Andrew Donegan

CS3031 – Advanced Telecommunications

Hitesh Tewari

Room 131 Lloyd Institute
Tel: 896 2896

Email: htewari@cs.tcd.ie
Web: http://www.cs.tcd.ie/~htewari

Course Details

- Duration - 12 weeks
- Lectures
  - 3 hrs a week (Mon, Tue & Thur 10-11)
- Assessment
  - Coursework 20%
  - Summer Exam 80%
- Coursework
  - Two project exercises which count for 20% of the marks

Course Outline

- Transport Layer
  - Multiplexing and Demultiplexing
  - Connectionless Transport: UDP
  - Connection-oriented Transport: TCP
  - TCP Congestion Control
- Application Layer
  - Network Application Architectures
  - WWW & HTTP
  - Simple Mail Transfer Protocol (SMTP)
  - Domain Name System (DNS)
  - P2P Applications
  - Web Applications

Course Outline II

- Network Security
  - Symmetric-Key Cryptography
  - Asymmetric-Key Cryptography
  - Digital Signatures, X.509 Certs & PKI
  - Authentication Protocols
  - Secure Socket Layer (SSL)
  - IPsec
  - DNSSEC
- Electronic Payment Systems
  - Ecash
  - Bitcoin
  - Micropayments

Reference Texts

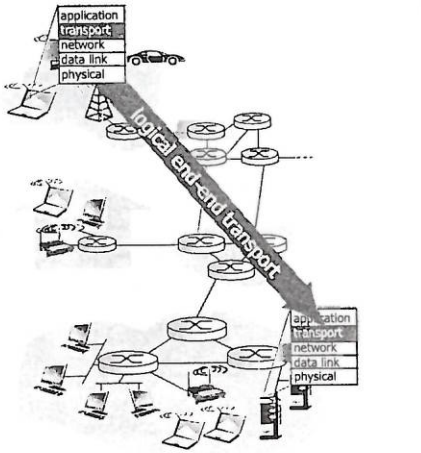
- Computer Networking - A Top-Down Approach, 7th Ed., James F. Kurose, Pearson Intl.
- Understanding Cryptography, Christoff Paar, Springer-Verlag

Transport Layer

- Transport Layer Services
- Multiplexing and Demultiplexing
- Connectionless Transport: UDP
- Connection-Oriented Transport: TCP
- TCP Congestion Control

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

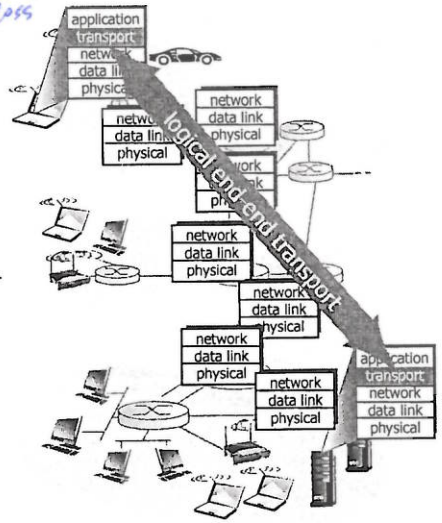


UDP/TCP
DNS/VoIP
Email

Internet Transport-Layer Protocols

- TCP - Reliable, in-order delivery
  - Congestion control
  - Flow control
  - connection setup
- UDP - Unreliable, unordered delivery
  - No-frills extension of "best-effort" IP
- Services not available
  - Delay guarantees
  - Bandwidth

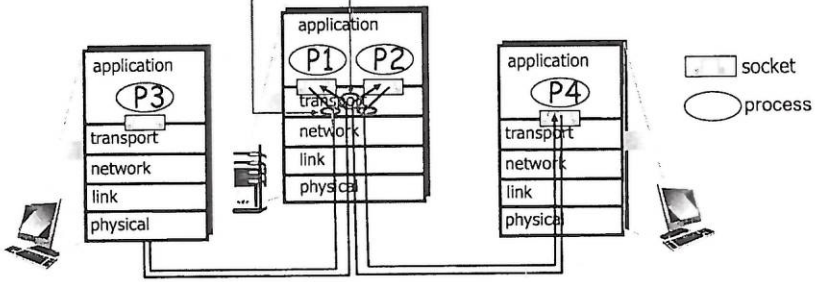
Q: What type of services can we run over TCP & UDP?



Multiplexing/Demultiplexing

multiplexing at sender: handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver: use header info to deliver received segments to correct socket

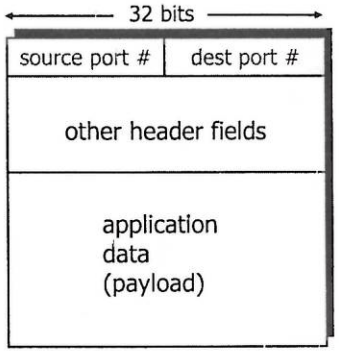


Q: How many addresses does a typical host have? 3 - MAC, IP, Port

How Demultiplexing Works

- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries one transport-layer segment
  - Each segment has source, destination port number

• Host uses IP address & port nums to direct segment to appropriate socket.

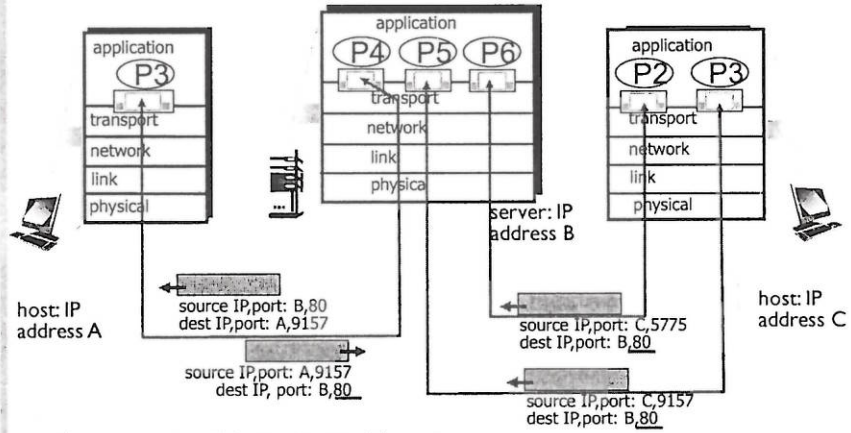


TCP/UDP segment format

Connection-Oriented Demux

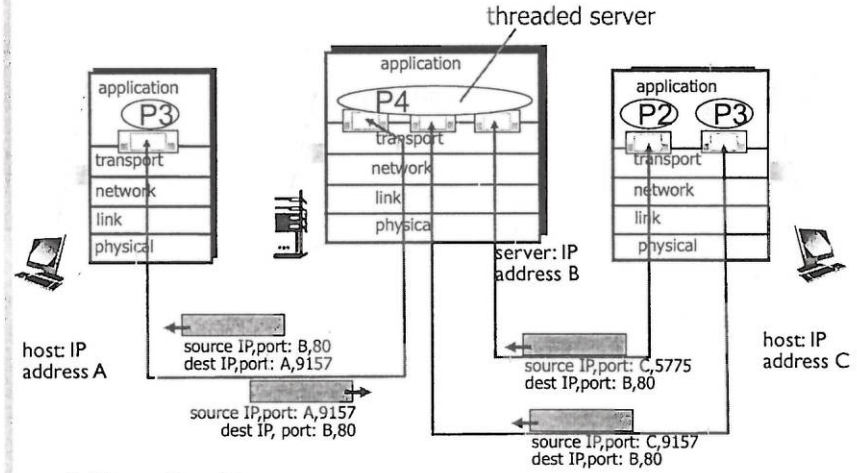
- TCP socket identified by 4-tuple
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Receiver uses all four values to direct segment to appropriate socket
  - Demultiplexing
- Server host may support many simultaneous TCP sockets
  - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have a different socket for each request

Connection-Oriented Demux Example



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

Connection-Oriented Demux Example II



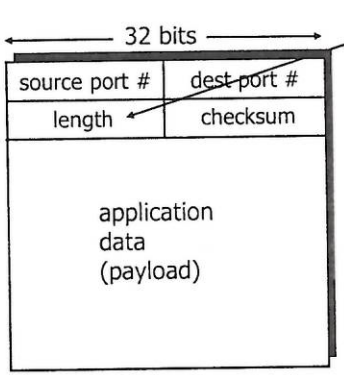
Q: Why use threads?

Lightweight sub-process

User Datagram Protocol (UDP)

- "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
- Uses
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP

UDP Header



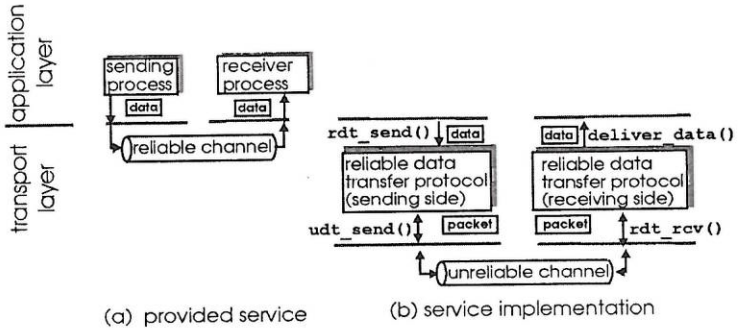
UDP segment format

Why is there a UDP?

- No connection establishment
  - which add delay
  - can
- Simple - No connection state at sender, receiver
- Small header size
- No congestion control
  - UDP can blast away as fast as desired

Principles of Reliable Data Transfer

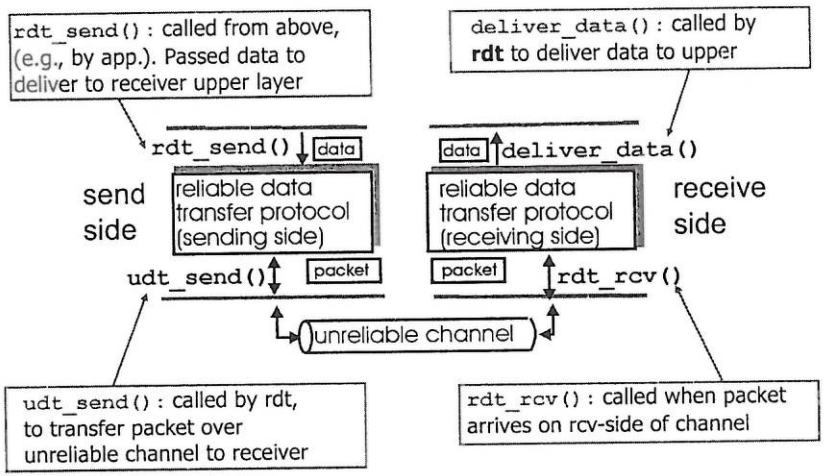
- Important in application, transport, link layers



(a) provided service (b) service implementation

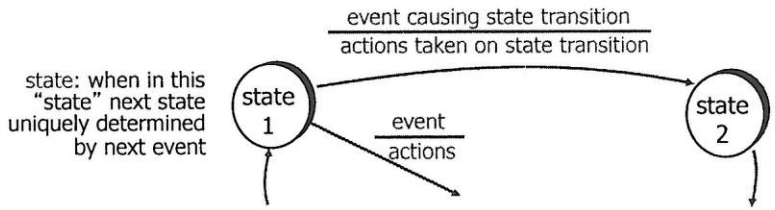
- Characteristics of unreliable channel will determine complexity of *reliable data transfer* protocol

Reliable Data Transfer (RDT)



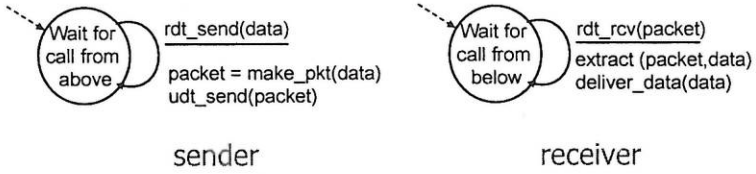
RDT

- We will incrementally develop sender, receiver sides of reliable data transfer protocol
- Consider only unidirectional data transfer
  - But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver



rdt1.0: Reliable Transfer Over a Reliable Channel

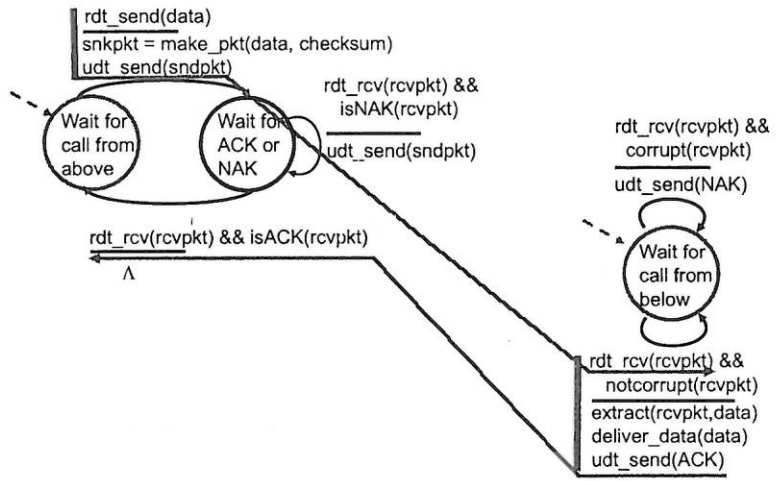
- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender and receiver
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



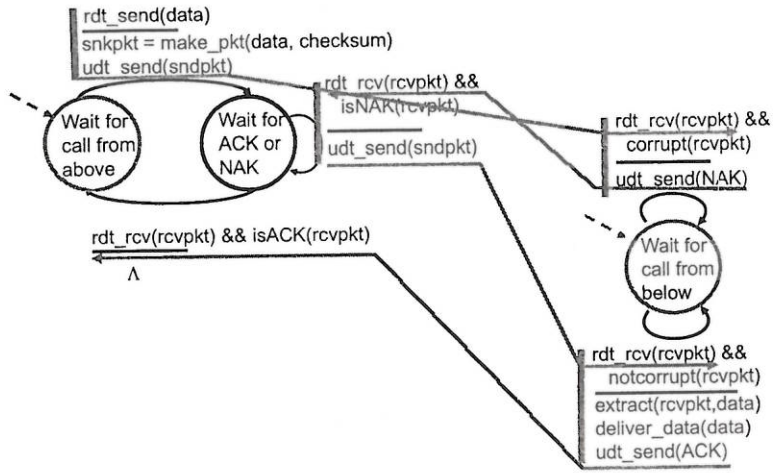
rdt2.0: Channel with Bit Errors

- Underlying channel may flip bits in packet
  - checksum to detect bit errors
- Q: 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
- New mechanisms in rdt2.0 (beyond rdt1.0)
  - Error Detection
  - Feedback
    - Control msgs (ACK, NAK) from receiver to sender

rdt2.0: Operation with No Errors



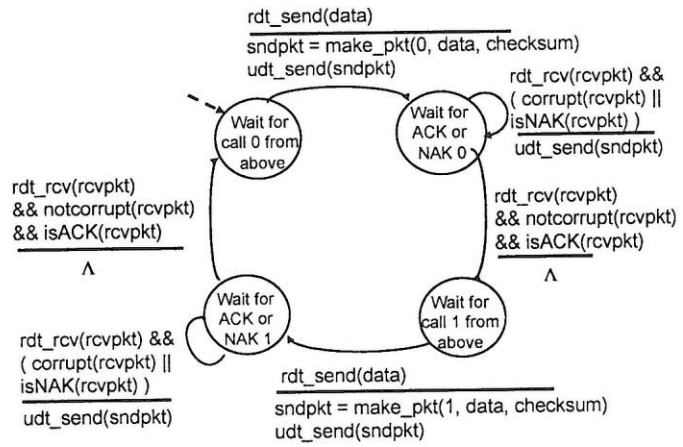
rdt2.0: Error Scenario



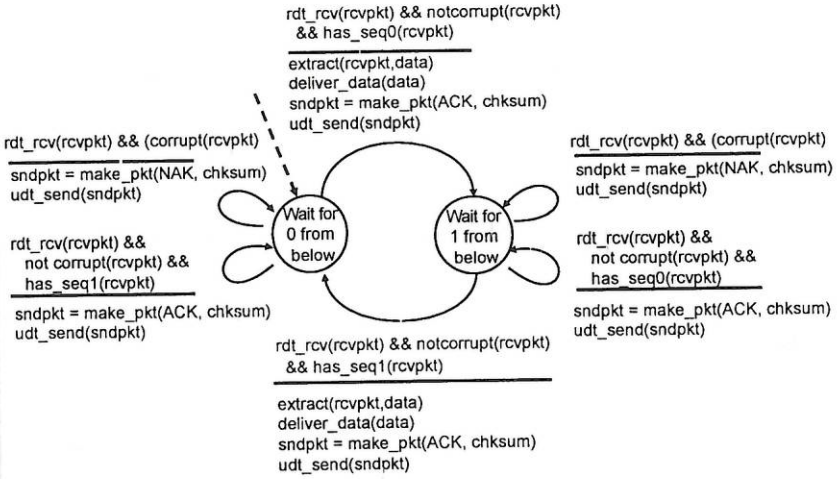
rdt2.0: Fatal Flaw

- What happens if ACK/NAK corrupted?
  - Sender does not know what happened at receiver!
  - Cannot just retransmit
- Handling Duplicates
  - Sender retransmits current pkt if ACK/NAK corrupted
  - Receiver discards (does not deliver up) duplicate pkt
- Stop and Wait
  - Sender sends one packet, then waits for receiver response

rdt2.1: Sender Handles Garbled ACK/NAKs



rdt2.1: Receiver Handles Garbled ACK/NAKs



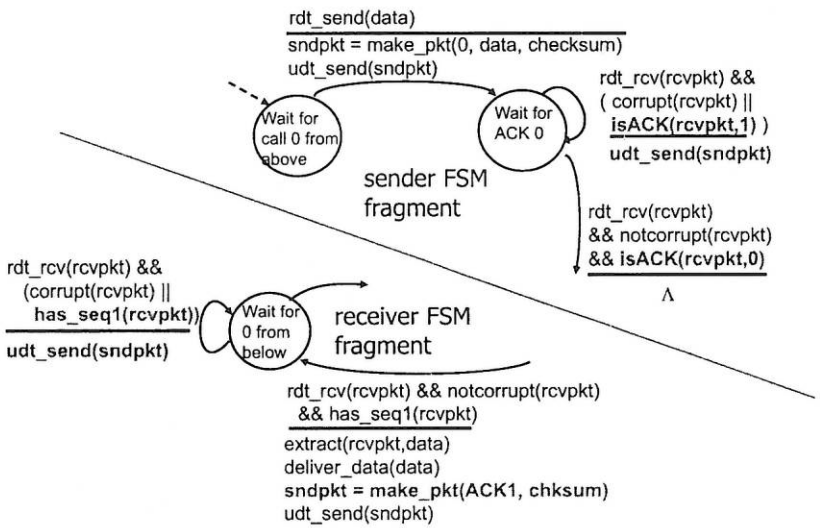
rdt2.1: Discussion

- 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 "expected" pkt should have seq # of 0 or 1
- Receiver
  - Must check if received packet is duplicate
    - State indicates whether 0 or 1 is expected pkt seq #
  - Note: receiver cannot tell if its last ACK/NAK received OK at sender

rdt2.2: NAK-free Protocol

- Same functionality as rdt2.1
  - Using ACKs 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

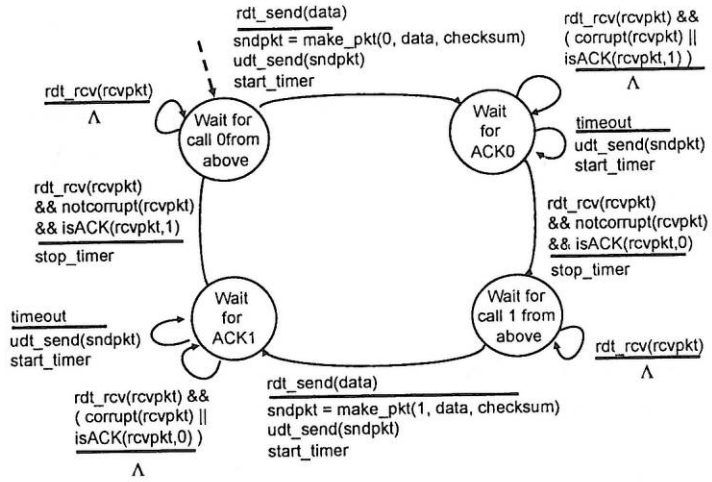
rdt2.2: Sender & Receiver FSM



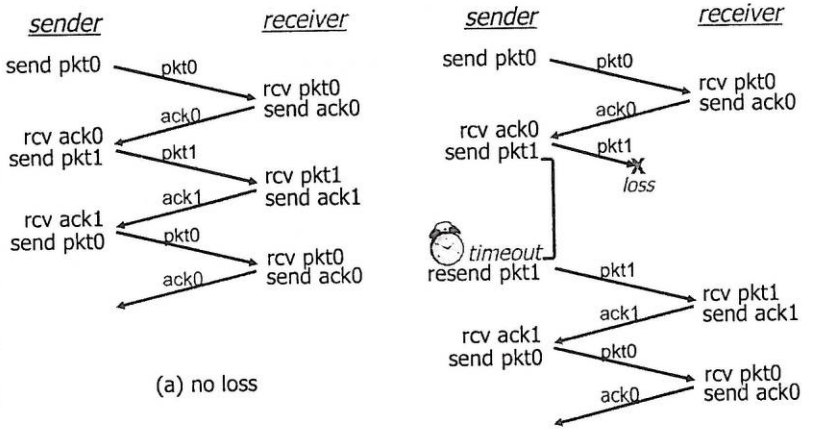
rdt3.0: Channels with Errors and Loss

- New Assumption
  - Underlying channel can also lose packets (data, ACKs)
  - Checksum, seq. #, ACKs, retransmissions will be of help
    - But not enough
- Approach
  - Sender waits "reasonable" amount of time
  - Retransmits if no ACK received in this time
  - If pkt (or ACK) just delayed (not lost)
    - Retransmission will be duplicate, but seq. #'s already handle this
    - Receiver must specify seq # of pkt being ACKed
  - Requires countdown timer

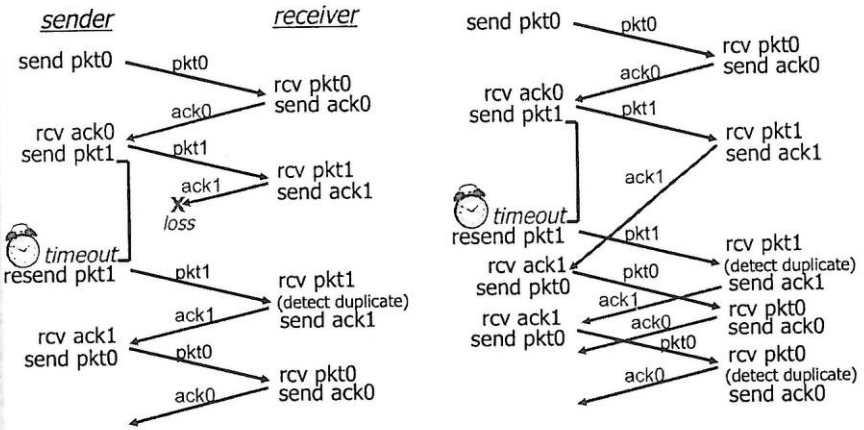
rdt3.0: Sender



rdt3.0: In Action



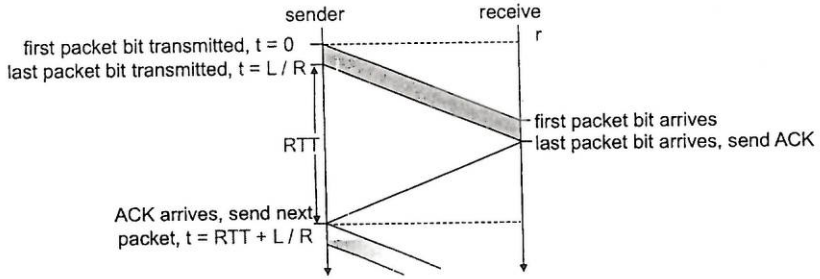
rdt3.0: In Action II



rdt3.0: Stop-and-Wait Operation

- rdt3.0 performance stinks
  - e.g. 1 Gbps link (R), 15 ms prop. delay, 8000 bit packet (L)

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$



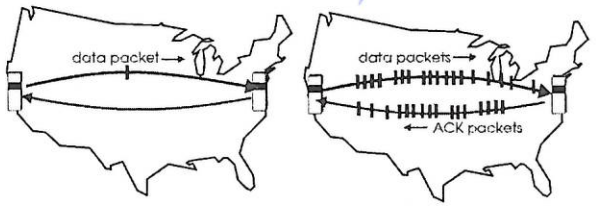
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.006}{30.008} = 0.00027$$

- Effective thruput of 267 kbps over a 1 Gbps link

Pipelined Protocols

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

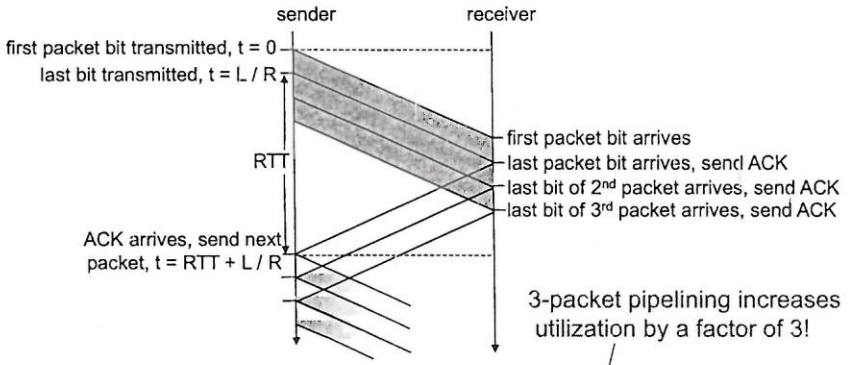
– Range of sequence numbers must be increased – Buffering at sender and/or receiver



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

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

Pipelining: Increased Utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined Protocols: Overview

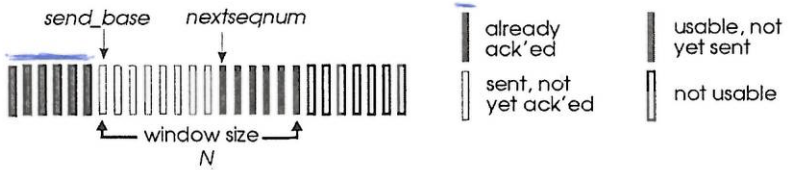
- Go-Back-N (GBN)
  - Sender can have up to *N* unacked packets in pipeline
  - Receiver can send cumulative acks Send ACK of last pkt received
  - Sender has timer for oldest unacked packet
    - When timer expires, retransmit all unacked pkts.
- Selective Repeat (SR)

– Sender can have up to N unacked pkts in pipeline

  - Rcvr sends individual ack for each packet
  - Sender maintains timer for each unacked packet
    - When timer expires, retransmit only that unacked packet

Go-Back-N: Sender

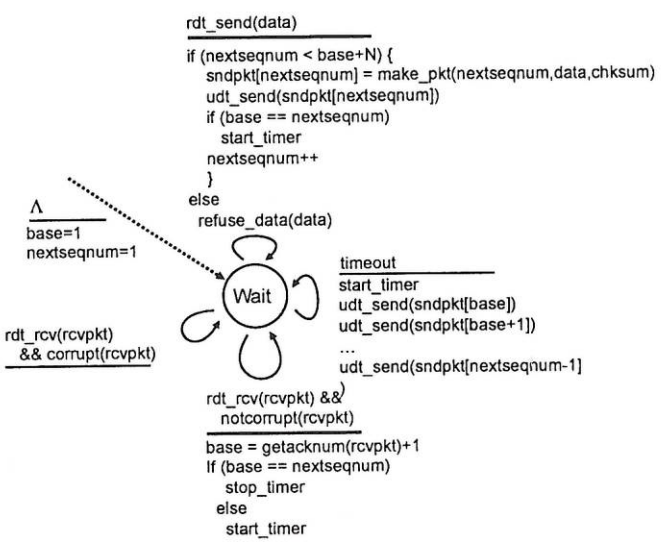
- *k*-bit seq # in pkt header
  - "Window" of up to *N*, consecutive unacked pkts allowed



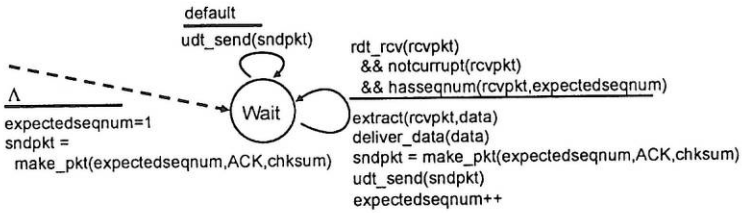
- Timer for oldest in-flight pkt

– Timeout(n): Retransmit pkt n & all higher seq # pkts in window

GBN: Sender Extended FSM

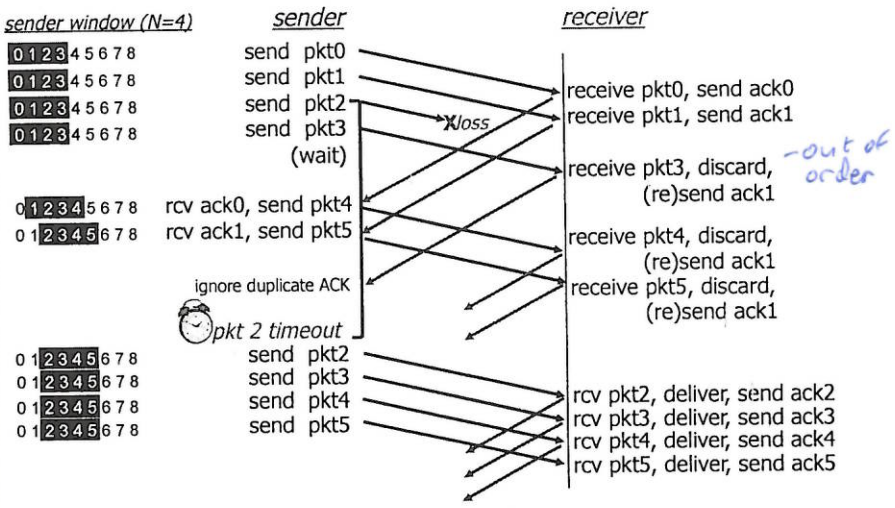


GBN: Receiver Extended FSM



- ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
  - Need only remember **expectedseqnum**
- Out-of-order pkt
  - Discard: no receiver buffering
  - re-ACK pkt with highest in-order seq #

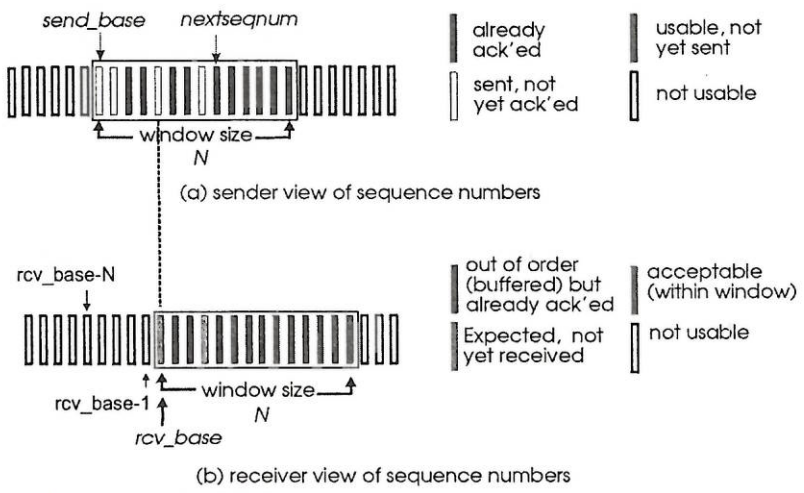
GBN in Action



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
  - Timer for each unACKed pkt
- Sender window
  - *N* consecutive seq #'s
  - Limits seq #*s* of sent, unACKed pkts

SR: Sender & Receiver Windows

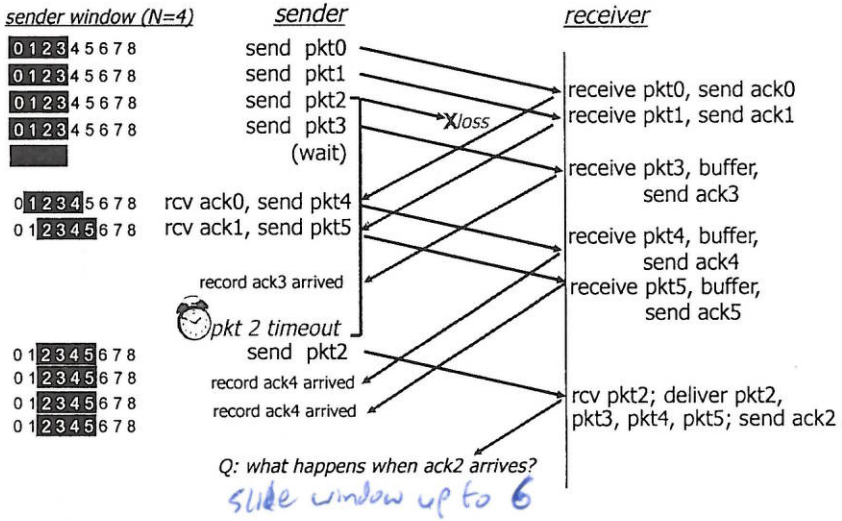


Ack pkts even if already received

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 (seq # == send-base), 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*) - previously acknowledged pkt
  - Otherwise
    - Ignore

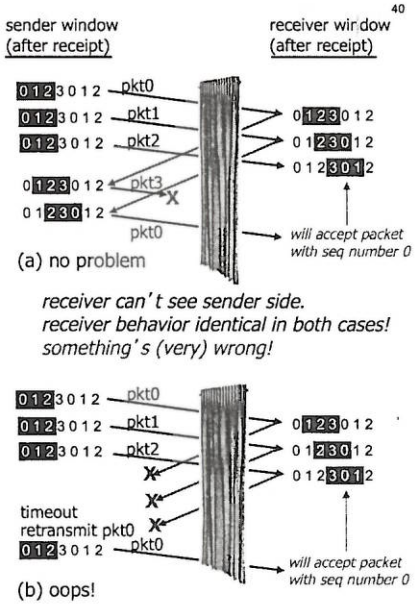
Selective Repeat in Action



Slide window up to 6

SR: Dilemma

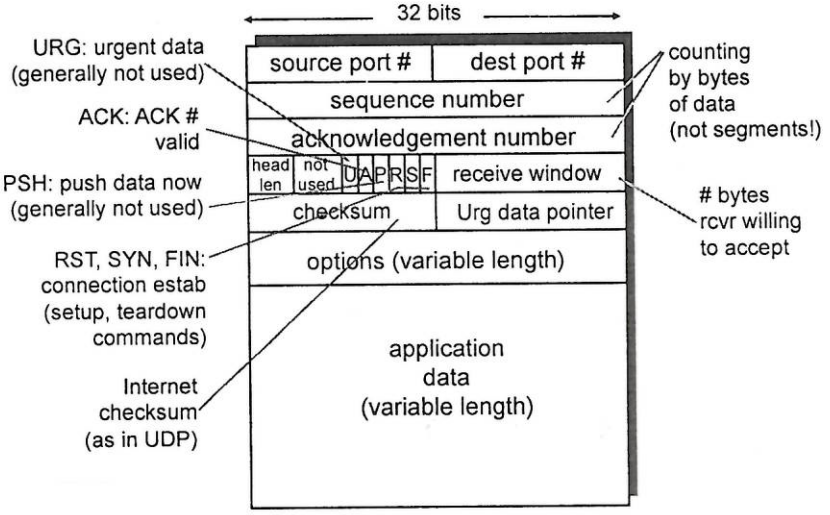
- Example
  - seq #'s: 0, 1, 2, 3
  - Window size = 3
- Receiver sees no difference in two scenarios!
  - Duplicate data accepted as new in (b)
- Exercise: What should be the relationship between seq # size and window size to avoid problem in (b)?



TCP: Overview

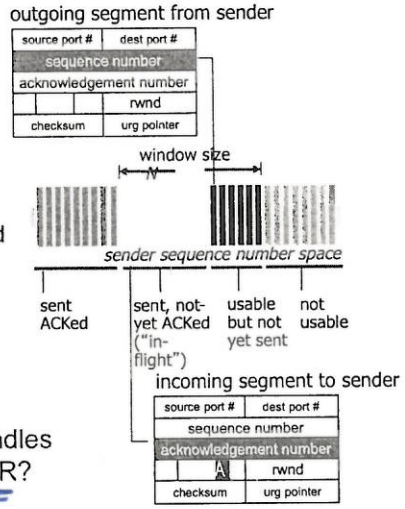
- Point-to-Point
  - One sender, one receiver
- Reliable, in-order byte steam
  - No "message boundaries"
- Pipelined
  - TCP congestion and flow control set window size
- Full Duplex Data
  - Bi-directional data flow in same connection
  - MSS: maximum segment size (max frame size)
    - ethernet - 1500 bytes
    - MTU minus 40 bytes TCP/IP header
- Connection-Oriented
  - Handshaking (exchange of control msgs) initialize sender, receiver state before data exchange
- Flow Controlled
  - Sender will not overwhelm receiver

TCP Header



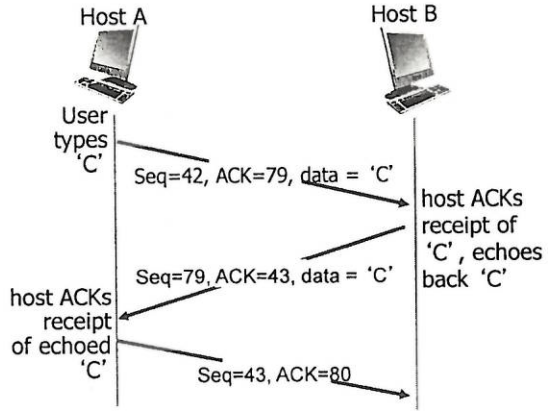
TCP Seq. Numbers & ACKs

- Sequence Numbers
  - Byte stream "number" of first byte in segment's data
- Acknowledgements
  - Seq # of next byte expected from other side



Exercise: How does receiver handles out-of-order segments GBN or SR?

Piggybacking



simple telnet scenario

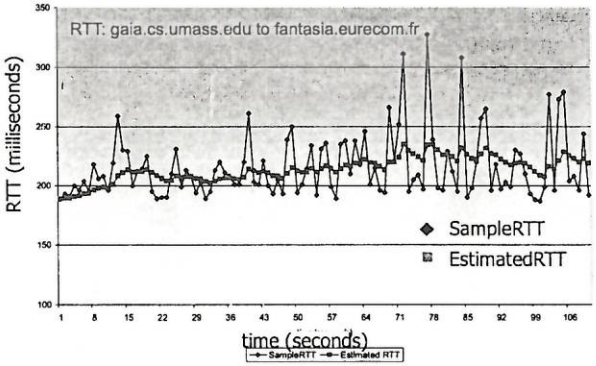
TCP Round-Trip Time (RTT)

- How to set TCP timeout value to recover from lost segments?
  - Longer than RTT - but RTT varies
  - Too short
    - premature timeout, unnecessary retransmissions
  - Too long
    - slow reaction to segment loss
- How to estimate RTT?
  - SampleRTT: measured time from segment transmission until ACK receipt
    - Ignore retransmissions
  - SampleRTT will vary due to congestion and load at routers
    - Want EstimatedRTT - smoother
      - Average several recent measurements, not just current SampleRTT

Average RTT

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

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$



Example RTT Calculation

Pkt #	SampleRTT	EstimatedRTT α = 0.125
20		40
21	30	39
22	80	44
23	40	43
24	90	49
25	50	?

- What happens if α is too small (say very close 0)
  - Sudden, real change in network load does not get reflected in Estimated RTT fast enough
    - May lead to under- or overestimation of RTT for a long time
- What happens if α is too large (say very close 1)
  - Transient fluctuations/changes in network load affects EstimatedRTT and makes it unstable when it should not
    - Also leads to under- or overestimation of RTT

Variations in RTT (DevRTT)

- Timeout Interval: `EstimatedRTT` plus "safety margin"
- Estimate SampleRTT deviation from EstimatedRTT

`DevRTT = (1-β * DevRTT +`
`β|SampleRTT-EstimatedRTT|`
`(typically, β= 0.25)`

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT — "safety margin"

- For TCP initial value of TimeoutInterval = 1 second
  - Value of TimeoutInterval doubled when a timeout occurs.

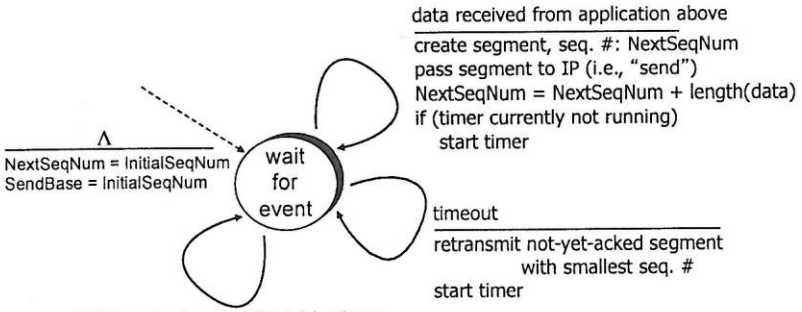
TCP Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
  - Pipelined Segments
  - Cumulative ACKs
  - Single Retransmission Timer
- Retransmissions triggered by
  - Timeout events
  - Duplicate ACKs
- Let us 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 ACK acknowledges previously unacked segments
    - Update what is known to be ACKed
    - Start timer if there are still unacked segments

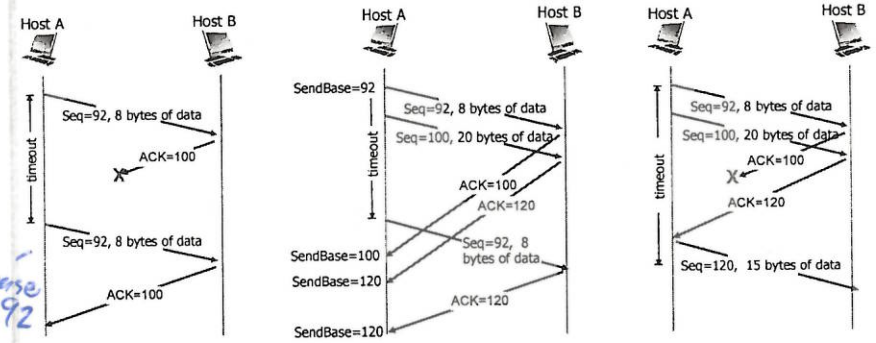
TCP Sender (Simplified)




```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
      start timer
   else stop timer
}
```

SendBase = 92

TCP: Retransmission Scenarios



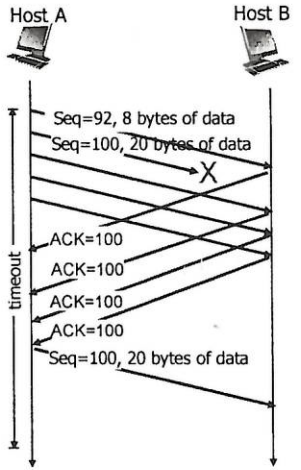
TCP ACK Generation

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 starts at lower end of gap

Q: Is TCP a GBN or SR protocol? Both

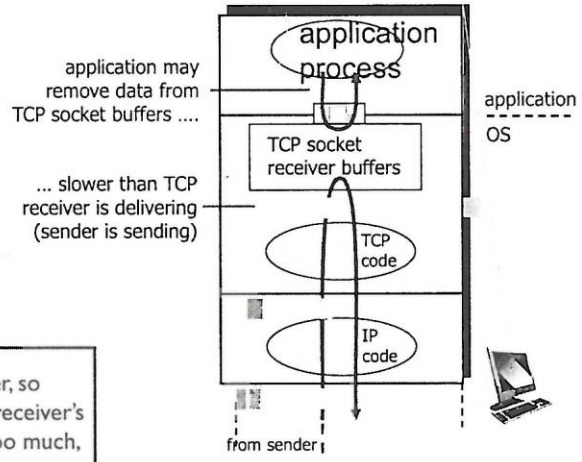
TCP Fast Retransmit

- 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
- TCP Fast Retransmit
  - If sender receives 3 ACKs for same data - "Triple Duplicate ACKs"
    - Resend unacked segment w. smallest seq #
  - Likely that unacked segment lost, so do not wait for timeout



fast retransmit after sender receipt of triple duplicate ACK

TCP Flow Control



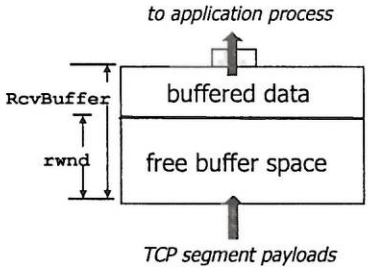
receiver protocol stack

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

TCP Flow Control II

- Receiver "advertises" free buffer space by including *rwnd* (receive window) value in TCP header of receiver-to-sender segments
  - RcvBuffer size set via socket options
    - Typical default is 4096 bytes
  - Many operating systems auto-adjust RcvBuffer

rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

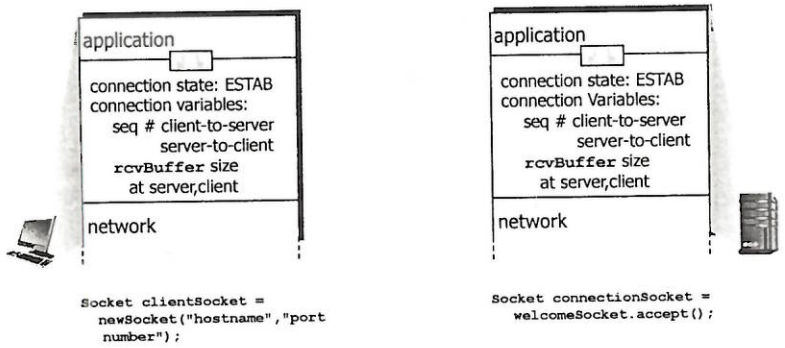


receiver-side buffering

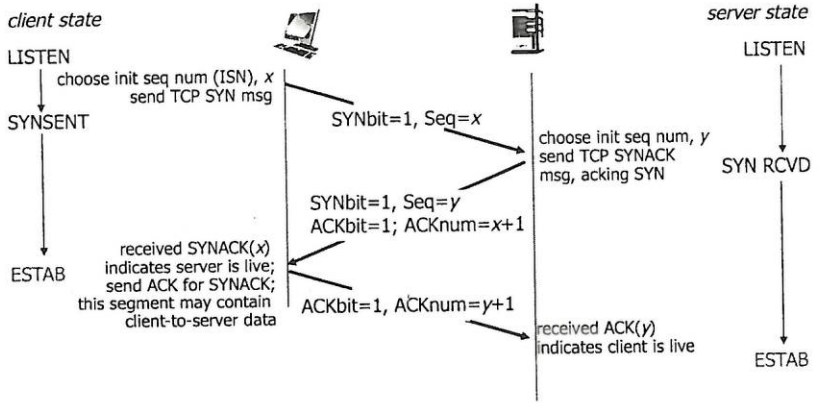
- Sender limits amount of unacked ("in-flight") data to receiver's *rwnd* value
  - Guarantees receive buffer will not overflow

Connection Management

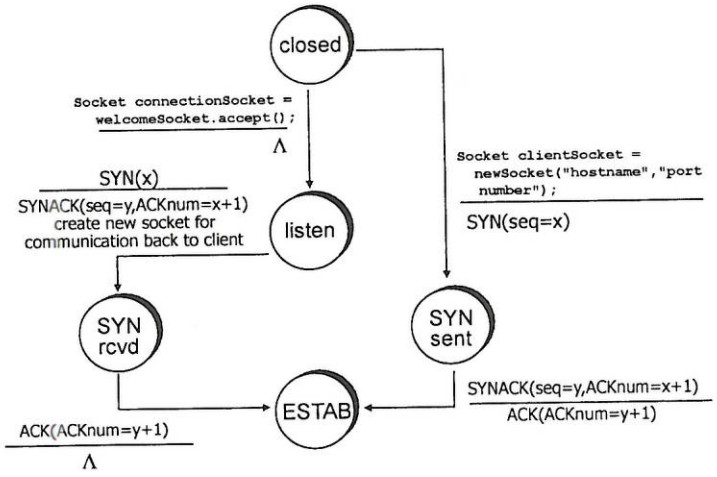
- Before exchanging data, sender/receiver "handshake"
  - Agree to establish connection (each knowing the other willing to establish connection)
  - Agree on connection parameters



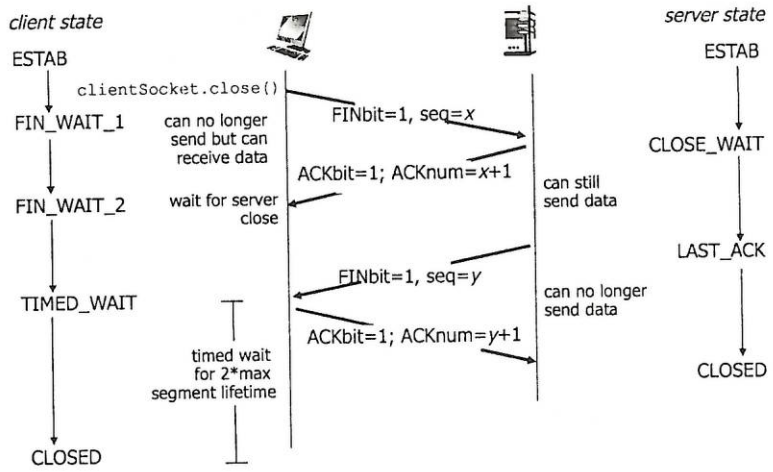
TCP 3-way Handshake



TCP 3-way Handshake FSM

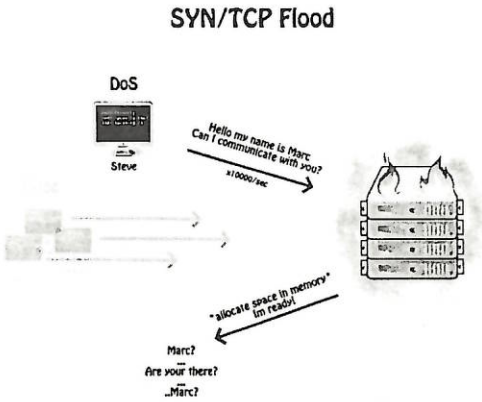


TCP: Closing a Connection



SYN Flood Attack (DoS)

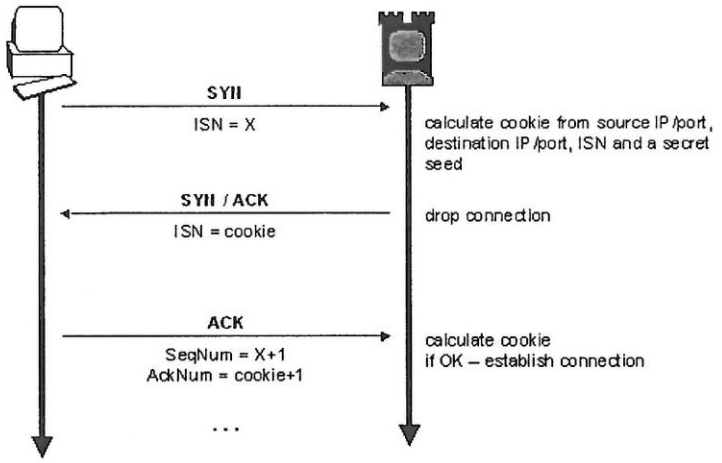
- Attacker sends large number of TCP SYN segments
  - Without completing the third handshake step



SYN Cookies

- Server does not know if the SYN segment is coming from a legitimate user
- Server creates an initial sequence number (ISN) or "cookie" from the hash of
  - Src IP addr & Port
  - Dest IP addr & Port
  - Timestamp
- Server sends SYNACK
  - Maintains no state info about corresponding to the SYN
- A legitimate client will return an ACK segment
  - Use the cookie information (ISN+1) in the ACK

SYN Cookies II



Principles of Congestion Control

- Informally:
  - "Too many sources sending too much data too fast for network to handle"
    - Different from flow control
- Manifestations
  - Lost packets
    - Buffer overflow at routers
  - Long delays
    - Queuing in router buffers

Approaches Towards Congestion Control

- End-to-End Congestion Control
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP
- Network-Assisted Congestion Control
  - Routers provide feedback to end systems
  - single bit indicating congestion (ATM)
  - Explicit rate for sender to send at

ATM ABR Congestion Control

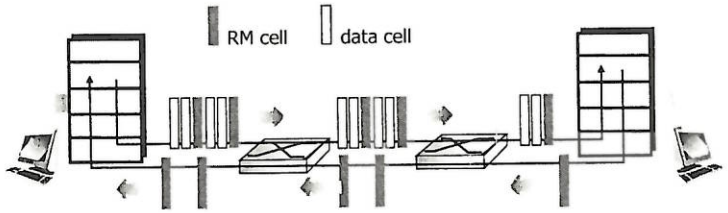
ABR: Available Bit Rate

- "Elastic Service"
- If sender's path "underloaded":
  - Sender should use available bandwidth
- If sender's path congested:
  - Sender throttled to minimum guaranteed rate

RM (Resource Management) Cells

- Sent by sender, interspersed with data cells
- Bits in RM cell set by switches ("*network-assisted*")
  - *NI bit:* no increase in rate (mild congestion)
  - *CI bit:* congestion indication
- RM cells returned to sender by receiver, with bits intact

ATM ABR Congestion Control II

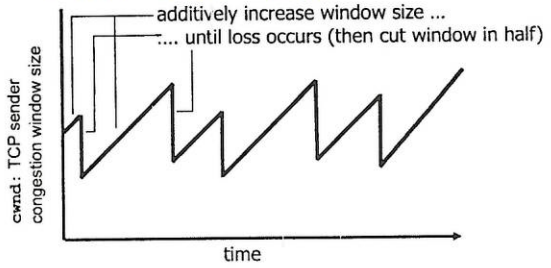


- Two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - Senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - If data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

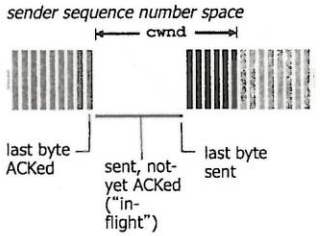
Additive Increase Multiplicative Decrease (AIMD)

- Sender increases transmission rate (window size)
  - Probing for usable bandwidth, until loss occurs
- Additive Increase
  - Increase congestion window (*cwnd*) by 1 MSS every RTT until loss detected
- Multiplicative Decrease
  - cut cwnd in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: Details



TCP sending rate

- Send *cwnd* bytes
- Wait RTT for ACKs
- Then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

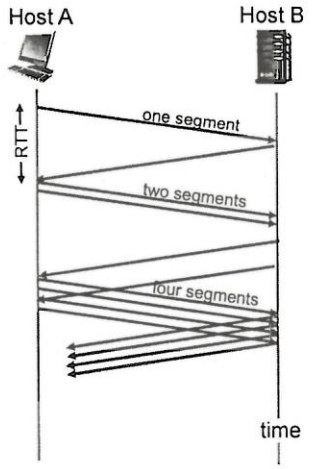
- Sender limits transmission

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

- *cwnd* is dynamic, function of perceived network congestion

TCP Slow Start

- When connection begins, increase rate exponentially until first loss event
  - Initially *cwnd* = 1 MSS
  - Double cwnd every RTT
  - Done by incrementing *cwnd* for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



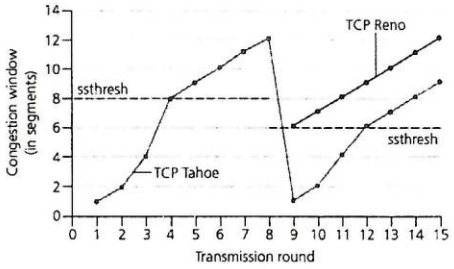
TCP: Detecting and Reacting to Loss

- Loss indicated by timeout
  - *cwnd* set to 1 MSS
- Window then grows exponentially (as in slow start) to threshold (*ssthresh*)
  - Then grows linearly
- Loss indicated by 3 duplicate ACKs: TCP Reno
  - Dup ACKs indicate network capable of delivering some segments
  - cwnd is cut in half window then grows linearly
- TCP Tahoe always sets *cwnd* to 1
  - Timeout or 3 duplicate ACKs

TCP: Switching from Slow Start to CA

Q: When should the exponential increase switch to linear?

A: When cwnd gets to 1/2 of its value before timeout



- Implementation
  - Variable *ssthresh*
  - On loss event, ssthresh is set to *cwnd*/2 just before loss event

Summary: TCP Congestion Control