Lecture 5: UDP/TCP Protocols

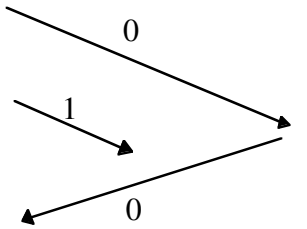
5.0 Main Points

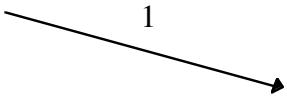
RTT estimation
UDP
TCP

5.1 RTT estimation

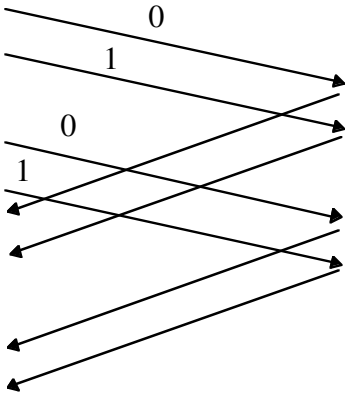
How do we determine timeout?

What if the timeout is too big? Low utilization



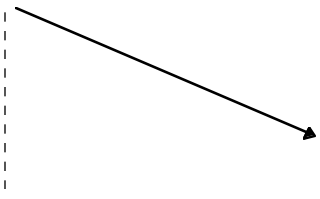


What if timeout is too small? Unnecessary retransmits and redundant acks.

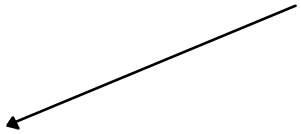


Want transport code to handle any kind of network, from local area Ethernet to wide area Internet. Round trip times may vary by several orders of magnitude.

Solution? Set timeout based on estimate of round trip time, measured by looking at how long acks take to return.



RTT



What if RTT changes? (due to rerouting or congestion)

Average samples together over time.

a. Average all samples

RTT = (1/N) * Sum X(i)

b. Windowed average : consider last N samples

c. Exponentially weighted moving average

RTT(k) = Sum X(i) * a^(k-i)

RTT(k) ~ a * X(k) + (1-a) * RTT(k-1)

What if RTT measurements are noisy?

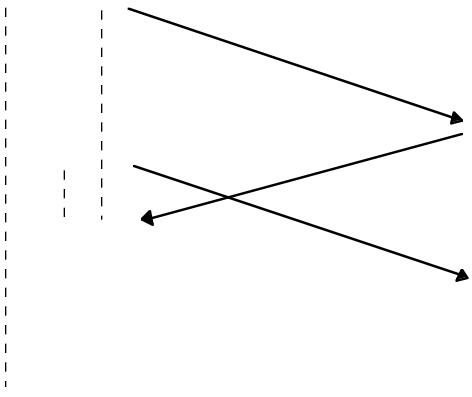
Timeout = estimated RTT?

Timeout = 2 * est RTT?

Timeout = est RTT + k * est variance in RTT

Example of self-tuning system: adapts to changes in underlying hardware.

Retransmission ambiguity: can't distinguish first ack from retransmitted ack



a. Karn-Partridge: ignore RTT estimates for retransmitted packets

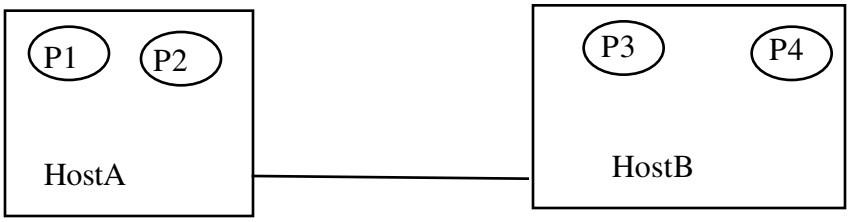
and double timer on every retransmission (until get a successful estimate)

b. add sequence numbers to retransmissions (e.g., retry #1, retry #2)

c. add timestamp to every send; ack returns timestamp

4.3 UDP -- User Datagram Protocol

Process to process communication on top of IP.



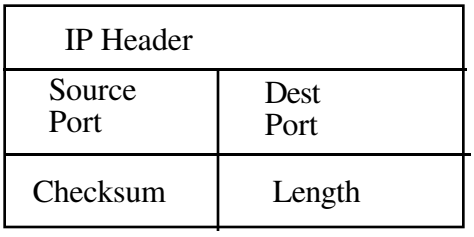
So far, assume host to host communication.

How do processes communcate, e.g., P1 talk to P3, and P2 to P4?
http client (netscape) process talks to http server (httpd) *process*

UDP is one approach:

unreliable message delivery
send to "port" (mailbox) on destination machine
no retransmissions
no sequencing
=> no persistent state

Assumes higher level protocol (e.g., RPC) provides retransmissions, etc.



IP header includes IP address of source, destination machine
Checksum is error detecting code to prevent corrupted packets from being delivered to the application

Target process reads from destination port, replies to source port.

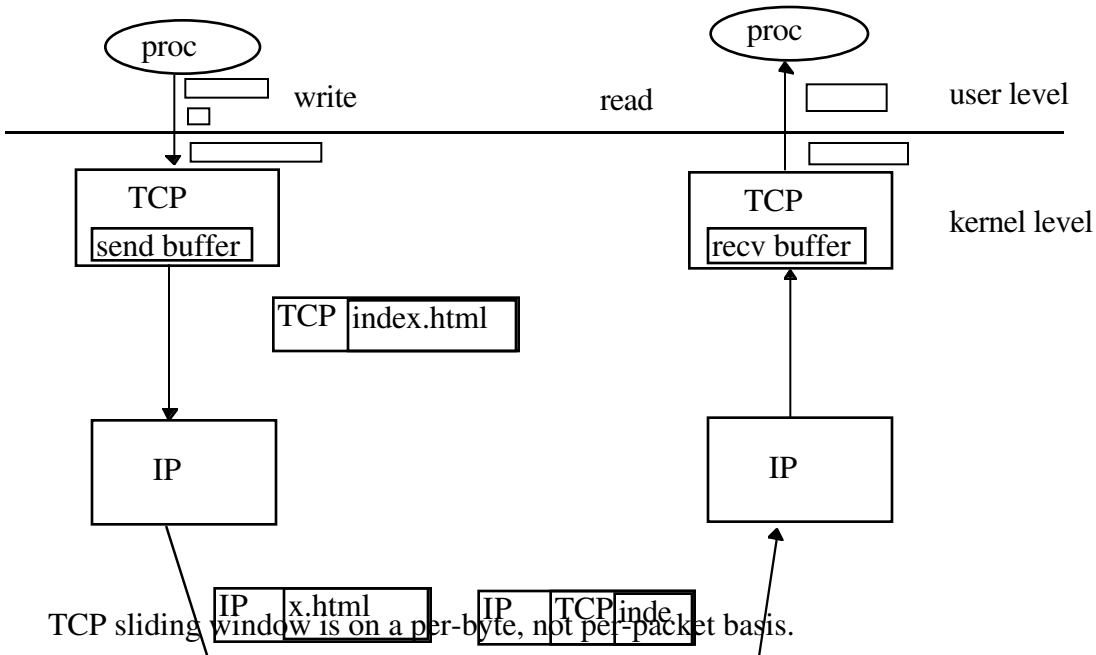
5.2 TCP (Transmission Control Protocol)

Reliable byte stream
full duplex
process to process (with flow control)
highly tuned congestion control algorithm

Uses sliding window, go back N, RTT estimation

Connection setup -- buffer sizes and initial sequence #s.

Maps byte stream onto "segments", which are mapped onto IP packets



TCP sliding window is on a per-byte, not per-packet basis.

Sender keeps track of: last byte acked, last byte sent, last byte written, buffers data LBA+1 -> LBW (max send window)

Receiver keeps track of: last byte read, next byte expected, last byte received, buffers data LBRead + 1 -> LBRecv (max receive window)

Invariants: LBA <= LBS
LBS <= LBW

LBRead < NBE
NBE < LBRecv

Sender blocks if tries to write LBA + max send buffer.

Receiver blocks if tries to read NBE

What if sender process (and network) is faster than receiver process?
Data builds up in receive window
Need to make sure pace of writes matches pace of reads!
TCP solution: in each ack, receiver gives how much space is left in the window: max receive buffer - (NBE - LBRead).

Tells sender when tostop sending.
What if receive window = 0?
How does sender know when receiver process has read bytes out?

Sender periodically pings one byte packet. Receiver acks with current window size.

What if sender transmits as soon as any space in the window opens?

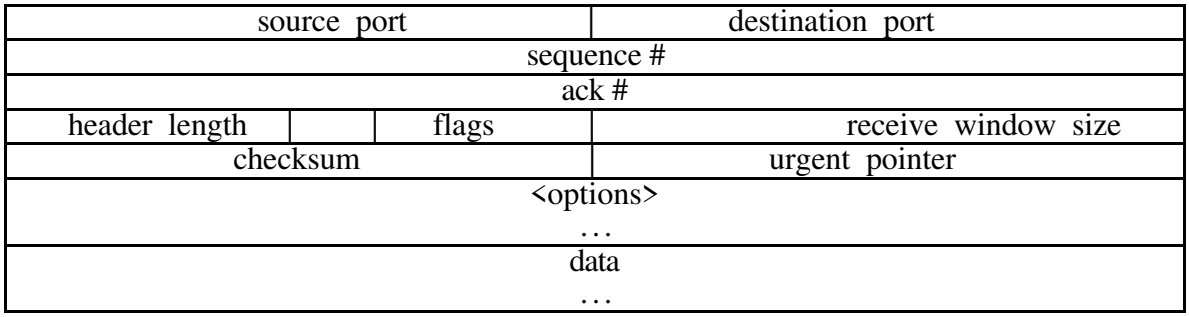
Silly window syndrome: one byte opens, so sender sends one byte, window closes; sender pings, another byte opens, so sender sends another byte, etc.
Sender doesn't start resending until window is half open.

If window is open, when should sender launch a segment?

a. when send buffer > max segment size
b. when app says topush (e.g., telnet)
c. on timeout (ex. 1/2 second)

Nagle: never send a partial segment if one is outstanding.

5.3 TCP packet header format



source port, destination port => as in UDP
Identifies "conversation" endpoints within hosts
<IP source, TCP source port, IP dest, TCP dest port> uniquely identifies connection

sequence # => relative byte offset of segment
    At connection setup, choose initial sequence #'s

ack # => next sequence number expected (NBE), on reverse channel

header length => in 32 bit words, identifies how many options there are

flags => bit vector of control fields (can have multiple set)
    SYN - connection startup
    FIN - connection finish
    RST - connection abort
    PSH - flush data
    ACK - ack # is valid
    URG - urgent pointer is valid

receive window size => can be changed dynamically, for flow control

checksum => detect bit errors (will cover later)

urgent pointer => offset past urgent data within current segment
    For "out of band" data (e.g., ctl-C)