Networking (from an OS perspective)

Yin Lou

10/08/2009

Outline

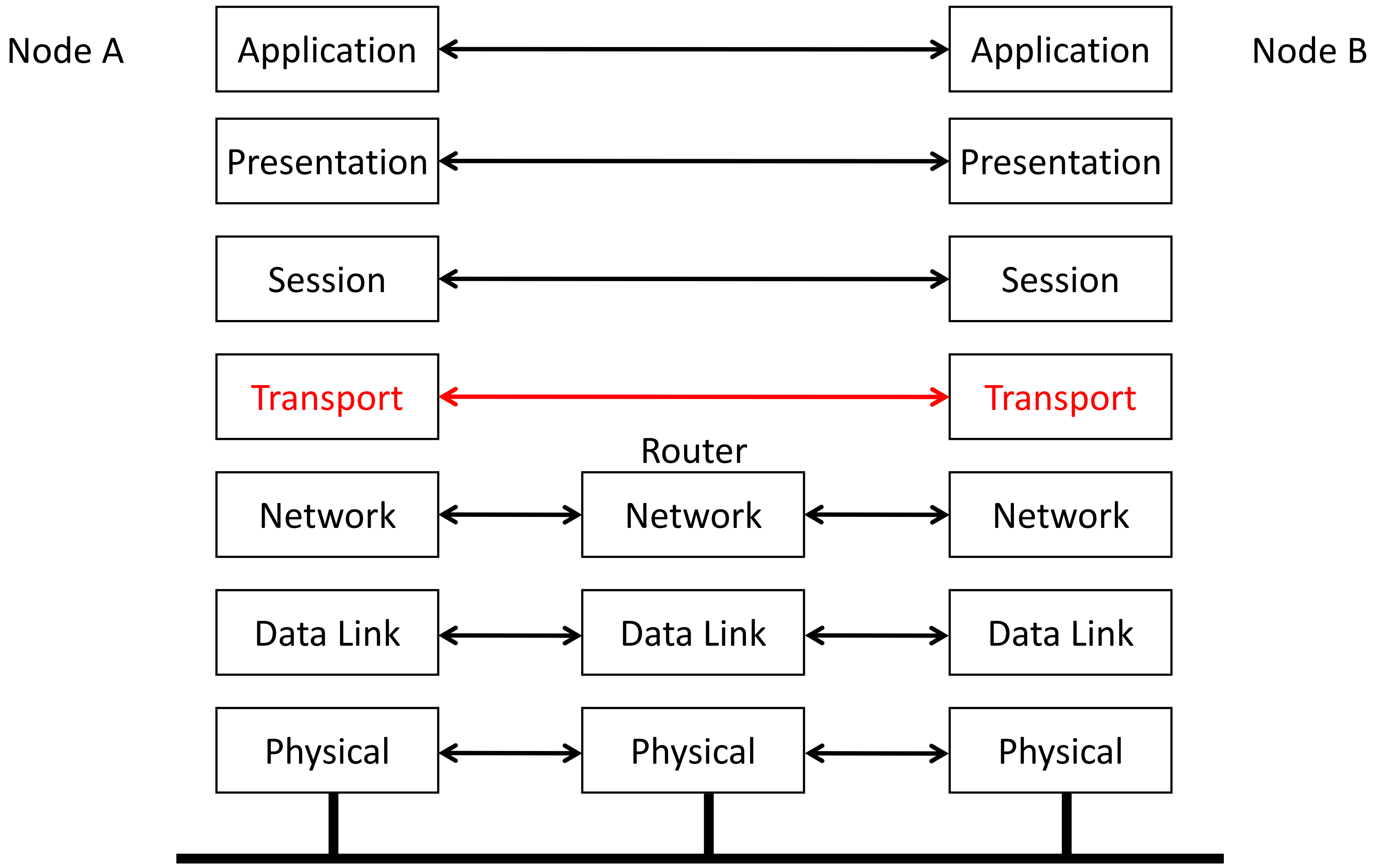
Background

- IEEE: “A Protocol for Packet Network Interconnection.”
  - In May, 1974
- Vint Cerf and Bob Kahn
  - Turing Award
    - "pioneering work on internetworking, including .. the Internet's basic communications protocols .. and for inspired leadership in networking."
  - “The father of the Internet”

Outline

- Background
- TCP Review
- Congestion Avoidance and Control
- TCP Congestion Control with a Misbehaving Receiver
- Summary

OSI Levels


Node A
Node B
Application
Application
Presentation
Presentation
Session
Session
Transport
Transport
Router
Network
Network
Network
Data Link
Data Link
Data Link
Physical
Physical
Physical

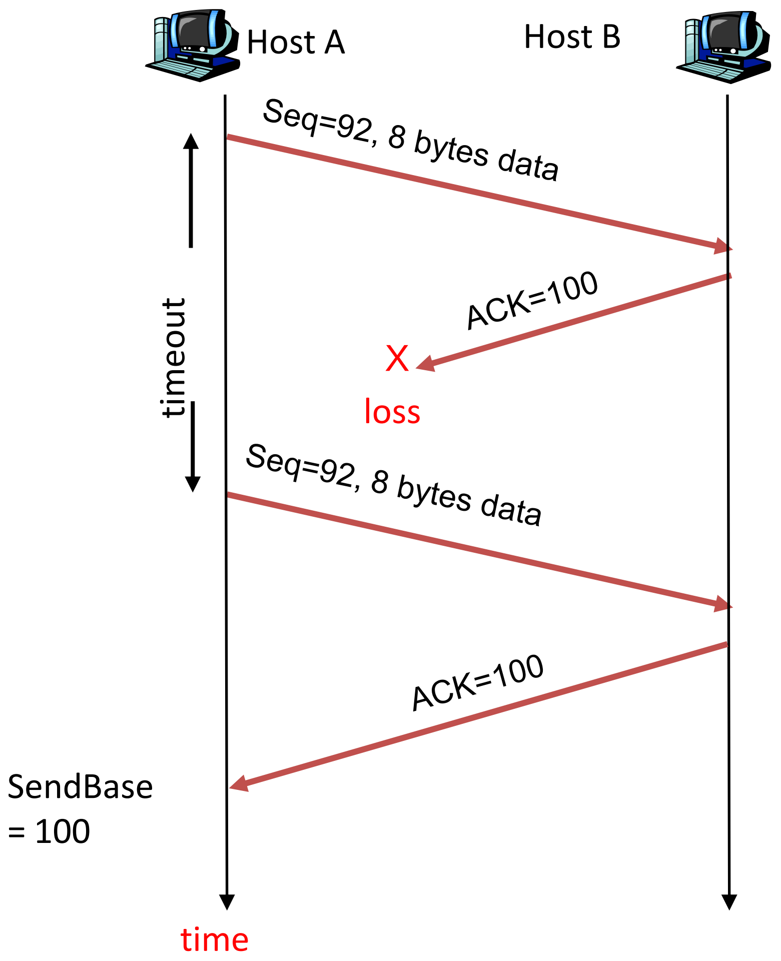
Overview

- Connection-oriented
  - Handshaking before data exchange
- Reliable, ordered, byte-stream protocol
- Full duplex data
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- Flow controlled
  - Sender will not overwhelm receiver

Sender Events

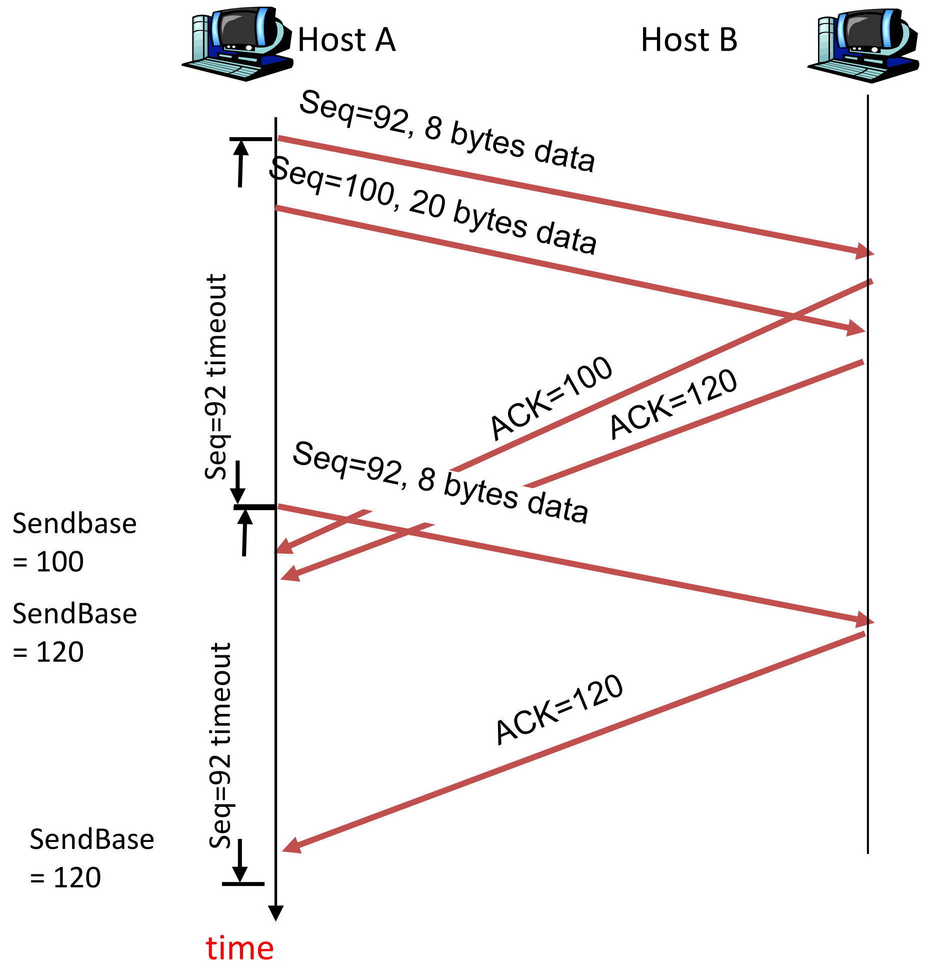
Event at Sender	TCP Sender action
Data received from app	Create segment with seq. # Start timer if not already running
Timeout	Retransmit segment that caused timeout Restart timer
ACK received	If acknowledges previously unacked segments Update what is known to be acked Start timer if there are outstanding segments

Retransmission



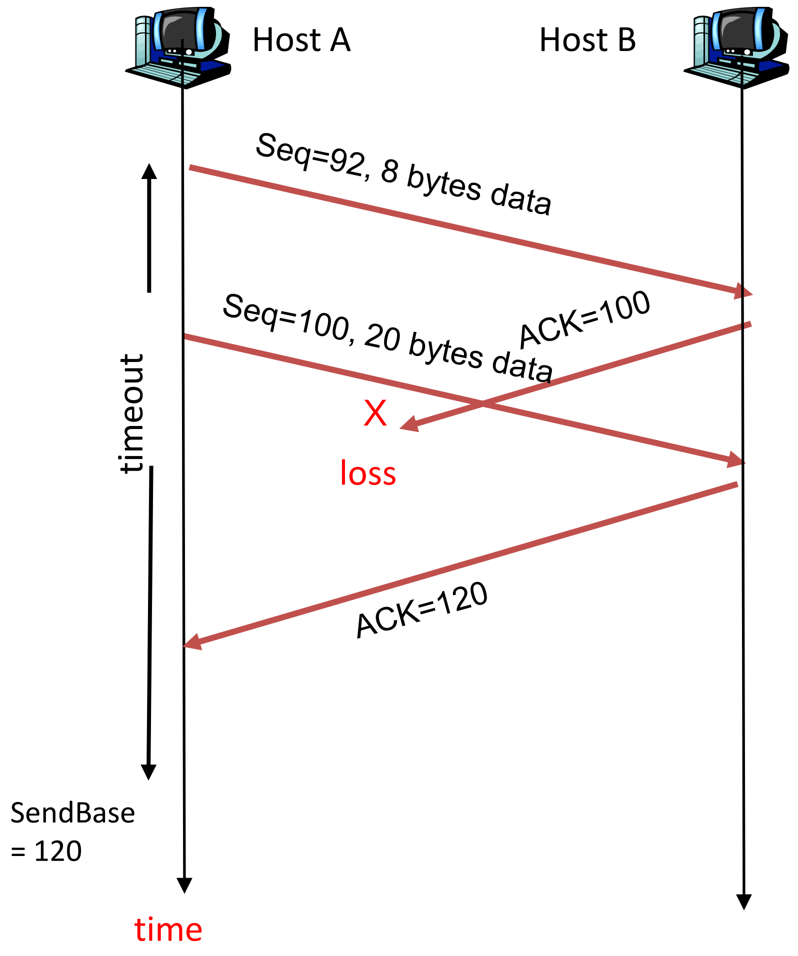
Lost ACK scenario

Retransmission



Premature timeout

Retransmission



Cumulative ACK scenario

Receiver Events

Event at Receiver	TCP Receiver action
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

Fast Retransmission

- Long time-out period
  - 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
- 3 ACKs indicates a segment loss
  - fast retransmit: resend segment before timer expires

Outline

- Background
- TCP Review
- Congestion Avoidance and Control
- TCP Congestion Control with a Misbehaving Receiver
- Summary

Congestion Avoidance and Control

- Motivation
  - Data throughput from LBL to UC Berkeley dropped from 32Kbps to 40 bps (factor-of thousand drop) in Oct.'86
- Conservation of packets
  - Connection "in equilibrium": running stably with a full window of data in transit
  - Packet flow "conservative": A new packet isn't put into the network until an old packet leaves.

4BSD TCP

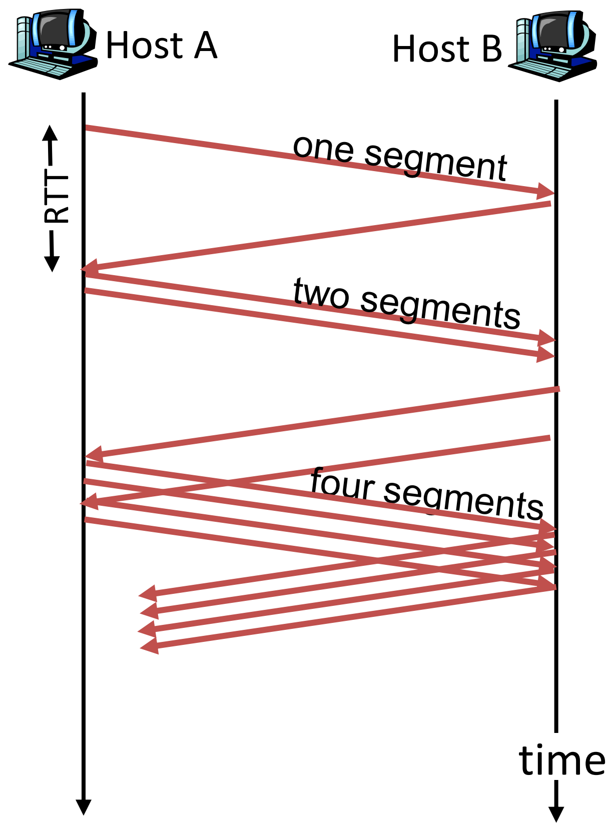
- slow-start
- round-trip-time variance estimation
- exponential retransmit timer backoff
- more aggressive receiver ACK policy
- dynamic window sizing on congestion
- Karn's clamped retransmit backoff
- fast retransmit

Packet Conservation to Fail

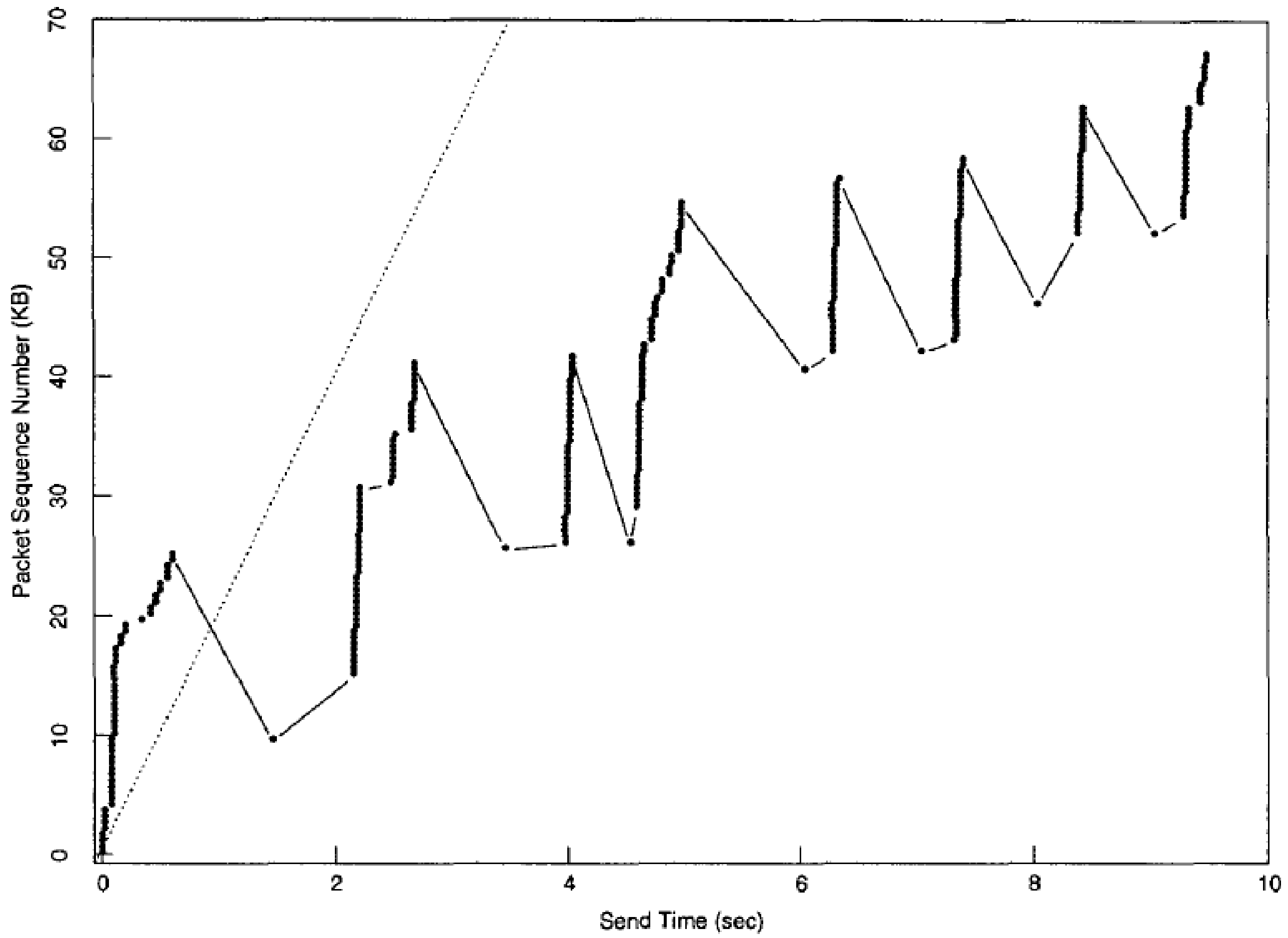
1. The connection doesn’t get to equilibrium
2. A sender injects a new packet before an old packet has exited
3. The equilibrium can’t be reached because of resource limits along the path

Getting to Equilibrium: Slow Start

- When connection begins, cwnd = 1 MSS (congestion window)
- Increase rate exponentially until first loss event
  - double cwnd every RTT
  - done by incrementing cwnd for every ACK received
- Summary
  - initial rate is slow but ramps up exponentially fast

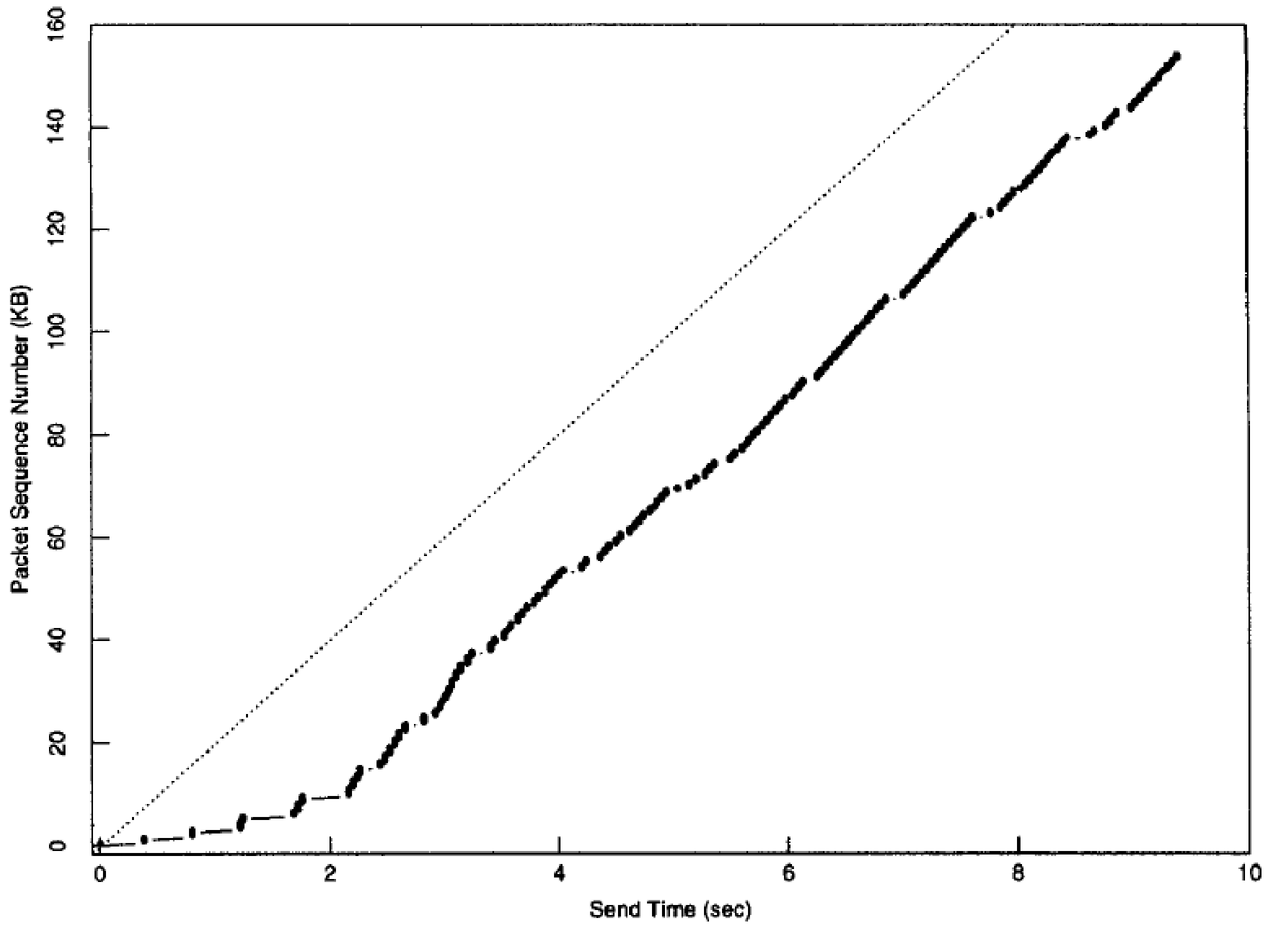


Getting to Equilibrium: Slow Start



Startup behavior of TCP without Slow-start

Getting to Equilibrium: Slow Start



Startup behavior of TCP with Slow-start

Packet Conservation

1. ~~The connection doesn’t get to equilibrium~~
2. A sender injects a new packet before an old packet has exited
3. The equilibrium can’t be reached because of resource limits along the path

Conservation at Equilibrium: RTT

- EstimatedRTT = α * EstimatedRTT + (1 - α) * SampleRTT
  - $\alpha = 0.9$, filter gain constant
- Timeout Interval = β * EstimatedRTT
  - B = 2
- Common mistakes
  - Not estimating the variation of the RTT
  - A connection will respond to load increases by retransmitting packets that have only been delayed in transit. Forces the network to do useless work

Conservation at Equilibrium: RTT

- EstimatedRTT = (1 - g) * EstimatedRTT + g * SampleRTT
  - The paper suggests g = 0.125, the gain
- EstimtedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- How much SampleRTT deviates from EstimatedRTT:
  - DevRTT = 0.75* DevRTT + 0.25 * |SampleRTT-EstimatedRTT|
- Timeout Interval = EstimatedRTT + 4 * DevRTT

Packet Conservation

1. ~~The connection doesn't get to equilibrium~~
2. ~~A sender injects a new packet before an old packet has exited~~
3. The equilibrium can't be reached because of resource limits along the path

Adapting to the Path: Congestion Avoidance

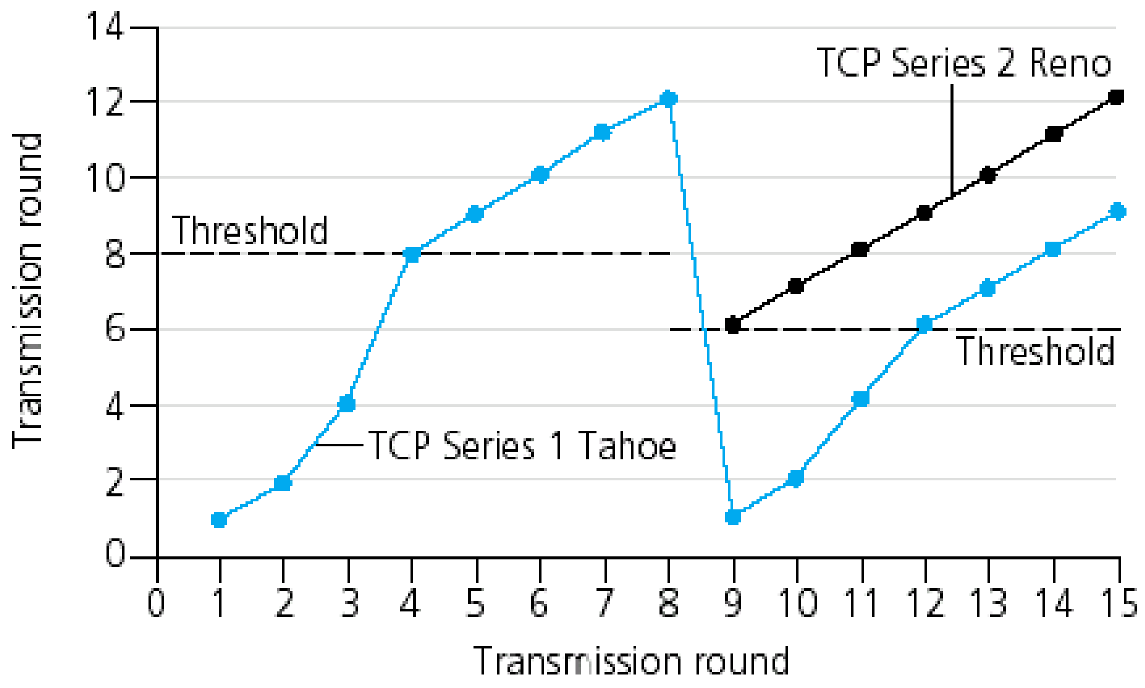
- Packets get lost for two reasons:
  - Damaged in transit (<< 1%)
  - Network is congested and somewhere on the path there was insufficient buffer capacity
- Congestion avoidance strategy:
  - Network must be able to signal the transport endpoints
  - The endpoints must have a policy

Adapting to the Path: Congestion Avoidance

- On no congestion (Additive Increase):
  - $\text{cwnd}_i = \text{cwnd}_{i-1} + u$ ($u \ll \text{cwnd}_{max}$)
  - The paper suggests $u = 1$
- On congestion (Multiplicative Decrease):
  - $\text{cwnd}_i = d * \text{cwnd}_{i-1}$ ($d < 1$)
  - The paper suggests $d = 0.5$

Adapting to the Path: Congestion Avoidance

- The combined slow-start with congestion avoidance algorithm
- Additive increase / Multiplicative decrease (AIMD)



Summary

- Slow start
- RTT estimation
- Congestion avoidance algorithm (AIMD)

Outline

- Background
- TCP Review
- Congestion Avoidance and Control
- TCP Congestion Control with a Misbehaving Receiver
- Summary

TCP Congestion Control with A Misbehaving Receiver

- Misbehaving receiver can achieve the same result of the misbehaving sender
  - Less obviously compared to a misbehaving sender
- TCP’s vulnerabilities arise from a combination of:
  - unstated assumptions
  - casual specification
  - a pragmatic need to develop congestion control mechanisms that are backward compatible with previous TCP implementations

Misbehaving Receiver – ACK Division

- RFC 2581:
  - During slow start, TCP increments cwnd by at most MSS bytes for each ACK received that acknowledges new data.
  - During congestion avoidance, cwnd is incremented by 1 full-sized segment per round-trip-time (RTT).
- Attack 1:
  - Receive a data segment containing N bytes
  - Receiver divides the resulting acknowledgement into M, where M ≤ N, separate acknowledgements
  - each covering one of M distinct pieces of the received data segment.

Misbehaving Receiver – ACK Division

- Growing cwnd at a rate that is M times faster

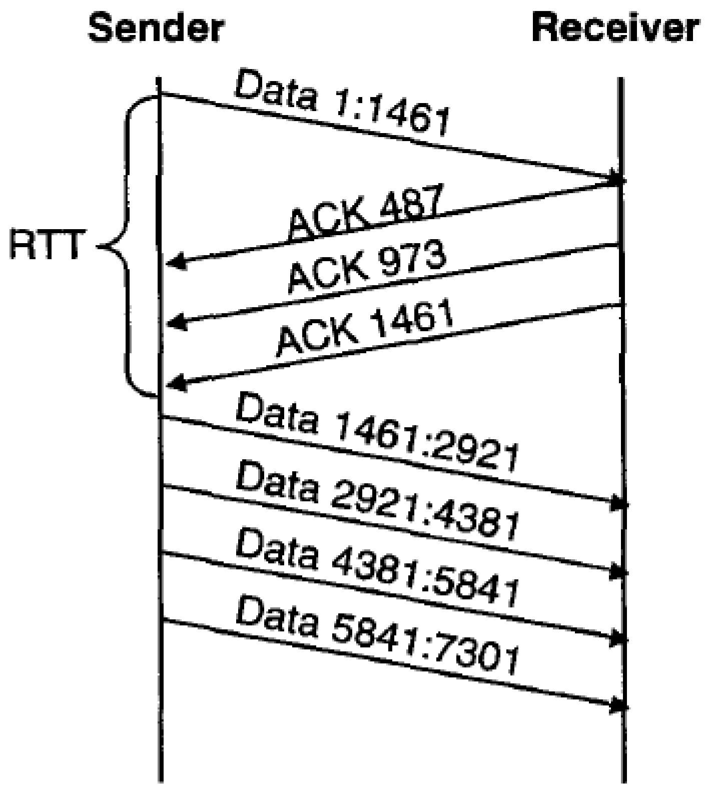


Figure 1: Sample time line for a ACK division attack. The sender begins with *cwnd*=1, which is incremented for each of the three valid ACKs received. After one round-trip time, *cwnd*=4, instead of the expected value of *cwnd*=2.

Misbehaving Receiver – DupACK Spoofing

- RFC 2581:
  - (Fast recovery) Set cwnd to ssthresh plus 3*MSS. This artificially "inflates" the congestion window by the number of segments(three) that have left the network and which the receiver has buffered.
  - For each additional duplicate ACK received, increment cwnd by MSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
- Problems:
  - It assumes that each segment that has left the network is full sized
  - TCP requires that duplicate ACKs be exact duplicates, there is no way to ascertain which data segment they were sent in response to.

Misbehaving Receiver – DupACK Spoofing

- Attack 2:
  - Receive a data segment
  - The receiver sends a long stream of acknowledgements for the last sequence number received.

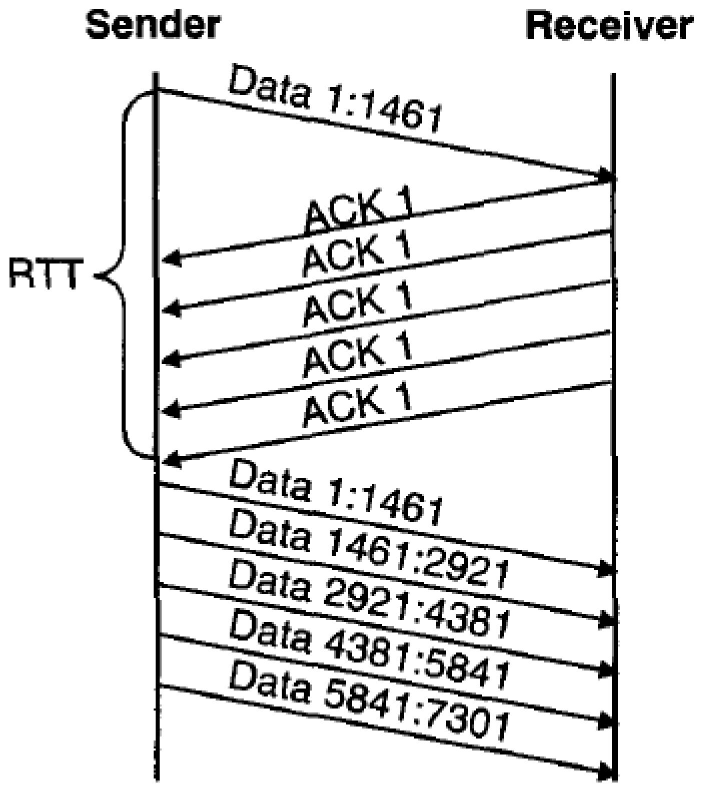


Figure 2: Sample time line for a DupACK spoofing attack. The receiver forges multiple duplicate ACKs for sequence number 1. This causes the sender to retransmit the first segment and send a new segment for each additional forged duplicate ACK.

Misbehaving Receiver – Optimistic ACKing

- Problems:
  - TCP’s algorithm assumes that the time between a data segment being sent and an acknowledgement for that segment returning is at least one round-trip time.
  - But the protocol does not use any mechanism to enforce its assumption.
- Attack 3:
  - Receive a data segment
  - The sender sends a stream of acknowledgements anticipating data that will be sent by the sender.

Misbehaving Receiver – Optimistic ACKing

- Data from the sender which is lost may be unrecoverable since it has already been ACKed.

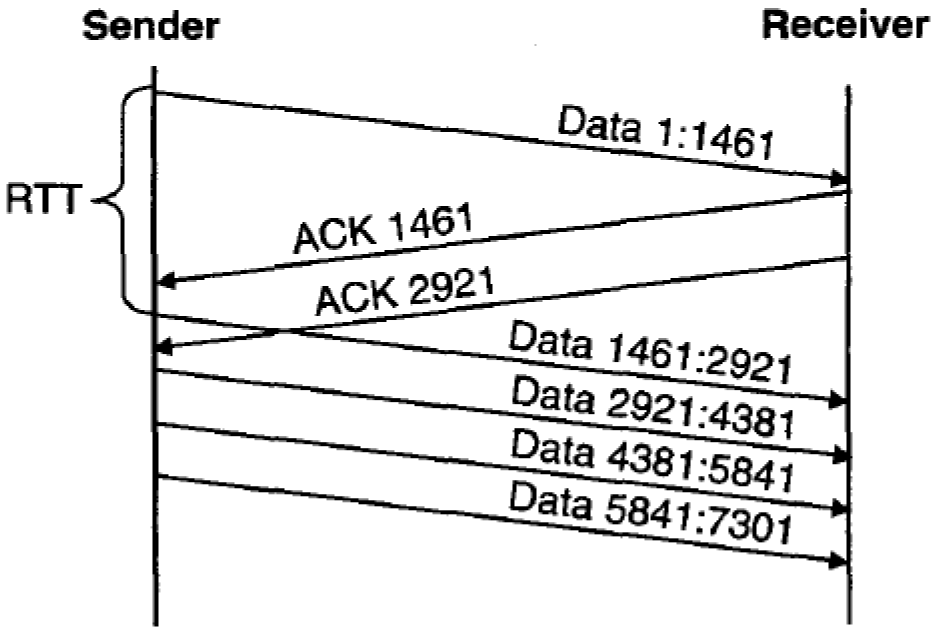


Figure 3: Sample time line for optimistic ACKing attack. The ACK for the second segment is sent before the segment itself is received, leading the receiver to grow *cwnd* more quickly than otherwise. At the end of this example, *cwnd*=3, rather than the expected value of *cwnd*=2.

Solutions – ACK division

- Modify the congestion control mechanisms to operate at byte granularity
- Guarantee that segment-level granularity is always respected
  - Perhaps simpler
  - Only increment cwnd by one MSS when a valid ACK arrives that covers the entire data segment sent.

Solutions – ACK division

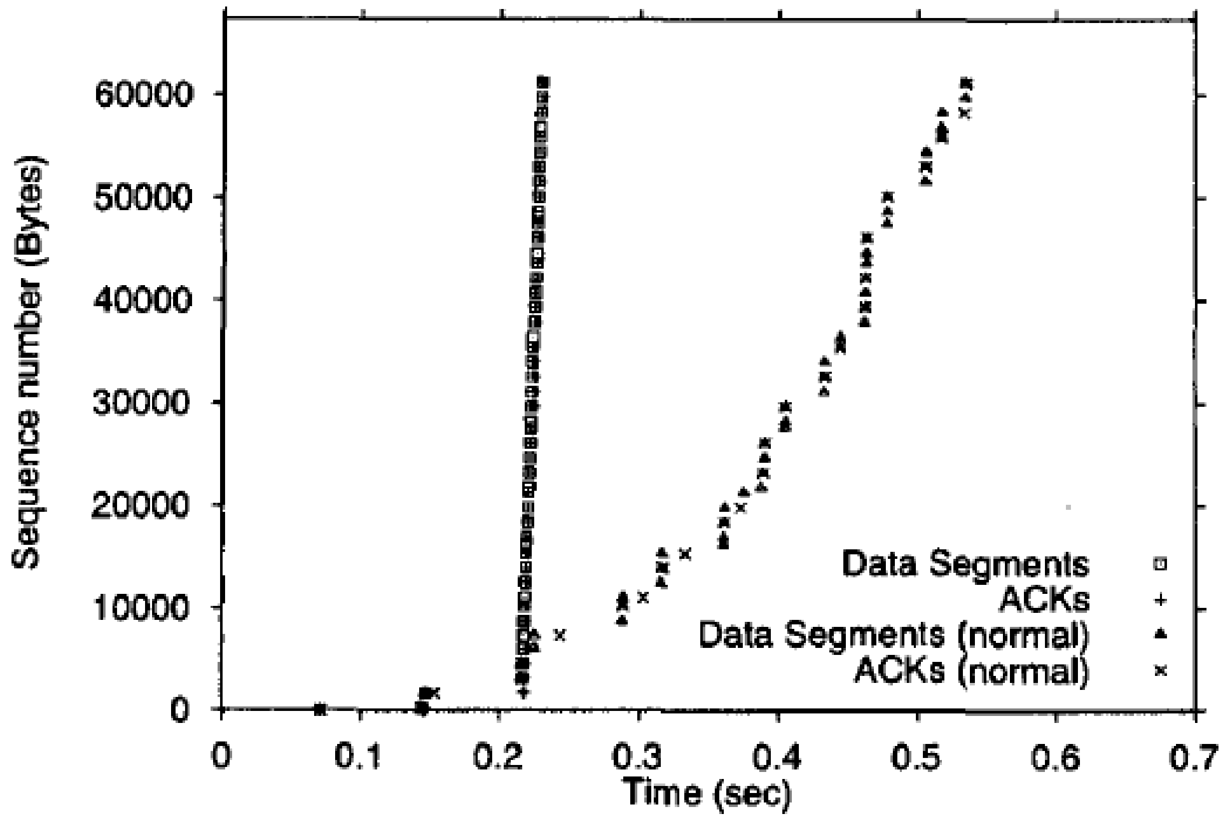


Figure 4: The TCP Daytona *ACK division* attack convinces the TCP sender to send all but the first few segments of a document in a single burst.

Solutions – DupACK Spoofing

- Traditional method: Singular Nonce
  - Two new fields introduced into the TCP packet format: Nonce and Nonce reply.
  - For each segment, the sender fills the Nonce field with a unique random number generated when the segment is sent.
- The solution requires the modification of clients and servers and the addition of a TCP field.
  - Sender maintains a count of outstanding segments sent above the missing segment
  - For each duplicate acknowledgement this count is decremented
  - When it reaches zero any additional duplicate ACKs are ignored.

Solutions – DupACK Spoofing

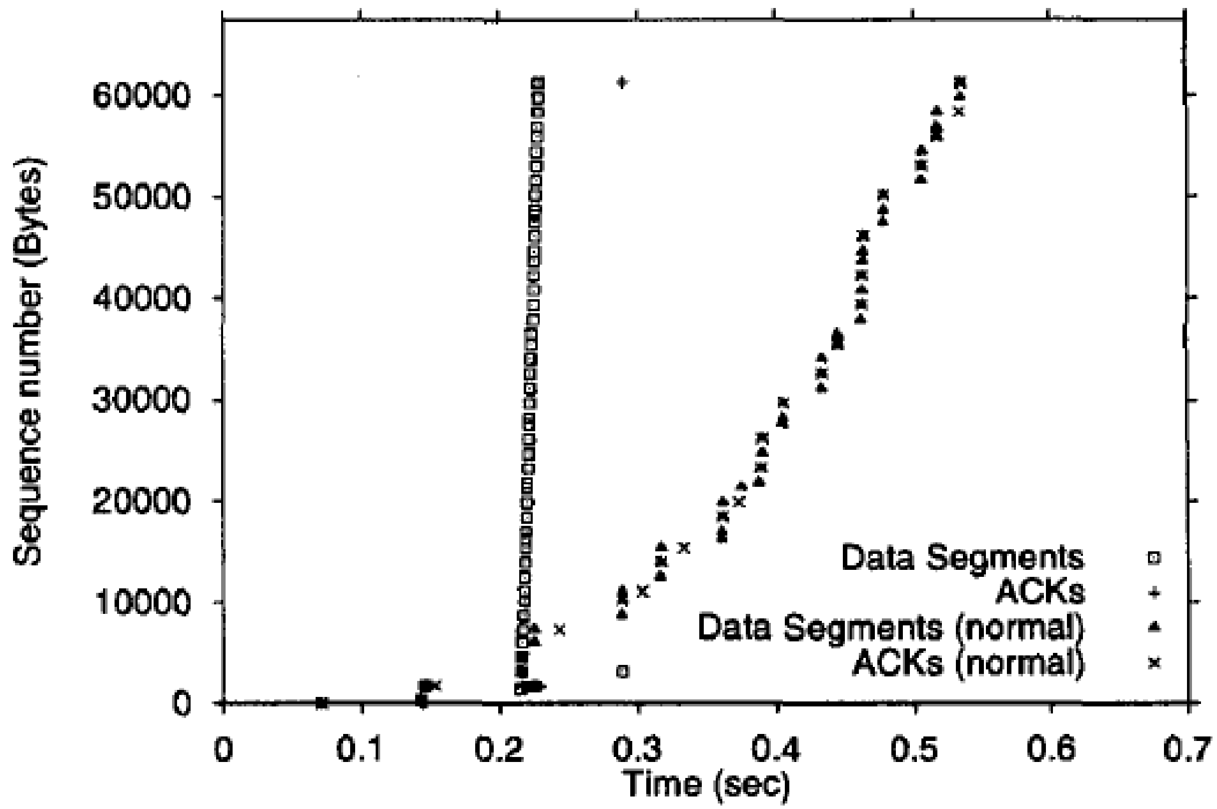


Figure 5: The TCP Daytona *DupACK spoofing* attack, like the ACK division attack, convinces the TCP sender to send all but the first few segments of a document in a single burst.

Solutions – Optimistic ACKing

- Well addressed using nonce
  - Single nonce is imperfect: does not mirror the cumulative nature of TCP.
- Cumulative Nonce

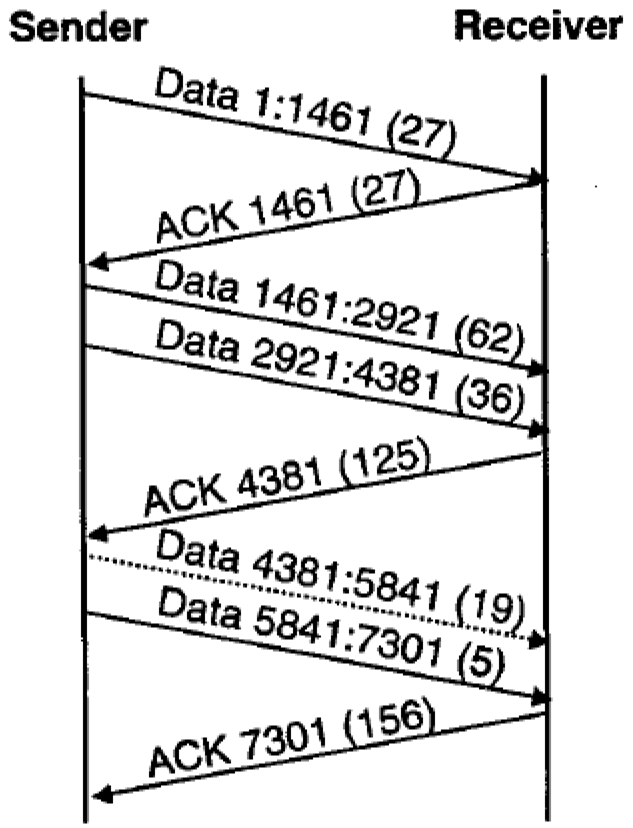


Figure 7: A time line for a transfer using a cumulative nonce. The nonce values are shown in parenthesis and it is assumed that each side starts with a nonce sum of zero. The dotted line indicates a data segment that was dropped. The final ACK, which attempts to conceal the loss of this segment, will be rejected because its cumulative nonce value is incorrect (156 instead of the expected value of 149).

Solutions – Optimistic ACKing

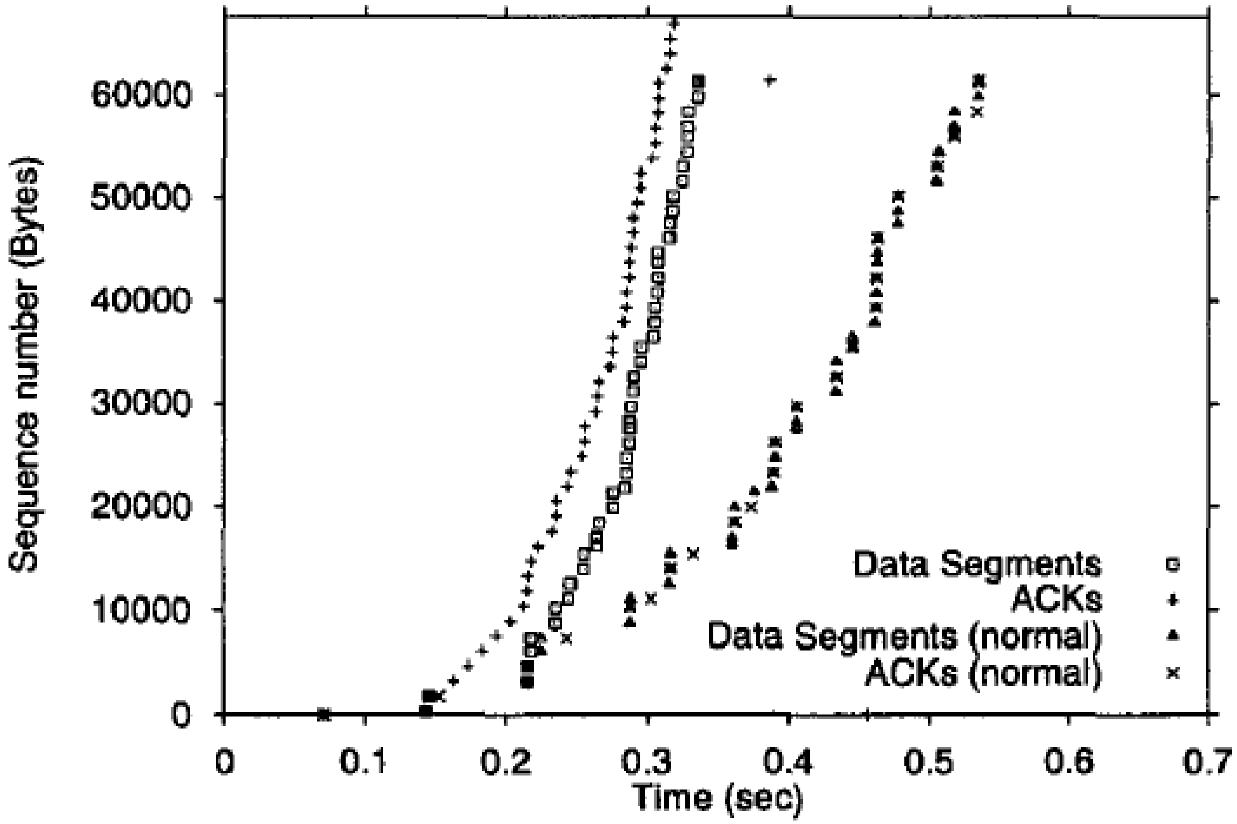
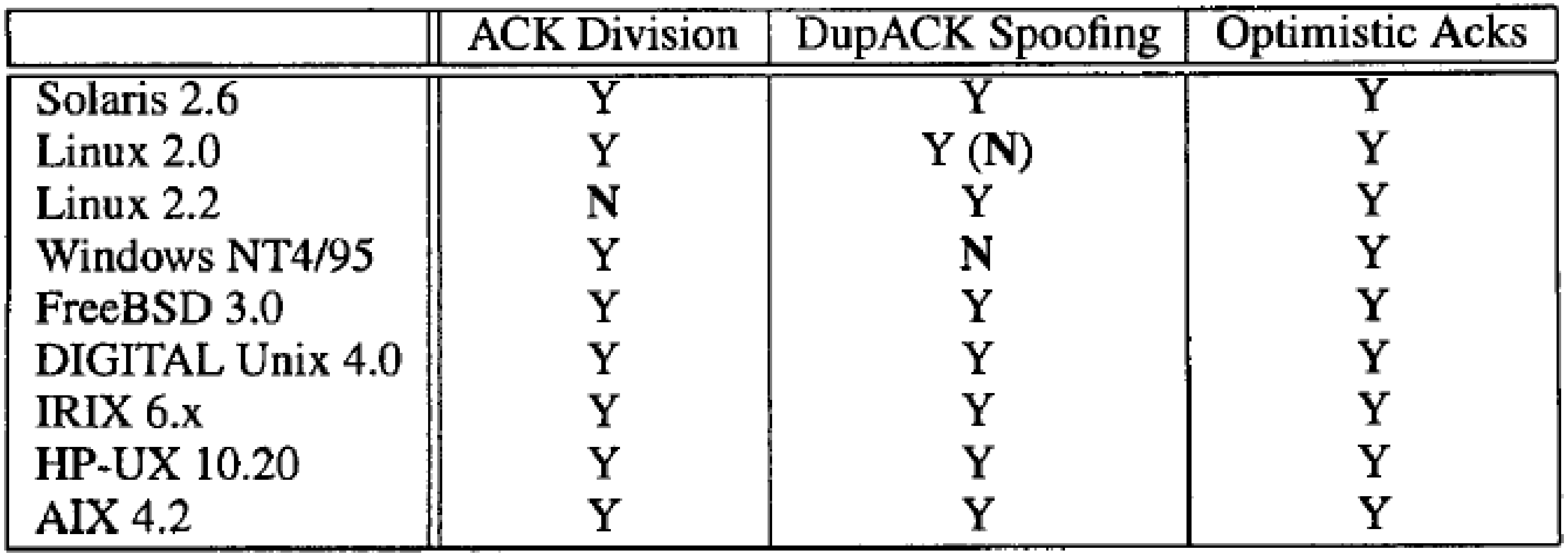


Figure 6: The TCP Daytona *optimistic ACK* attack, by sending a stream of early ACKs, convinces the TCP sender to send data much earlier than it normally would.

Different TCP Implementations



	ACK Division	DupACK Spoofing	Optimistic Acks
Solaris 2.6	Y	Y	Y
Linux 2.0	Y	Y (N)	Y
Linux 2.2	N	Y	Y
Windows NT4/95	Y	N	Y
FreeBSD 3.0	Y	Y	Y
DIGITAL Unix 4.0	Y	Y	Y
IRIX 6.x	Y	Y	Y
HP-UX 10.20	Y	Y	Y
AIX 4.2	Y	Y	Y

Outline

- Background
- TCP Review
- Congestion Avoidance and Control
- TCP Congestion Control with a Misbehaving Receiver
- Summary

Summary

- Congestion Avoidance and Control
  - Slow start
  - RTT estimation: Timeout Interval
  - Congestion avoidance
- Misbehaving Receiver
  - ACK division
  - DupACk spoofing
  - Optimistic ACKing

Thank You