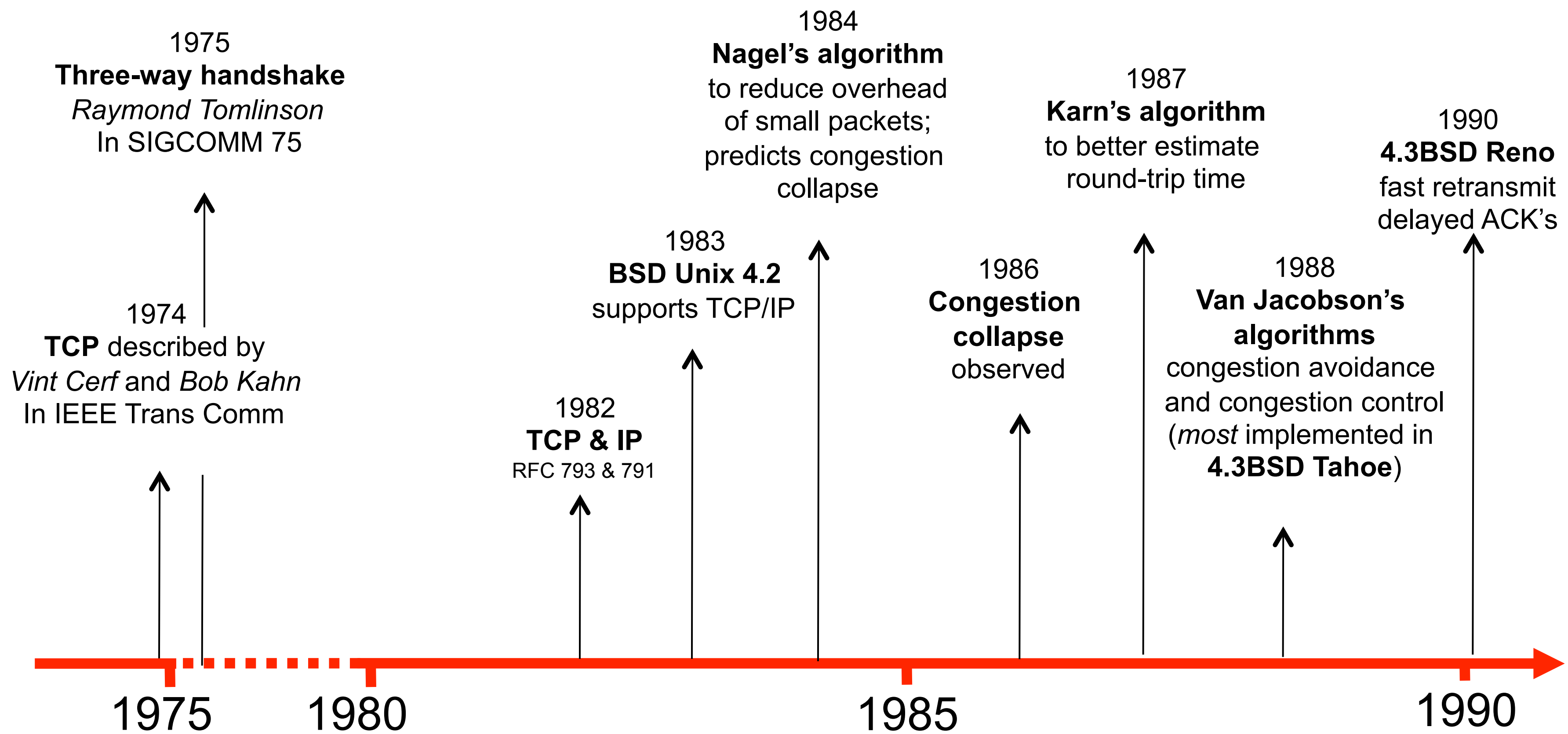
Last Lecture

- Overview of the transport layer
- Principles of Reliable Data Transfers
  - Error detection/correction
  - ACK/NACK & retransmission (ARQ)
  - Timeout
  - Sequence numbers
  - Sliding window protocols
    - Go back N
    - Selective repeat
  - Problems not addressed yet
    - Delayed duplicates
    - Timeout estimation

This Lecture

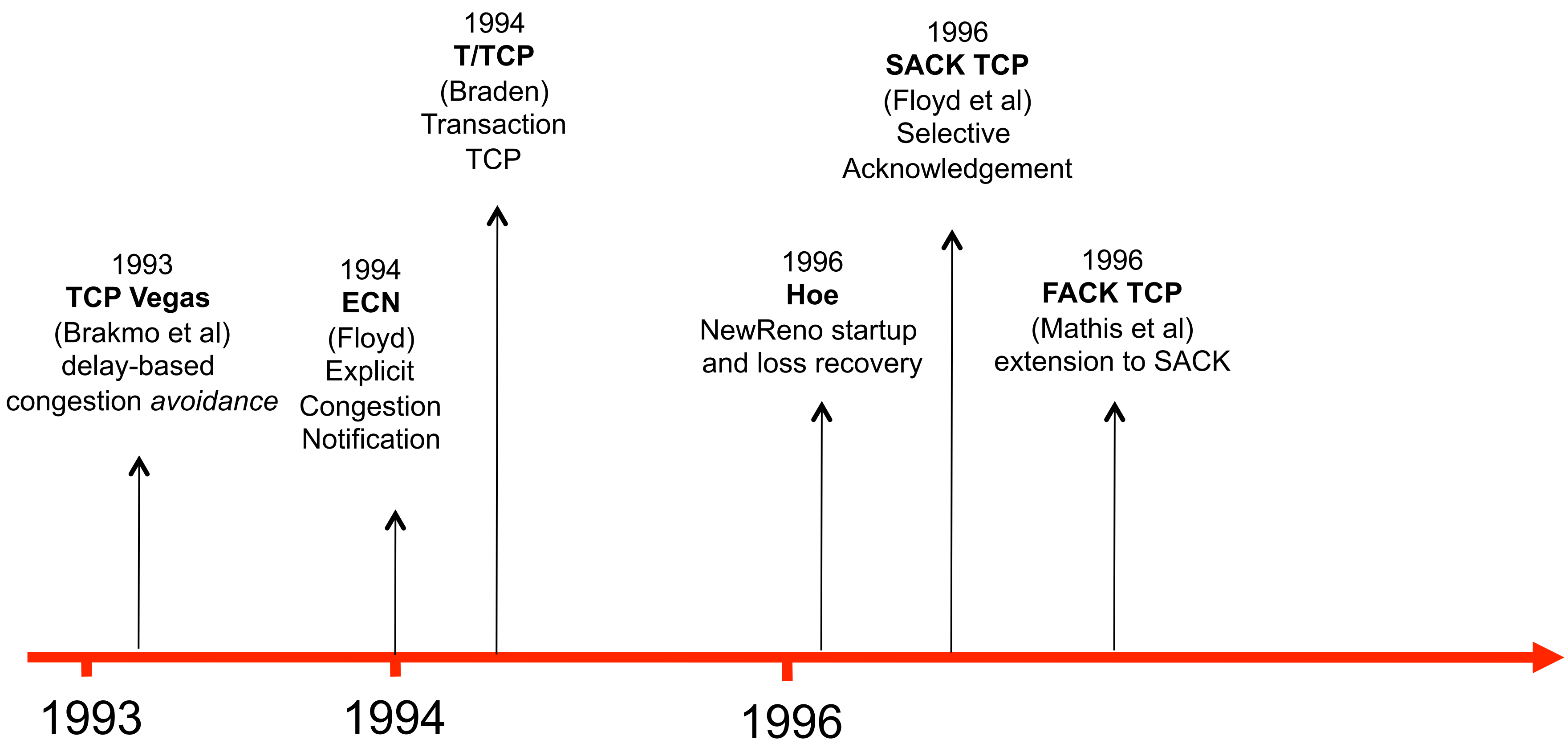
- How TCP Actually Works
  - Reliable *and efficient* data transfer
- Next lecture
  - Connection management
  - Flow control
- Congestion control will be addressed separately

TCP Evolution



Reno is the "least common denominator"

TCP Evolution



- This history is incomplete (see website & RFC 4614 for more links)
- Not all implementations implement all these features
- We won’t be able to cover every feature, only most common ones

Tips and Tricks

- Why TCP Tahoe, TCP Reno?

Answer

- TCP Tahoe: TCP implementation from 4.3BSD-Tahoe (released in June 1988)

The name Tahoe came from the development name used by Computer Consoles, Incorporated, for the machine that they eventually released as the Power 6/32. Computer Consoles gave CSRG a few machines to develop cross-platform BSD

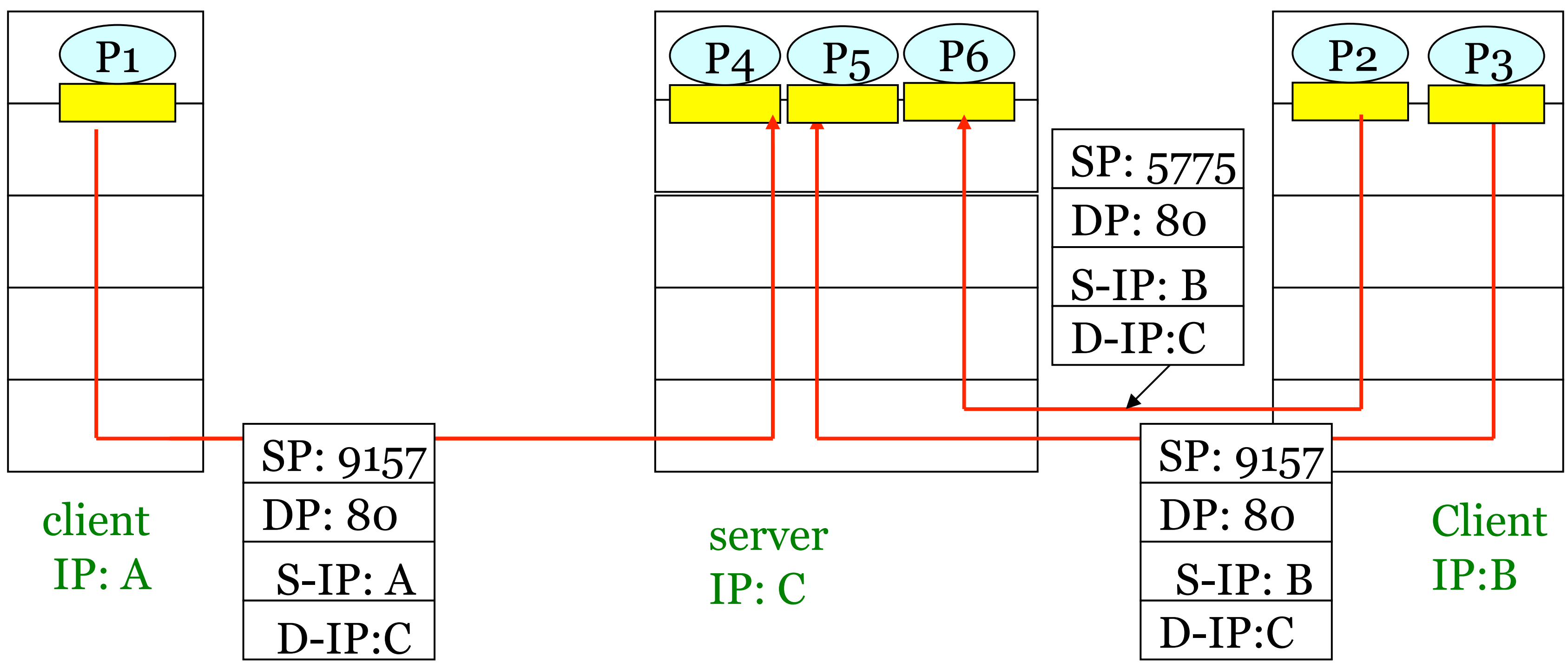
- TCP Reno: TCP implementation from 4.3BSD-Reno (released in 1988)

The release was named after a big gambling city in Nevada as an oblique reminder to its recipients that running the interim release was a bit of a gamble.

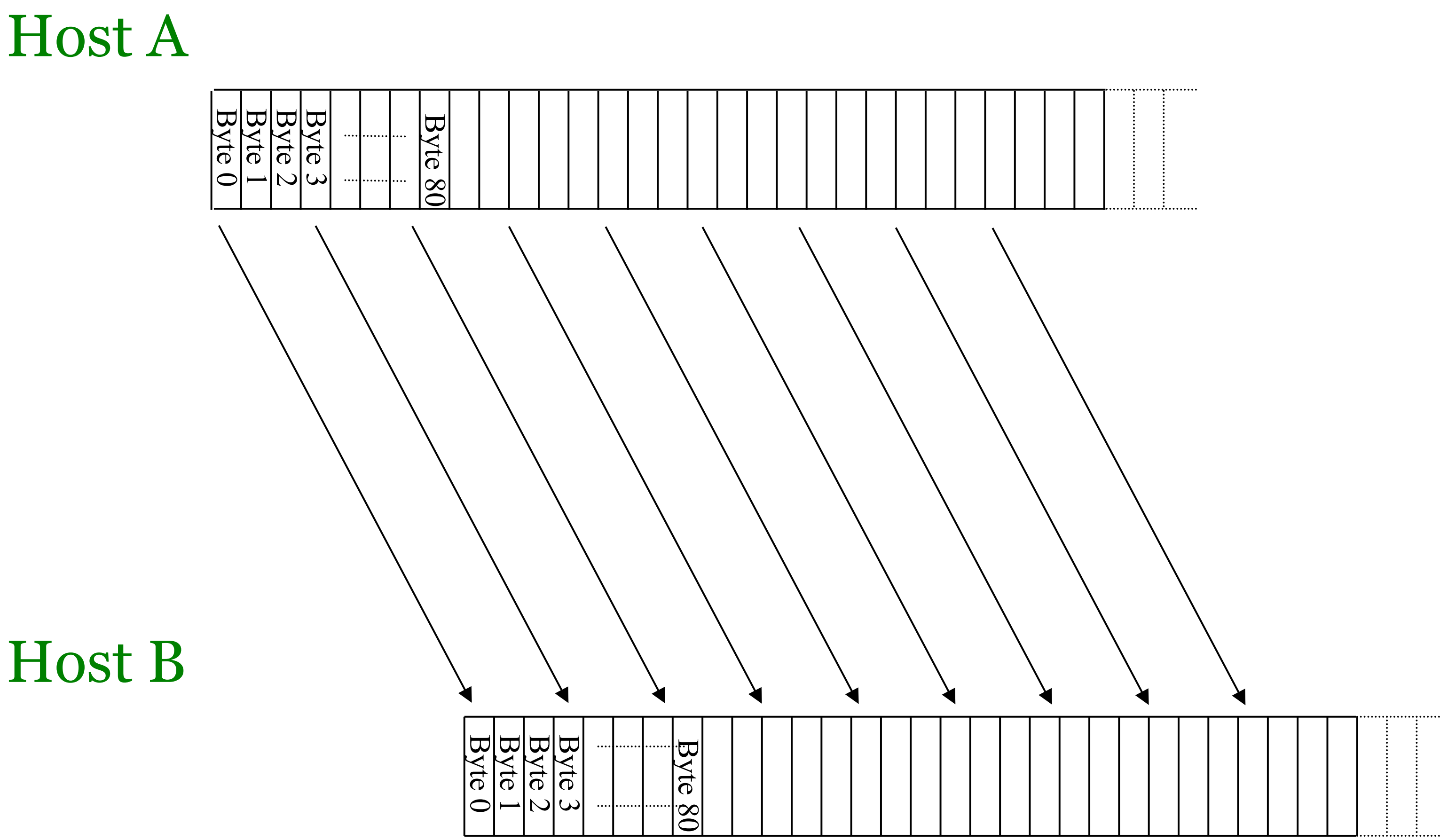
TCP Overview

1. *Multiplexing and Demultiplexing*
2. *Byte-stream service*
   - Stream of bytes sent and received, not stream of packets
3. *Reliable data transfer*
   - A combination of go-back-N and selective repeat
4. *Connection management*
   - Connection establishment and tear down
5. *Flow control*
   - Prevent sender from overflowing receiver
6. *Congestion control* (later)

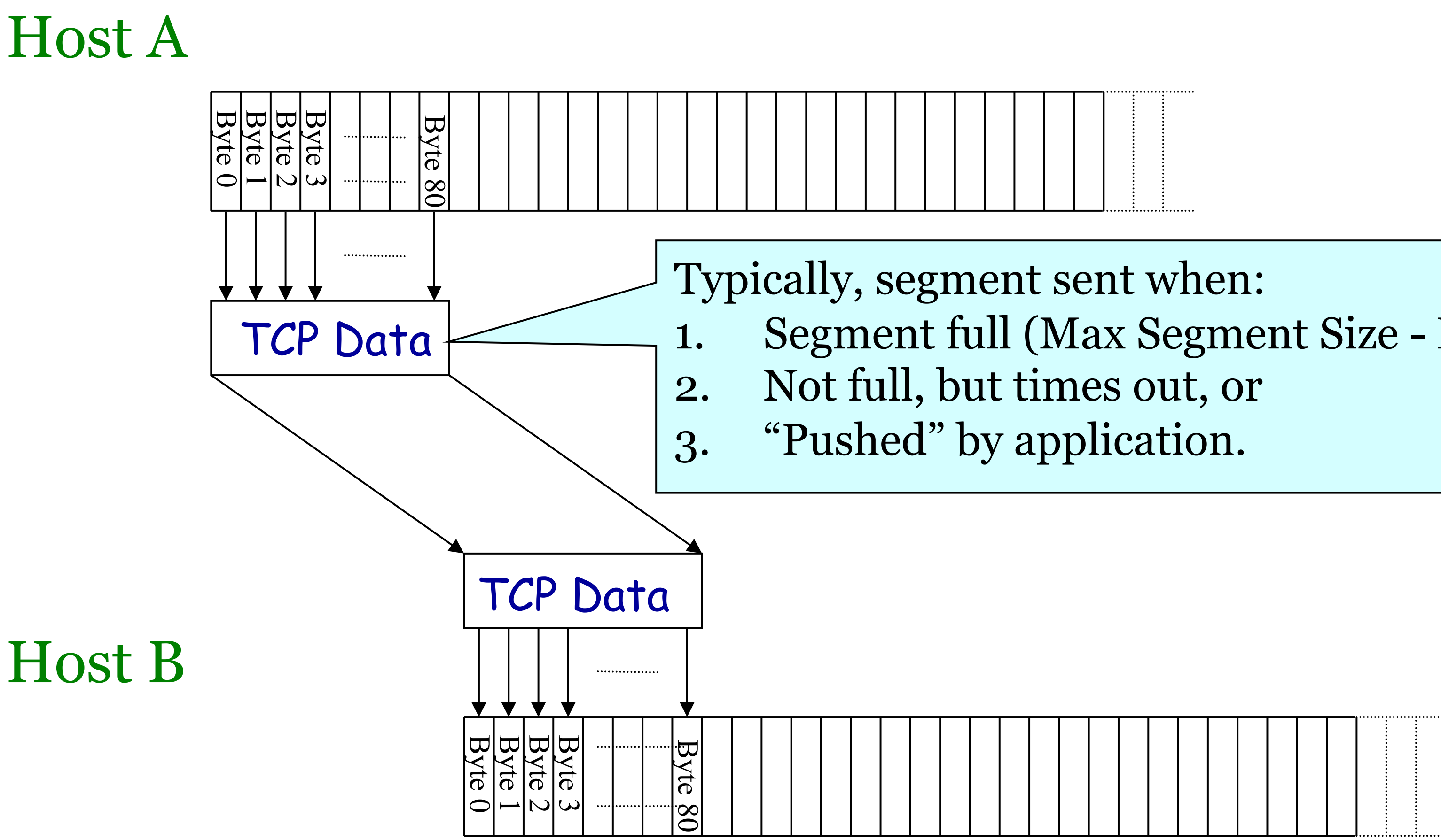
1. Multiplexing & De-multiplexing



2. TCP Byte-Stream Service



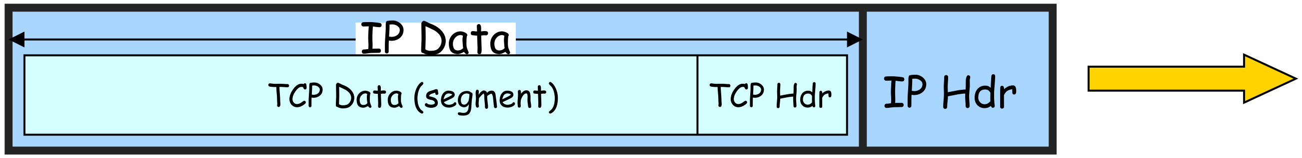
… Emulated by Breaking Up into *Segments*



Typically, segment sent when:

1. Segment full (Max Segment Size - MSS),
2. Not full, but times out, or
3. "Pushed" by application.

How Large Should a Segment Be?



- *IP packet size*
  - Should be ≤ Maximum Transmission Unit (MTU) along the path to the destination
  - E.g., Ethernet has MTU = 1500 bytes
- *IP Header + TCP Header* is typically 40 bytes
- *TCP data segment*
  - Should be ≤ Maximum Segment Size (MSS)
  - MSS should be MTU minus 40
  - E.g., up to 1460 consecutive bytes from the stream

Typical MTU for Various Networks

Hyperchannel	65535
16Mbps token ring (IBM)	17914
4Mbps token ring	4464
FDDI	4352
Ethernet	*1500*
802.3/802.2	1492
X.25	**576**

Maximum Segment Size (MSS)

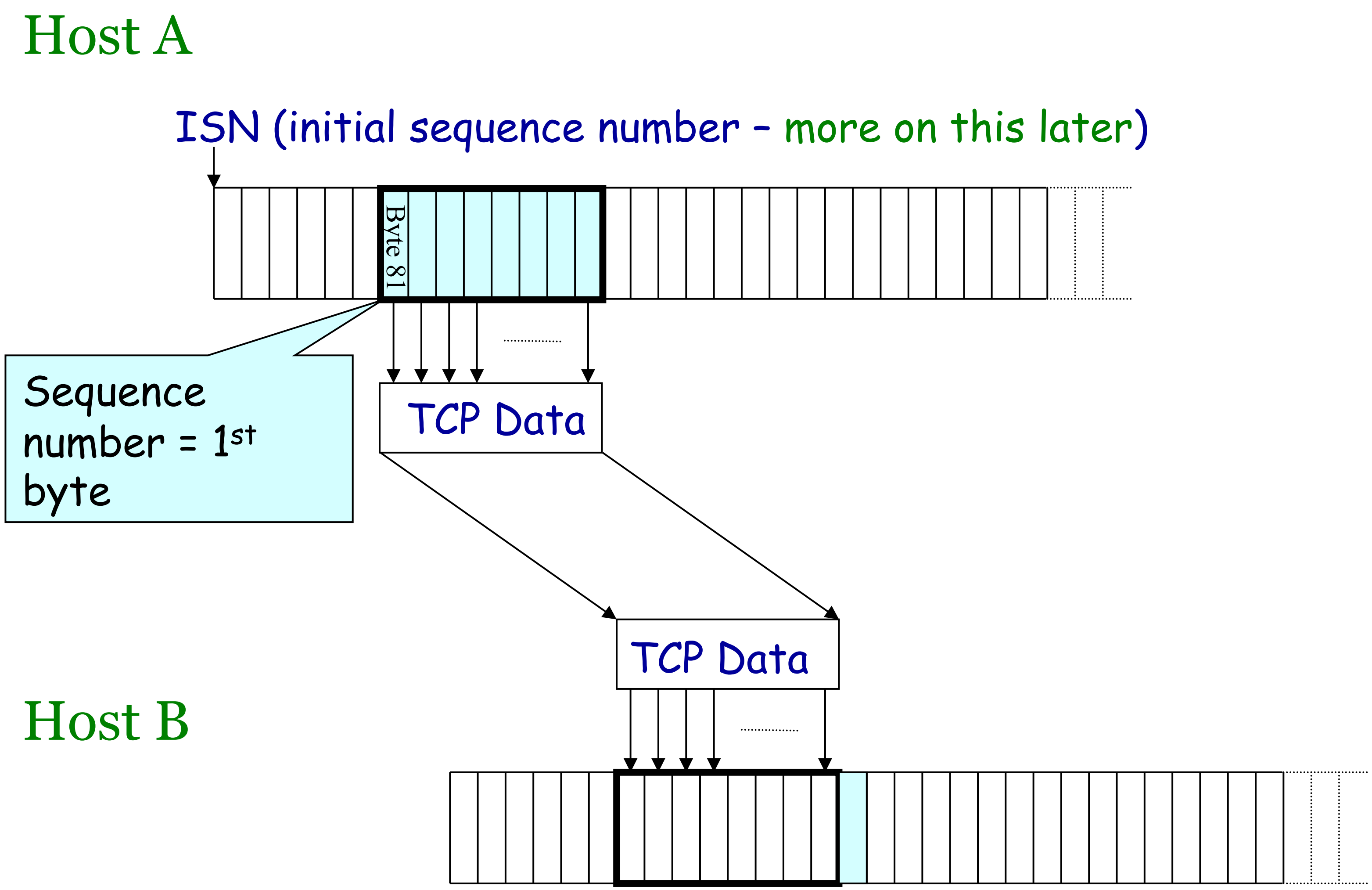
- MSS for opposite directions of the same connection might be different!
- MSS is negotiated at connect time
  - Remember the *small packet vs. large packet* tradeoff?
- TCP default MSS: **536** (which is 576-40)
- Implementation options:
  - At the very least least, TCP will check the outgoing interface MTU, minus IP and TCP header, to get max MSS
  - There's also a *path MTU discovery* mechanism

Path MTU Discovery (RFC 1191)

- Path MTU discovery algorithm:
  - Initially use *min*(MSS, MTU of the outgoing interface)
  - Set "Don't Fragment" (DF) bit for all transmissions
    - ICMP "*fragmentation needed*" is reported – when appropriate -- from a router with the next-hop MTU in it
    - TCP decreases its estimated MTU accordingly

- There are a few problems with this process
  - Security devices block ICMP packets
  - Path MTU might change; kernel periodically probes (about 10 minutes in Linux)

In TCP Every Byte Has a Sequence Number

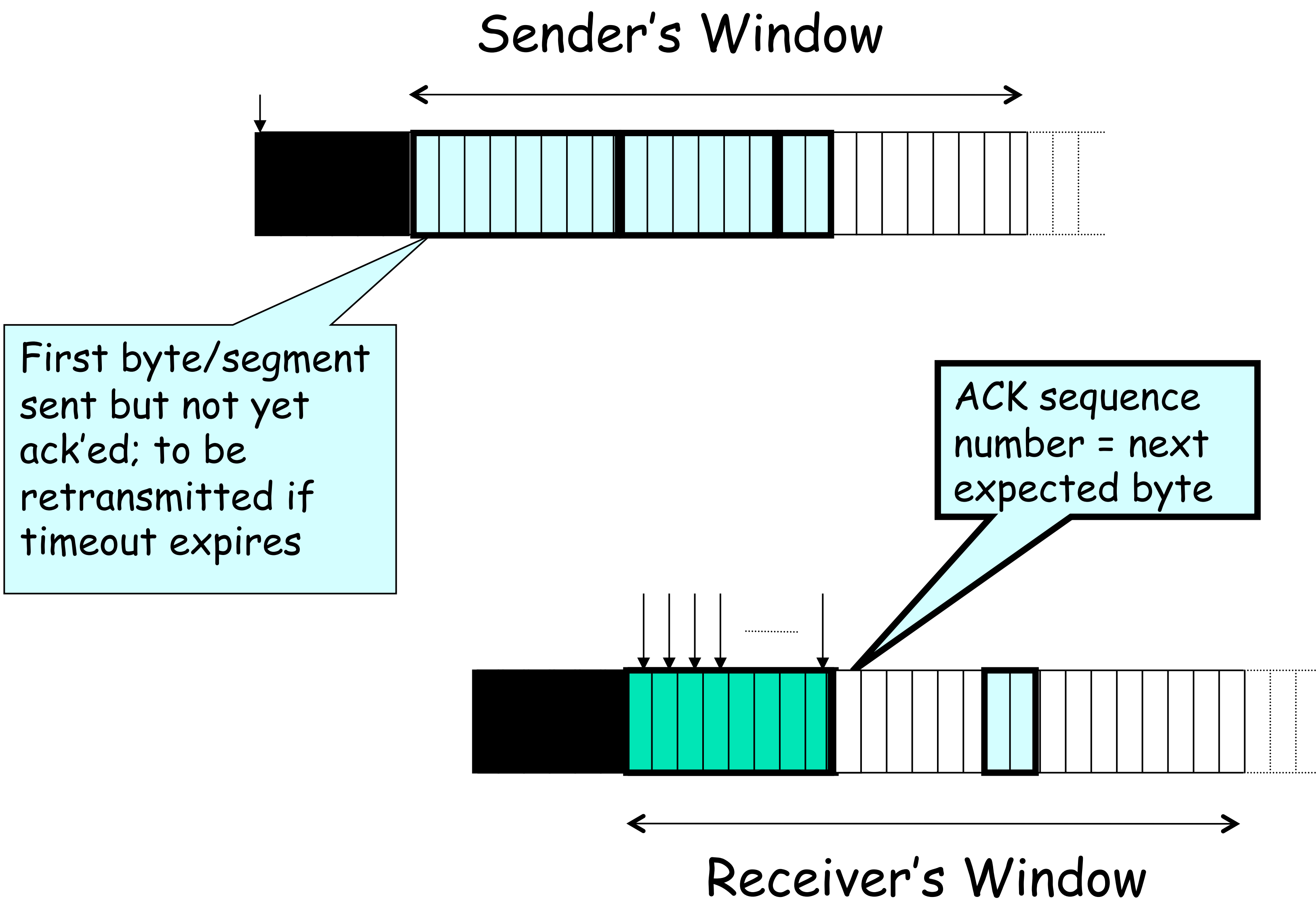


3. Basic TCP Reliable Data Transfer

- *Basic TCP* (for TCP/IP stacks of the 90's) is a variation of the *go-back-N* protocol
  - One single *timer* for all outstanding segments
  - When a timer expires, the first segment is retransmitted
  - Major implementations **do** buffer out of order segments if within window (basic RFCs do not require this!)
  - ACKs are *cumulative*, if sender receives ACK up to byte # *n*, then it will not retransmit bytes with # < *n*

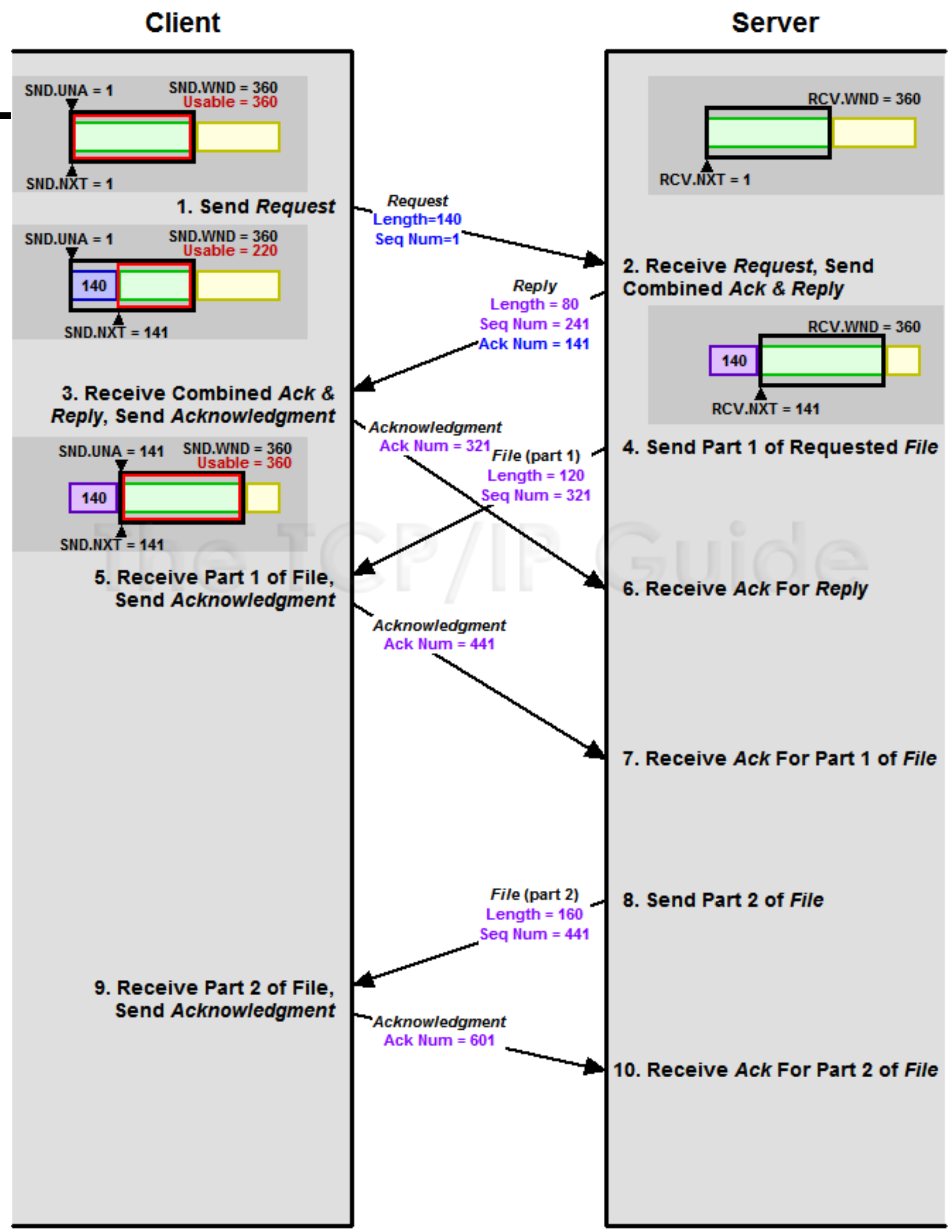
- More about extensions beyond the basic TCP later
  - Implementation dependent
  - Following all the RFCs makes the implementation very complicated

Sender's and Receiver's Windows

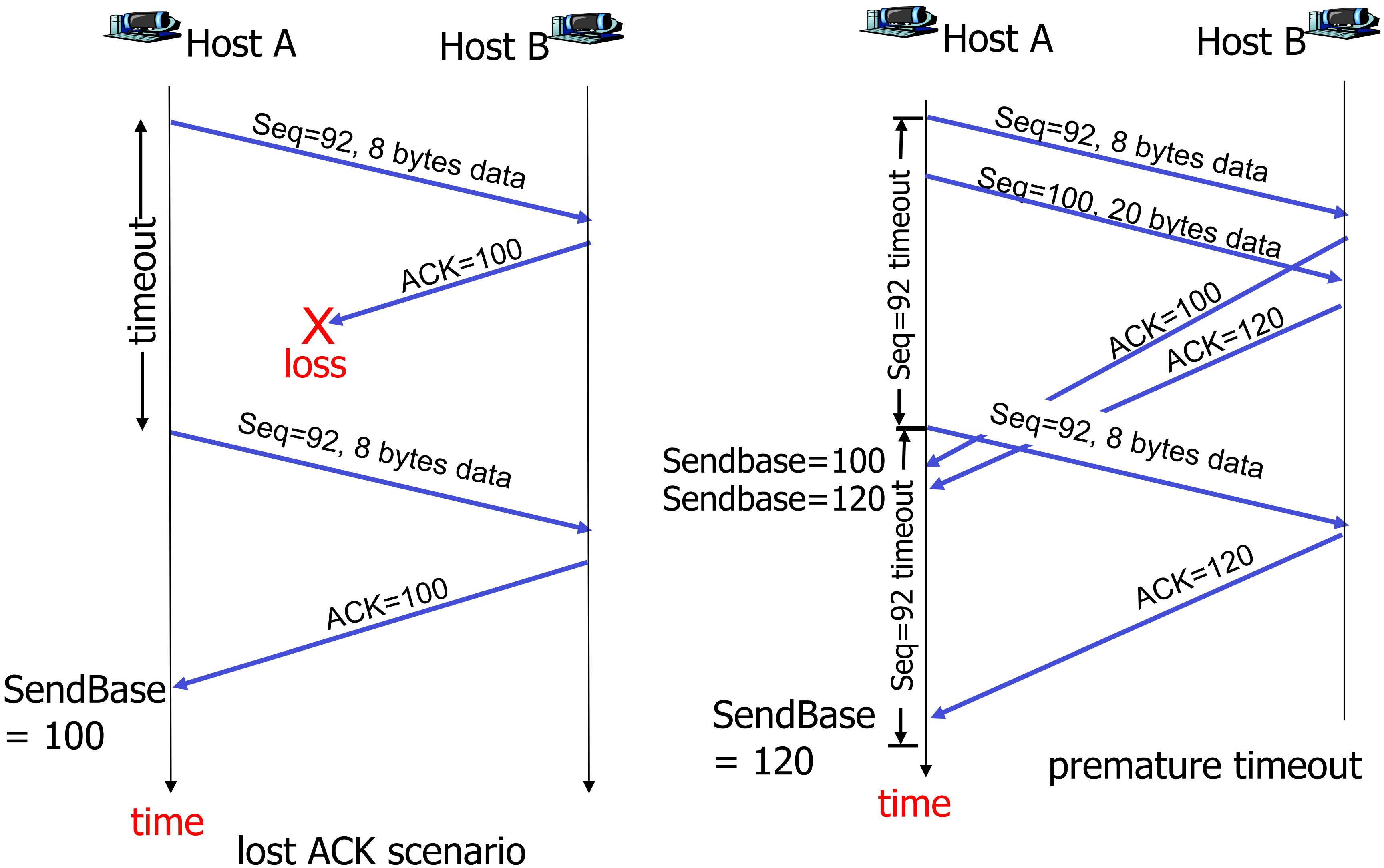


TCP's Cumulative ACKs and Full-Duplex Operation.

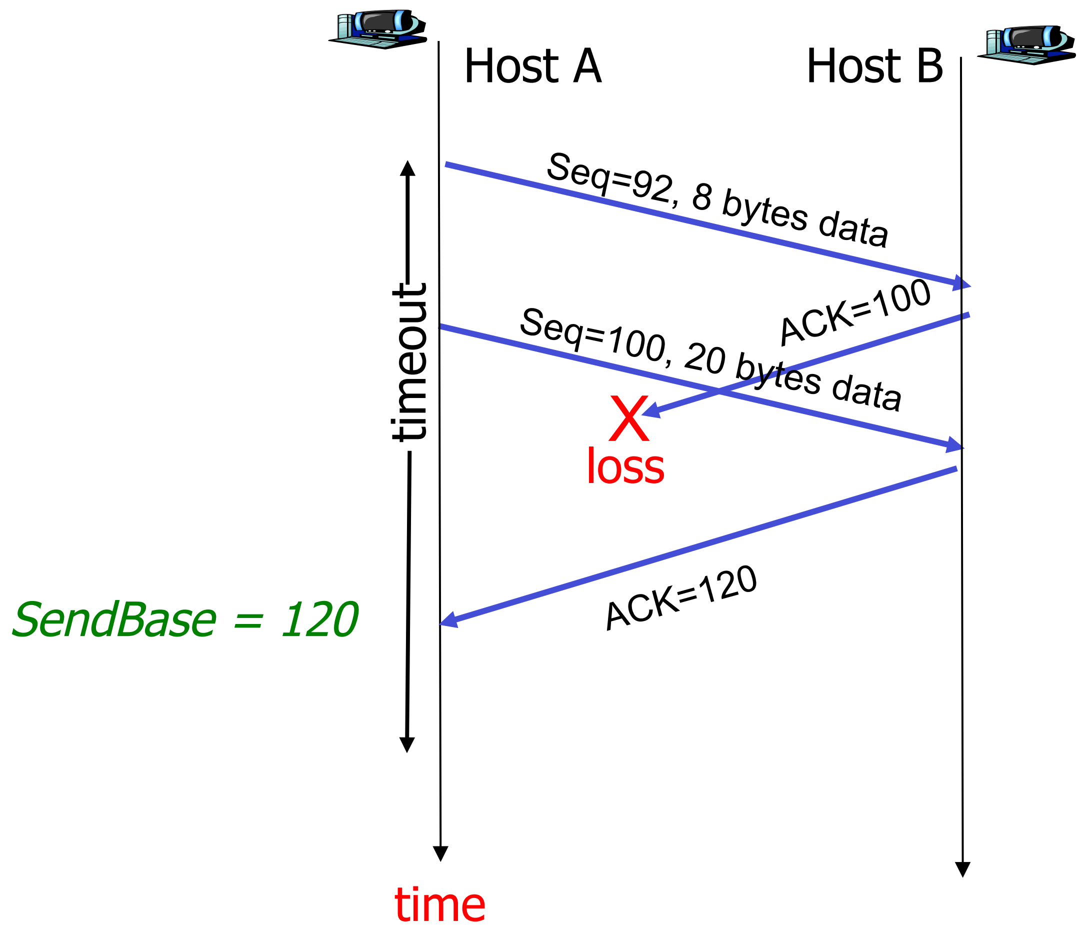
Note the *Piggy-Backing* of ACKs in the replies



TCP's Typical Retransmission Scenarios



TCP’s Cumulative ACK Scenario

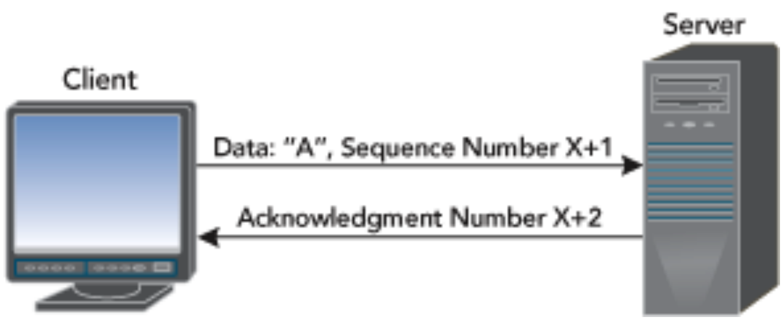


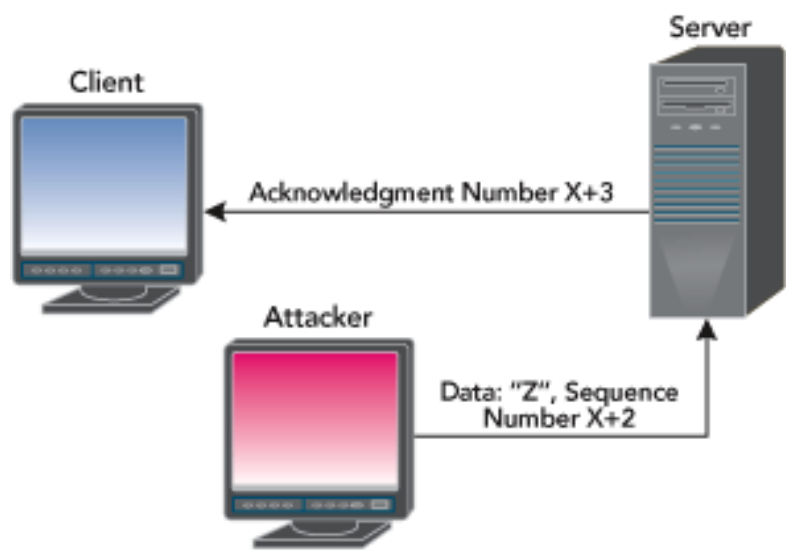
TCP ACK Generation [RFC 1122, RFC 2581]

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

Tips and Tricks

- *(TCP/UDP) Session Hijacking*
  - *How do you know you're talking to the party you're supposed to be talking to?*
  - Many toolkits available for script kiddies
  - Susceptible applications: telnet, ftp, dns, rlogin, rsh
  - (Partial) solution: ssh, SSL, IPSec, and the likes

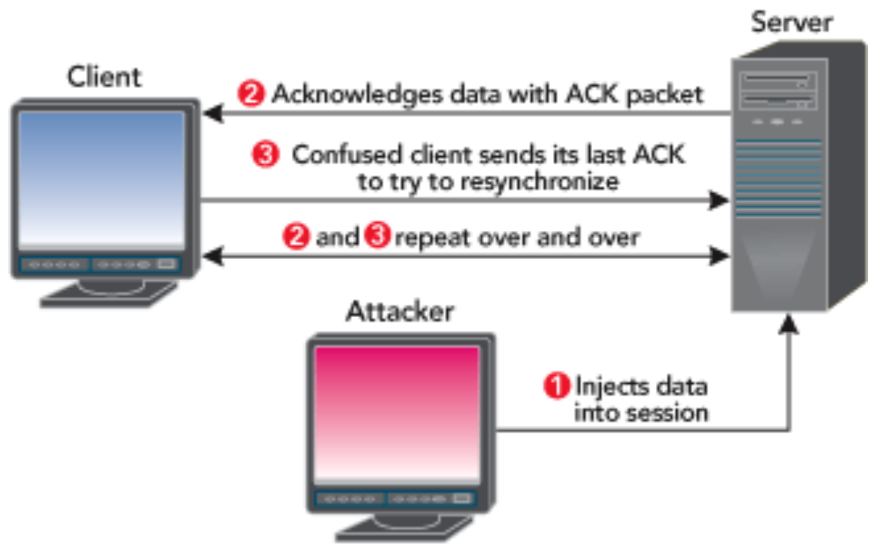




Now, if this was a `telnet` session, replace 'Z' by '`rm *`' ☺

Tips and Trics

- *TCP ACK Storm*



- 28/07/2006: CERT advisory – No. 2006/VULN414
  - Solaris Hosts are Vulnerable to a DoS induced by a TCP "ACK Storm”
  - Product: Solaris 8, 9, and 10
  - Solution: install a patch, which stops replying after a few bad ACKs

How's *Retransmission Timeout* Computed?

- Ideally, *RTO* should be *just a little* more than RTT

- *Question*: but RTT fluctuates
- Answer:
  - Take sample RTT *R* and "smooth" it out to get SRTT
  - Set *RTO* = some function of SRTT

- *Question*: but initially there's no *R* yet
- Answer: (RFC 2988)
  - Before having the first *R*, set *RTO* = 3sec
  - (But also use *exponential backoff*.)

Exponential Back-off

- This is implementation dependent

- On BSD, it goes something like
  - By default RTO = 1.5 sec
  - First retransmission: RTO
  - *n*th retransmission: 2^{n-1} RTO
  - up to 64 sec (implementation specific)

- On Windows, I think you can edit some registries to set these (and many other) parameters

After the First Sample RTT *R* is Measured

- SRTT = R
- *RTTVAR* = R/2
  - (RTTVAR is RTT's variance)
- RTO = SRTT + max (G, 4*RTTVAR)
  - Where G is the clock's granularity (in seconds)
  - Thus, typically RTO = SRTT + 4*RTTVAR

For Each Newly Measured *R*

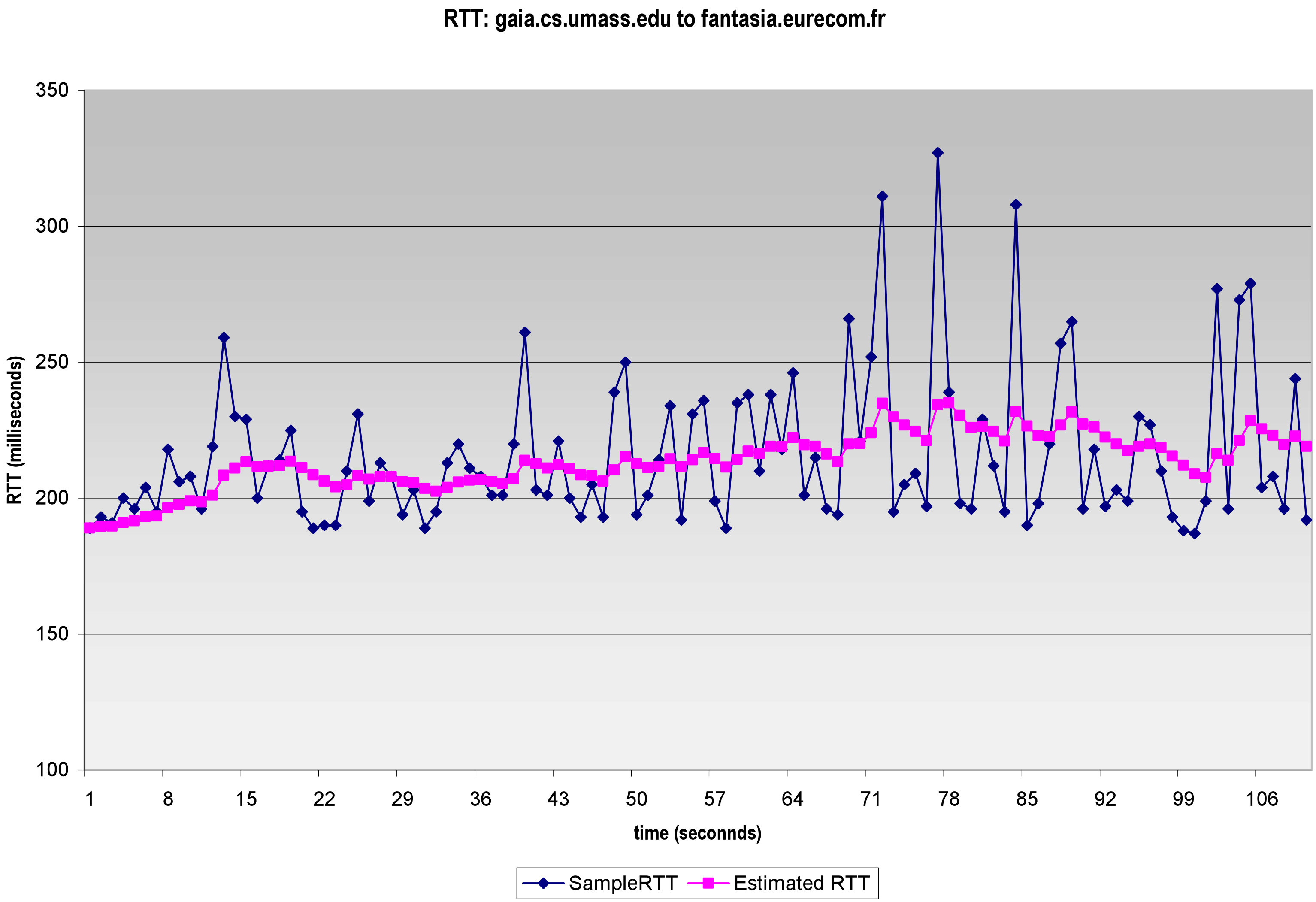
- $RTTVAR = (1 - \beta) * RTTVAR + \beta * |SRTT - R|$
  - Typical value: $\beta = 1/4$

- $SRTT = (1 - \alpha) * SRTT + \alpha * R$
  - Exponential weighted moving average
  - Influence of past sample decreases exponentially fast
  - Typical value: $\alpha = 1/8$

- They must be updated in the above order

- Finally, $RTO = SRTT + \max(G, 4*RTTVAR)$

Smoothed RTT vs. Real RTT



How to Measure Sample RTT *R*?

- *Method 1:*
  - Segment sent, timer start -------*R*------- ACK comes back
  - Flaw:
    - If we retransmitted the segment, no idea if ACK is for which copy
  - *Karn/Partridge Algorithm*:
    - Do not measure R using retransmitted segments
- *Method 2:*
  - TCP *timestamp* option
    - Sender stamps a packet with sending time
    - Receiver puts the stamp on the ACK
    - Sender subtracts current time from the stamp

More on Timer Management [RFC 2988]

- An implementation **MUST** manage the retransmission timer(s) in such a way that a segment is never retransmitted before RTO

RFC 2988: Recommended Timer Management

- Every time a packet containing data is sent (including a retransmission), if the timer is not running, start it running so that it will expire after RTO seconds (for the current value of RTO).
- When all outstanding data has been acknowledged, turn off the retransmission timer.
- When an ACK is received that acknowledges new data, restart the retransmission timer so that it will expire after RTO seconds (for the current value of RTO).
- When timer expires:
  - Retransmit oldest segment
  - Recompute RTO (double it)
  - Start new timer

Performance Tuning: Fast Retransmit

- Long RTO → long delay before retransmission
  - Need a way to detect loss packets **before** timing out
- *Idea*: 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.
- *Fast retransmit*
  - If sender receives 3 duplicate ACKs for the same data, it assumes that segment after ACKed data was lost
  - Resend segment before timer expires

Effectiveness of Fast Retransmission

- When does Fast Retransmit work best?
  - High likelihood of many packets in flight
    - Long data transfers
    - High window size
  - Low burstiness in packet losses
    - Higher likelihood that later packets arrive successfully
- Implications for Web traffic
  - Most Web transfers are short (e.g., 10 packets)
    - Short HTML files or small images
  - So, often there aren't many packets in flight
  - … making fast retransmit less likely to "kick in"
  - Forcing users to like "reload" more often… ☺