the right socket)

Segment - unit of data exchanged between transport layer entities; transport protocol data unit (TPDU)



Transport protocol example

- 2 households each with 12 children all cousins.
  - cousins all write letters to each other every week
  - In each house, one child volunteers to collect all the outgoing letters and distribute all the incoming letters
- Analogy to the Internet
  - Hosts = houses
  - Processes = cousins
  - Application messages = letters in envelopes
  - Network layer protocol = postal service
  - Transport layer protocol = volunteers
    - If note any missing letters and rerequest them etc. then like TCP
    - If just hand out whatever comes in then like UDP

UDP: User Datagram Protocol [RFC 768]

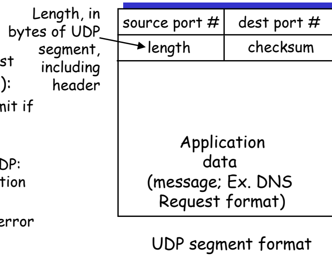
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- TCP is based on a full duplex connection so can't use to send to multiple receivers at once (I.e. broadcast or multicast)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
  - Conducive to multicast
- other UDP uses (why?):
  - DNS: small, retransmit if necessary
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recover!

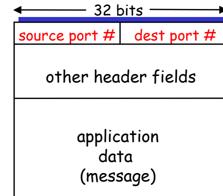


UDP segment format

Multiplexing/demultiplexing

Multiplexing: gathering data from multiple application processes on the same host and sending out the same network interface

Demultiplexing: Stream of incoming data into one machine separated into smaller streams destined for individual processes



TCP/UDP segment format

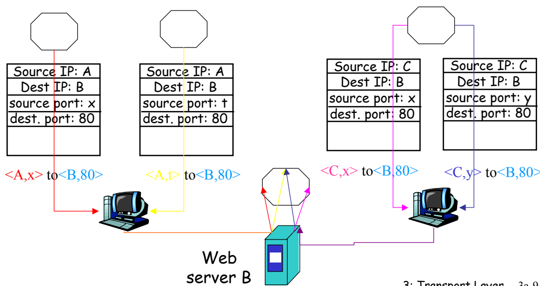
Demultiplexing based on IP addresses and port number for both the sender and receiver

- Can distinguish traffic coming to same port but part of separate conversations (like multiple client connections to a web server)

Multiplexing/demultiplexing example

Two Web browsers on host A each open 1 socket

One Web browser on host C opens 2 sockets

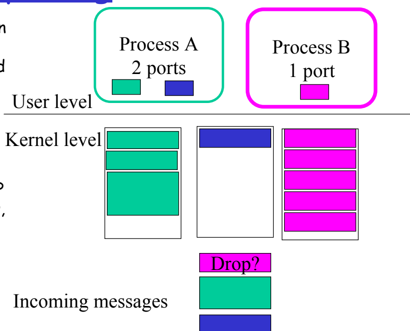


Port Implementation

- Message queue
  - Append incoming message to the end
  - Much like a mailbox file
  - Choose which message queue based on <src ip+ port, dest ip +port>
- If queue full, ,message can be discarded
  - why is that ok? Best effort delivery
  - The network doesn't guarantee not to drop, so the OS needn't guarantee that either
- When application, reads from socket, operating system removes some bytes from the head of the queue
- If queue empty, application blocks waiting

Demultiplexing

- Packets arrive on network interface, copied up into system memory
- Placed in message queue by transport protocol, dest IP and port number, src IP and port number
- Copied to user level when app reads socket



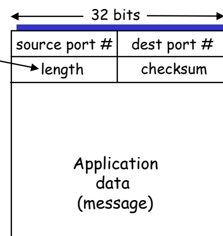
Demultiplexing (cont)

- Receiving process may specify combinations of <srcaddr, srcport, destaddr, destport> it will receive or ANY
- Demultiplexing by port numbers and IP address: other choices?
  - Ip address and process id? high overhead of coordination and couldn't have multiple streams per process
  - Additional level of addressing by port number provides level of indirection and finer granularity addressing

UDP Headers

- Entire UDP header is 8 bytes
- Source and destination ports for demultiplexing
- Port field is 16 bits; so 2^{16} or 64K possible ports -not enough for whole Internet, why ok? Just for the single host!
- Length is 16 bits
- Checksum is 16 bits

Length, in bytes of UDP segment, including header



UDP segment format

UDP header field: checksum

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

Sender:

- treat segment contents as sequence of 16-bit integers (add 0 pad to get even 16 bit chunks if necessary)
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field
- Checksum optional but should always be used

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonethless?* More later ....
  - Errors could be anywhere – in data, in headers, even in checksum

UDP checksum

- Checksum over UDP header and the "pseudo header" – not just the data
- 12 byte Pseudo header precedes UDP header
  - duplicates source and destination IP addresses and the 8 bit protocol ID from IP header
  - also duplicates 16 bit UDP length from UDP header
- Why?Double-check message correctly delivered between endpoints.
  - Ex.Detect if IP address modified in transit

UDP checksum

- Actually optional
  - If sender does not compute set checksum field to 0
  - If calculated checksum is 0? Store it as all one bits (65535) which is equivalent in ones-complement arthimetic
- If checksum is non-zero and receiver computes a different value, silently drop packet; no error msg generated
- Note: We will talk about more about error detection and correction at the link layer....

UDP Header: length

- Length of data and header (min value 8 bytes = 0 bytes data)
- 16 bit length field => max length of 65535 bytes
- Can you really send that much?
  - May be limited by kernel send buffer (often <= 8192 bytes)
  - May be limited by kernel's IP implementation (possibly <= 512 bytes) ; Hosts required to receive 576 bytes of UDP data so senders may limit themselves to that as well

Experimenting with UDP

- Programs like sock, ttcp or pcattcp allow you to generate streams of TCP or UDP data according to your specifications (total amount of data to send, size of each segment sent, etc.)
- Normally, procedure is as follows
  - Start tracer like Ethereal
    - Consider a filter like (ip.addr eq senderIPaddress)
  - Start server process (ex. pcattcp –r –u)
  - Start client process sending traffic (ex. pcattcp –t –u <ip address of server)
    - Note: loopback or own IP address may not appear in Ethereal
  - Experiment with different size sends –l bytes (default is 8192) or number of buffers to send –n sends (default is 2048)
    - On Ethernet 1473 causes fragmentation, 1472 does not

UDP Performance Experiments

- Vary buffer size , keep total data size the same
- Should see higher overall throughput when sending in larger units Why? Many overheads are fixed
  - Packet headers
  - Kernel processing
  - Grabbing channel at physical layer
- Also interesting to repeat experiment across different network conditions (on same hub, in same AS, across ASes)
  - Throughout?
  - Data loss

TCP vs UDP on a LAN

- Compare overall throughput for TCP vs UDP
- Expect much lower throughput for TCP – Why?
  - Connection establishment
  - Slowstart
  - Header overhead
- On a LAN, TCP shouldn't see many retransmissions

TCP vs UDP on a WAN

- Should see retransmissions and thus more slow start/congestion avoidance overhead
- Quantify the effect

Roadmap

- UDP is a very thin layer over IP
  - multiplexing /demultiplexing
  - error detection
- TCP does these things also and then adds some other significant features
- TCP is quite a bit more complicated and subtle
- We are not going to jump right into TCP
- Start gently thinking about principles of reliable message transfer in general

The Problem

- Problem: send big message (broken into pieces) over unreliable channel such that it arrives on other side in its entirety and in the right order
- No out of band communication! All communication sent along with the pieces of the message
- Receiver allowed to send information back but only over the same unreliable channel!

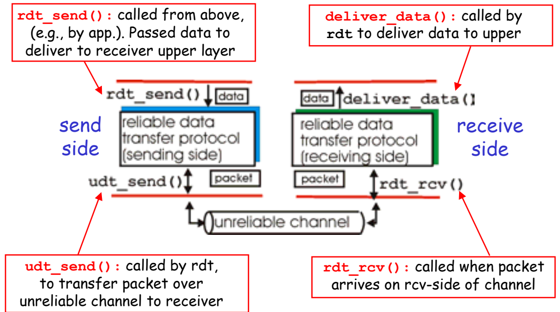
Intuition: Faxing a document With Flaky Machine

- Can't talk to person on the other side any other way
- Number the pages – so sender can put back together
- Let receiver send you a fax back saying what pages they have and what they still need (include your fax number on the document!)
- What if the receiver sends their responses with a flaky fax machine too?
- What if it is a really big document? No point in overwhelming the receiver. Receiver might like to be able to tell you send first 10 pages then 10 more…
- How does receiver know when they have it all? Special last page? Cover sheet that said how many to expect?

Principles of Reliable data transfer

- Solving this problem is one on top-10 list of most important networking topics!
  - important in application, transport, link layers
- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol– what is worst underlying channel can do?
  - Drop packets/pages?
  - Corrupt packet/pages (even special ones like the cover sheet or the receiver's answer?)
  - Reorder packets/pages?

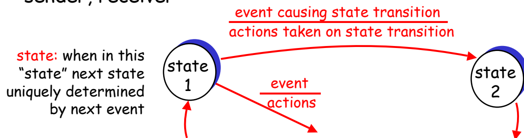
Reliable data transfer: getting started



Reliable data transfer: getting started

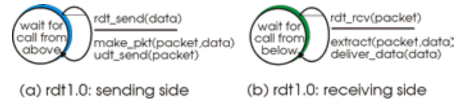
We'll:

- 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 (so this should be easy ☺)
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



(a) rdt1.0: sending side (b) rdt1.0: receiving side

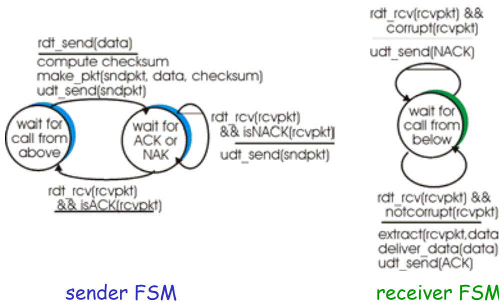
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet (can't drop or reorder packets)
  - recall: UDP checksum to detect bit errors
- *Once can have problems, the receiver must give the sender feedback (either that or the sender would just have to keep sending copy after copy forever to be sure)*
- *After receiving a packet, the receiver could say one of two things:*
  - *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?

Rdt2.0: channel with bit errors

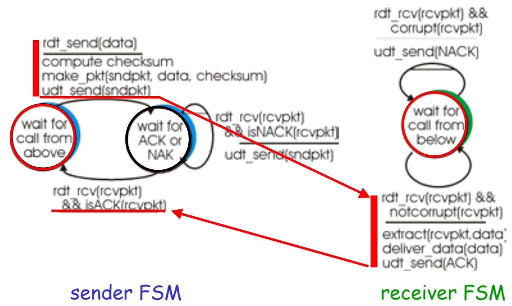
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender (let receiver fax you back info?)
  - Possible retransmission – detection of duplicates (number fax pages?)
  - error detection (checksums? Cover sheet summary?)

rdt2.0: FSM specification



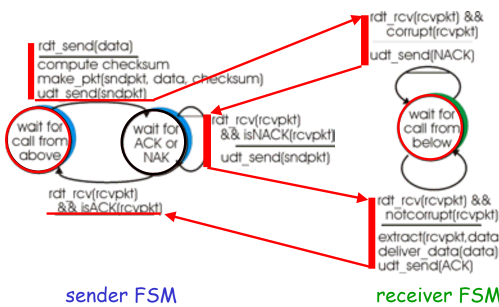
sender FSM receiver FSM

rdt2.0: in action (no errors)



sender FSM receiver FSM

rdt2.0: in action (error scenario)



sender FSM receiver FSM

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- FSM implied could tell if it was and ACK or a NACK
- What if is a FLACK?

What to do?

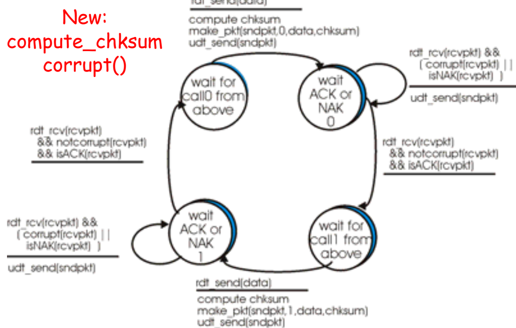
- Assume it was an ACK and transmit next? What if it was a NACK? Missing data
- Assume it was a NACK and retransmit; What if it was an ACK? Duplicate data

Handling duplicates:

- To detect duplicate, sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- If receiver has pkt with that number already it will discards (I.e. not deliver up duplicate pkt)

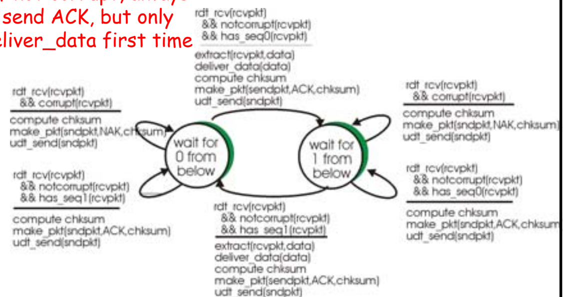
rdt2.1: sender, handles garbled ACK/NAKs

New:
compute_chksum
corrupt()



rdt2.1: receiver, handles garbled ACK/NAKs

If not corrupt, always send ACK, but only Deliver_data first time



rdt2.1: discussion

Sender:

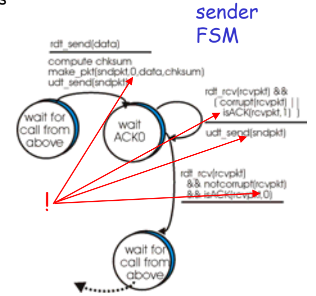
- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
  - Note: This protocol can also handle if the channel can duplicate packets
- note: when can sender and receiver safely exit? receiver can *not* know if its last ACK/NAK received OK at sender
  - Missing connection termination procedure

rdt2.2: a NAK-free protocol

- Less intuitive but getting us closer to TCP
- same functionality as rdt2.1, using NAKs only
- instead of NAK, receiver sends ACK for last pkt received OK (or for other number on the first receive)
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate (or unexpected) ACK at sender results in same action as NAK: *retransmit current pkt*
- TCP really ACKS the next thing it wants



rdt3.0: channels with errors (and duplicates) *and* loss

New assumption: underlying channel can also lose packets (data or ACKs)

- How to deal with loss? Retransmission plus seq # to detect duplicates
- but not enough

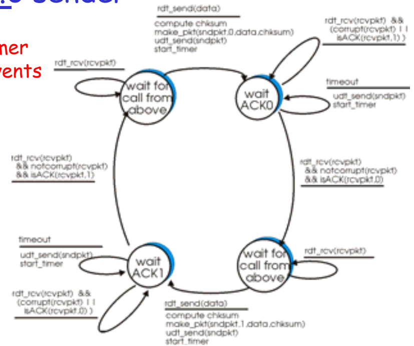
Q: how to detect loss?

Approach: sender waits "reasonable" amount of time for ACK

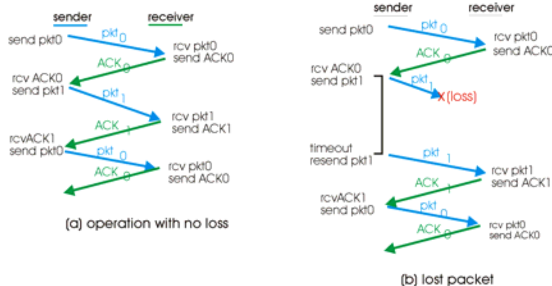
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 sender

Start_timer
Timeout events



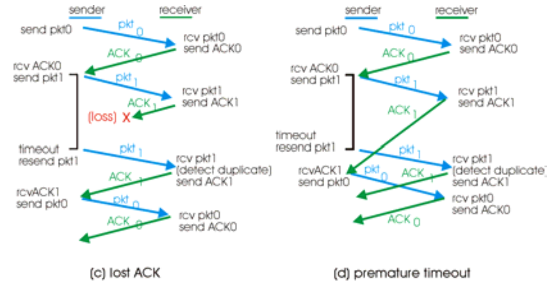
rdt3.0 in action



(a) operation with no loss

(b) lost packet

rdt3.0 in action



(c) lost ACK

(d) premature timeout

Stop and Wait

- Rdt3.0 also called Stop and Wait
  - Sender sends one packet, then waits for receiver response
- What is wrong with stop and wait?
  - Slow!! Must wait full round trip time between each send\
- Obvious Fix?
  - Instead send lots, then stop and wait
  - Call this a pipelined protocol because many packets in the pipeline at the same time

Pipelined protocols

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

- range of sequence numbers must be increased to be able to distinguish them all
- Additional buffering at sender and/or receiver
- Once allow multiple "in-flight" consider that channel may reorder the packets

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

How bad is Stop and Wait?

- Depends on network conditions
- example: 1 Gbps link, 15 ms end-to-end prop. delay, 1KB packet:

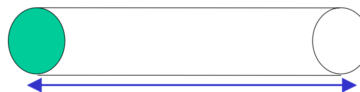
$$T_{transmit} = \frac{1kb/pkt}{10^{**}9 \text{ b/sec}} = 1 \text{ microsec}$$

$$\text{Utilization} = U = \text{Utilization of the channel} = \frac{1 \text{ microsec}}{30.001 \text{ msec}} = 0.003\%$$

  - 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
  - network protocol limits use of physical resources!
- In general, smaller packets, longer RTT and higher maximum bandwidth, all make the situation worse

Filling the pipeline

- How much in-flight data is needed to "fill the pipeline"?
- Similar to question of how much water needed to fill a pipe (area of crosssection * length of pipe)



- For networks, it is bandwidth*delay

Pipelined protocols

- Two generic forms of pipelined protocols
  - *Go-Back-N*
  - *Selective repeat*
- Many possible variations on each

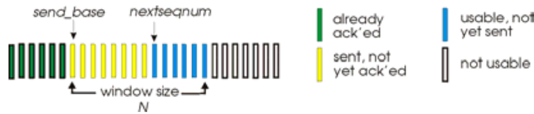
Go-Back-N

- Sender keeps track of beginning of a window of up to N packets
- Each time get an ACK for the beginning of the window can advance the window
- If get a timeout for the first packet in the window, retransmit all packets in the window
- Some of those retransmitted packets may have been correctly received

Go-Back-N

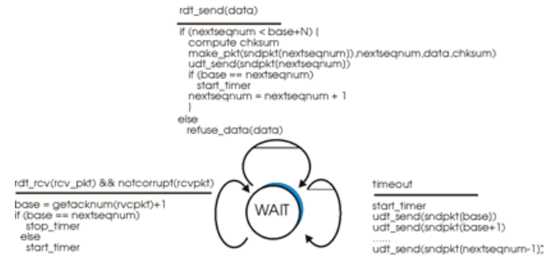
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed (want N large enough to fill the pipeline, based on link characteristics)



- *Cumulative ACK:* ACK(n): ACKs all pkts up to, including seq # n
  - may receive 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



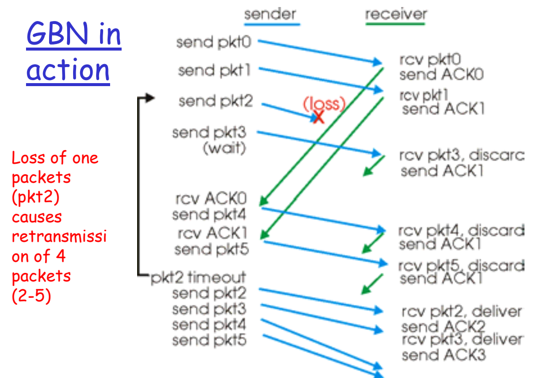
GBN: receiver extended FSM

receiver simple:

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

GBN in action

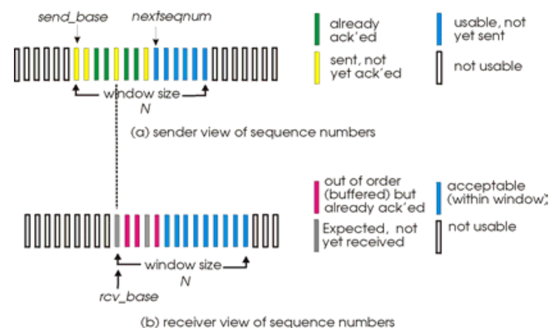
Loss of one packets (pkt2) causes retransmission of 4 packets (2-5)



Selective Repeat

- GBN forces sender to retransmit all packets in window even if some have been correctly received
- To avoid that we need a finer granularity of acknowledgement
  - individual acknowledgements vs cumulative acknowledgements

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat

sender

data from above :
- if next available seq # in window, send pkt

timeout(n):
- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:
- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

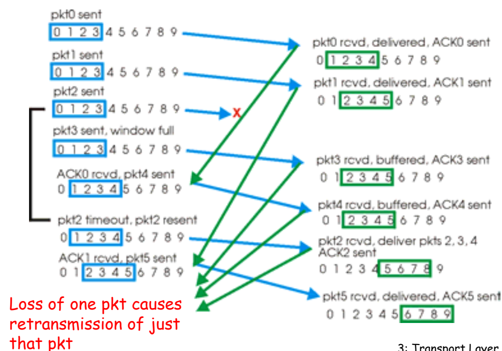
pkt n in [rcvbase-N,rcvbase-1]
- Duplicate
- ACK(n)

otherwise:
- ignore

Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - Must buffer any packet acknowledged for eventual in-order delivery to upper layer (even if cannot deliver right now)
  - Can still choose not to ACK an out of order packet if insufficient buffer space
- 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 in action



Loss of one pkt causes retransmission of just that pkt

Selective Repeat vs GBN

- Selective Repeat requires individual acknowledgements rather than chance for cumulative acknowledgements
- GBN results in unnecessary retransmission of data correctly received
- In Selective Repeat, sender can choose to buffer out of order and avoid unnecessary retransmission (but not required)

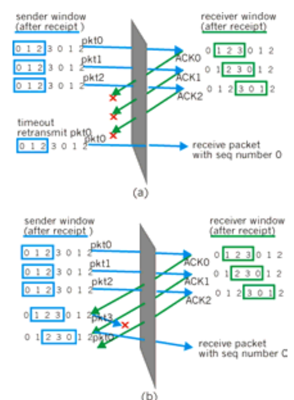
TCP?

- TCP is most like GBN
  - But many TCP implementations will buffer correctly received but out of order segments and senders use duplicate acknowledgments to infer which segment dropped .. This is sort of like Selective Repeat
- TCP uses cumulative acknowledgements but counts bytes not packets and receiver ACKS what it wants not last thing it received
- Window size is not fixed like N in GBN
  - TCP allows receiver to set a maximum (dynamically)
  - Effective window size also changed over time in response to signs of congestion in the network

Pipelined protocols

Sequence Number 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)



Sequence Numbers

- Q: what relationship between seq # size and window size?
- A: window size <= ½ sequence number space
  - True for Stop and Wait (1 <= ½*2)
  - need old and new version of every sequence #
- Still one problem, packets could conceivably delayed for arbitrarily long in the network so could get an old packet N even after the sequence number space has wrapped around
- Solution? Not really. In practice, assume a maximum time a packet could live in the network

Roadmap

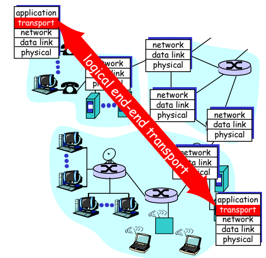
- Discussed general principles of reliable message delivery over unreliable channel
  - Lots of it is common sense (like with our flaky fax machine)
  - But there is a significant degree of subtlety in getting it right!
- We are going to move on to talking specifically about TCP
  - Flow control? Congestion control?
- We have most of the tools we need now: sequence numbers, cummulative acknowledgments, retransmisson timers….

Outtakes

Transport-layer protocols

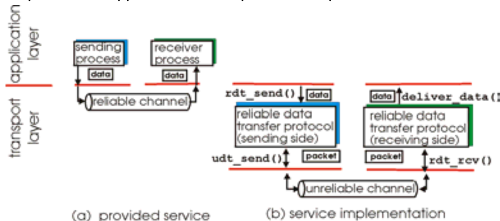
Internet transport services:

- reliable, in-order unicast delivery (TCP)
  - Connection oriented
  - flow control
  - congestion control
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - Interarrival time gurantee
  - bandwidth guarantee
  - If IP layer can't provide no way to simulate on top



Principles of Reliable data transfer

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



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt) - what is worst underlying channel can do?