Adaptive Flow Control

Introduction

- Guarantees reliable delivery of data.
- Ensures data delivered in order.
- Enforces flow control between sender and receiver.
- The idea is the sender does not overrun the receiver's buffer

DLL Sliding Window

Sliding Window Revisited

- Sender Window
- Receiver Window
- The sender and receiver slides its window

Sender Window

- *SeqNum* → Sender assigns sequence number to each frame which grows infinitely
- *SWS* →*Sender Window Size*, The size of sender window. ie The number of unacked frames the sender can transmit.(outstanding frames)
- *LAR* → Seqnum of the *Last Ack Received*
- *LFS* → Seqnum of the *Last Frame Sent*

Sender Window

- Sender Maintains LFS – LAR ≤ SWS
- Ack → LAR to right (sliding)
- A timer is associated with each frame
- The sender buffers SWS no. of frames.

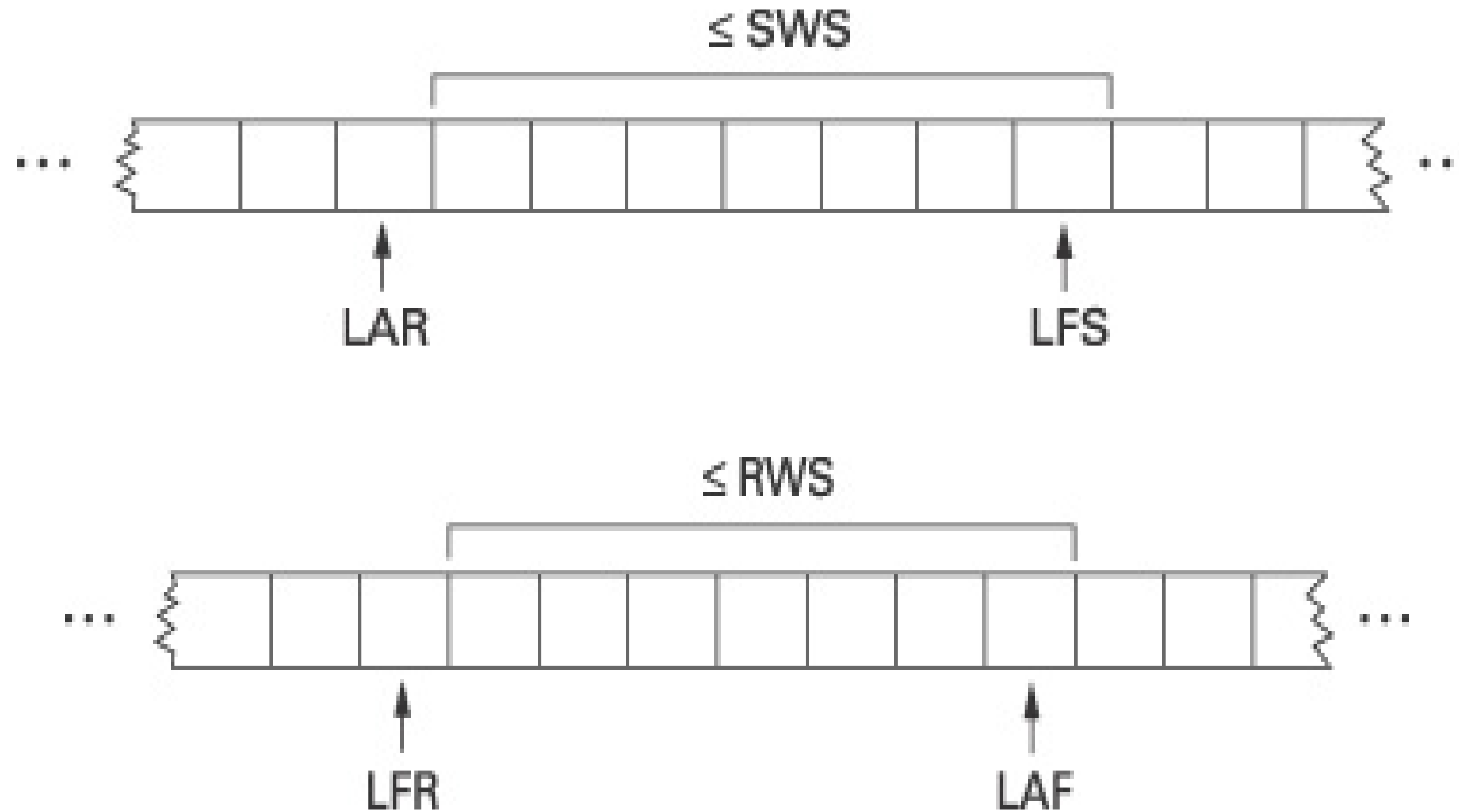
Receiver Window

- *RWS* → Receiver Window Size, The size of receiver window. ie The number of frames the receiver willing to accept.(out-of-order frames)
- *LAF* →Seqnum of the *Largest Acceptable Frame*
- *LFR* →Seqnum of the *Last Frame Received*

Receiver Window

- Receiver Maintains $LAF - LFR \leq RWS$
- $SeqNum \leq LFR$ or $SeqNum > LAF$
  - frame is outside the receiver's window, discard it
- SeqNum within the window and arrived in out of order → buffer the received frame and send a NAK to the sender for the expected frame.
- SeqNum within the window and arrived in order → slide the window
- $LFR < SeqNum \leq LAF$ →frame within the receiver's window and it is accepted.

Sender & Receiver Window



Relationship between send buffer (a) and receive buffer (b).

TCP Sliding Window

Window Size

- The big difference is the size of the sliding window
- Size at the receiver is not fixed.
- The receiver *advertises* an adjustable window size (Advertised Window field in TCP header).
- Sender is limited to having no more than Advertised Window size of unACKed data at any time.

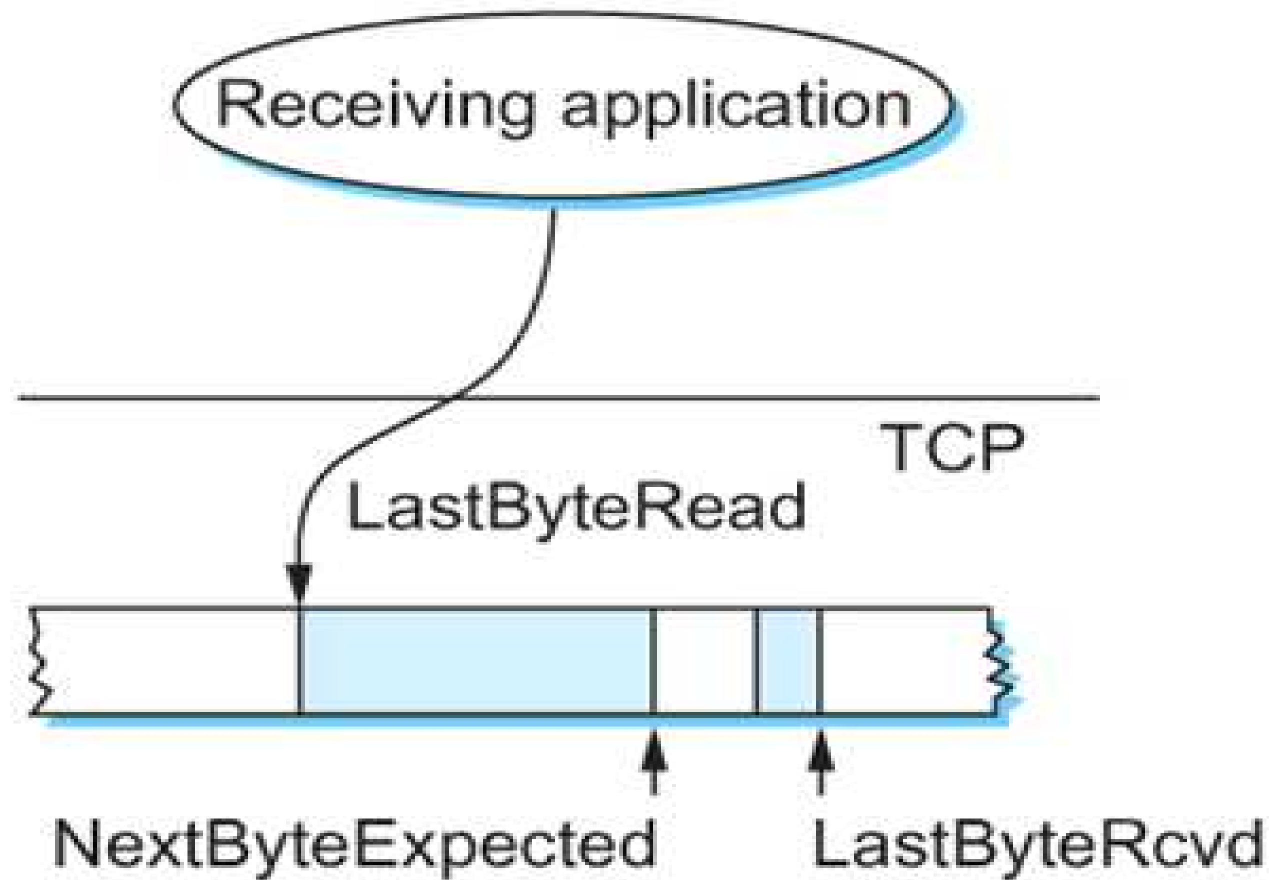
TCP Flow Control

- The discussion is similar to the previous sliding window mechanism except we add the complexity of sending and receiving *application processes* that are filling and emptying their local buffers.
- Also introduce complexity that buffers are of finite size, but not worried about where the buffers are stored.
  - MaxSendBuffer
  - MaxRcvBuffer

Receiver Window

- *LastByteRead*
- *NextByteExpected*
- *LastByteReceived*

Receiver Window



Receiver Window

- Receiver throttles sender by advertising a window size no larger than the amount it can buffer.

- On TCP receiver side:

LastByteRcvd - LastByteRead <= MaxRcvBuffer

to avoid buffer overflow!

Receiver Window

- TCP receiver advertises:

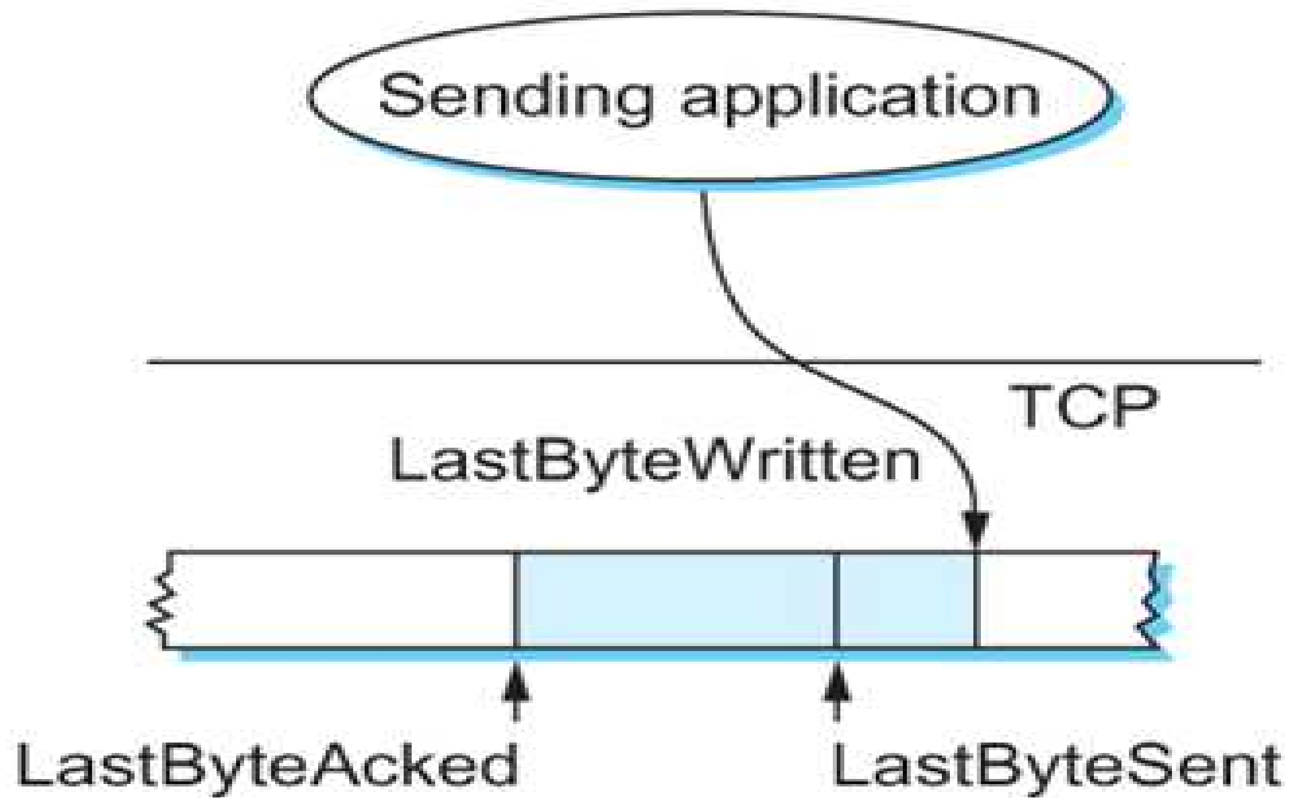
AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - LastByteRead)

- i.e., the amount of free space available in the receive buffer.

Sender Window

- *LastByteAcked*
- *LastByteSent*
- *LastByteWritten*

Sender Window



Sender Window

- TCP sender must adhere to AdvertisedWindow from the receiver such that

$$LastByteSent - LastByteAcked <= AdvertisedWindow$$

or use EffectiveWindow:

$$EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)$$

Sender Window

- Flow Control Rules

1. *EffectiveWindow > 0* for sender to send more data
2. *LastByteWritten – LastByteAcked <=MaxSendBuffer*

- *Send buffer is full!!*
- *TCP sender must **block** sender application.*

TCP Congestion Window

- CongestionWindow →a variable held by source for each connection.
- TCP is modified such that the maximum number of bytes of unacknowledged data allowed is the *minimum of* CongestionWindow and Advertised Window.
- Maximum Window (Max number of bytes unacknowledged data) → min (CongestionWindow , AdvertisedWindow)

TCP Congestion Window

- And finally, we have:

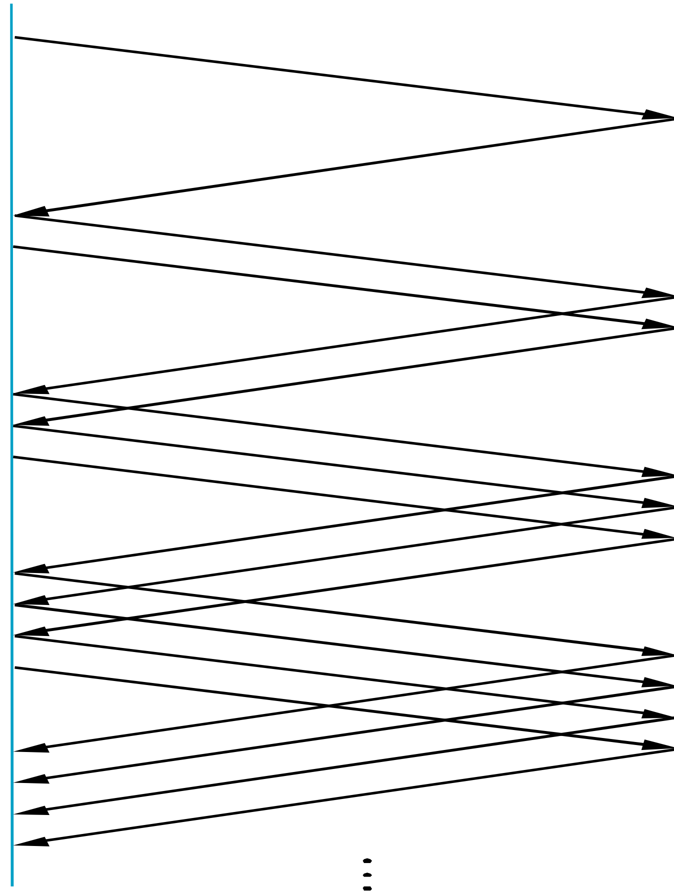
EffectiveWindow = MaxWindow – (LastByteSent – LastByteAcked)

- The TCP source receives implicit and/or explicit indications of congestion by which to reduce the size of *CongestionWindow.*

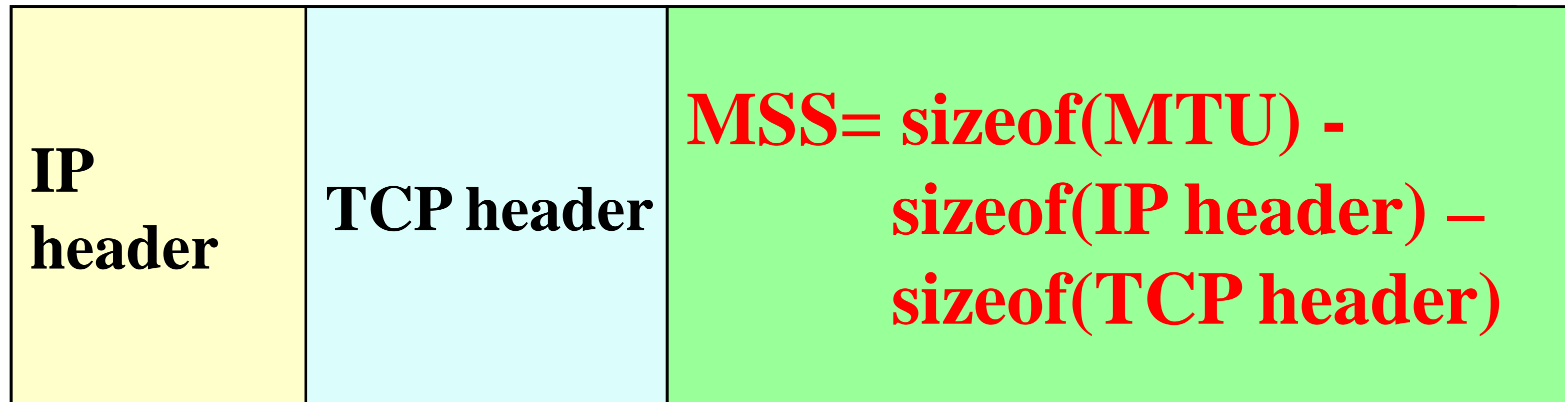
Additive Increase

- Additive Increase is a reaction to perceived available capacity.
- Linear Increase → For each "cwnd's worth" of packets sent, increase cwnd by 1 packet.
- In practice, cwnd is incremented exponentially for each arriving ACK.

Additive Increase



Maximum Segment Size

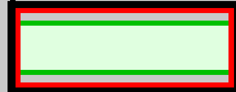


Silly Window Syndrome

- If a server with this problem is unable to process all incoming data, it requests that its clients reduce the amount of data they send at a time.
- MSS/2
- If the server continues to be unable to process all incoming data, the window becomes smaller and smaller, sometimes to the point that the data transmitted is smaller than the packet header, making data transmission extremely inefficient.
- The name of this problem is due to the window size shrinking to a "silly" value.

Client

SND.UNA = 1
SND.WND = 360
Usable = 360



SND.NXT = 1

1. Send 360-Byte Segment

Segment
Length=360
Seq Num=1

SND.UNA = 1
SND.WND = 360
Usable = 0

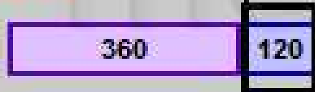
SND.NXT = 361

Acknowledgment
Ack Num = 361
Window = 120

3. Reduce Send Window to 120; Send 120-Byte Segment

Segment
Length=120
Seq Num=361

SND.WND = 120
Usable = 0
SND.UNA = 361



SND.NXT = 481

Acknowledgment
Ack Num = 481
Window = 80

3. Reduce Send Window to 80; Send 80-Byte Segment

Segment
Length=80
Seq Num=481

SND.WND = 80
Usable = 0
SND.UNA = 481

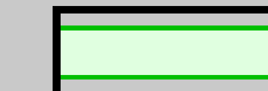
SND.NXT = 561

Acknowledgment
Ack Num = 561
Window = 67

⋮

Server

RCV.WND = 360



RCV.NXT = 1

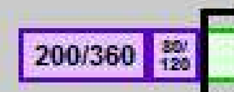
2. Receive Segment; Send *Ack*, Reduce Window To 120

RCV.WND = 120

RCV.NXT = 361

4. Receive Segment; Send *Ack*, Reduce Window To 80

RCV.WND = 80



RCV.NXT = 481

4. Receive Segment; Send *Ack*, Reduce Window To 67

RCV.WND = 80

RCV.NXT = 561

Nagle's algorithm

- Consider An application repeatedly emits data in small chunks, frequently only 1 byte in size. Since TCP packets have a 40 byte header (20 bytes for TCP, 20 bytes for IPv4),
- This results in a 41 byte packet for 1 byte of useful information, a huge overhead.
- This situation often occurs in Telnet sessions, where most key presses generate a single byte of data which is transmitted immediately.
- Worse, over slow links, many such packets can be in transit at the same time, potentially leading to congestion collapse.

Nagle's Algorithm

- If there is data to send but the window is open less than MSS, then wait some amount of time before sending the available data
- But how long to wait?
- If waiting for too long, then interactive applications like Telnet are being hurt
- If don't wait long enough, then the risk is sending a bunch of tiny packets and falling into the *silly window syndrome*
  - The solution is to introduce a timer and to transmit when the timer expires

Nagle's Algorithm

- Use a clock-based timer, for example one that fires every 100 ms
- Nagle introduced an elegant self-clocking solution
- Key Idea
  - As long as TCP has any data in send, the sender will eventually receive an ACK
  - This ACK can be treated like a timer firing, triggering the transmission of more data

Nagle's algorithm

if there is new data to send

if the window size >= MSS **and** available data is >= MSS send complete MSS segment now

Else

if

if there is unACKed data in flight

buffer the new data until an ACK arrives

else

send data immediately

end if

end if

end if

Adaptive Retransmission

Introduction

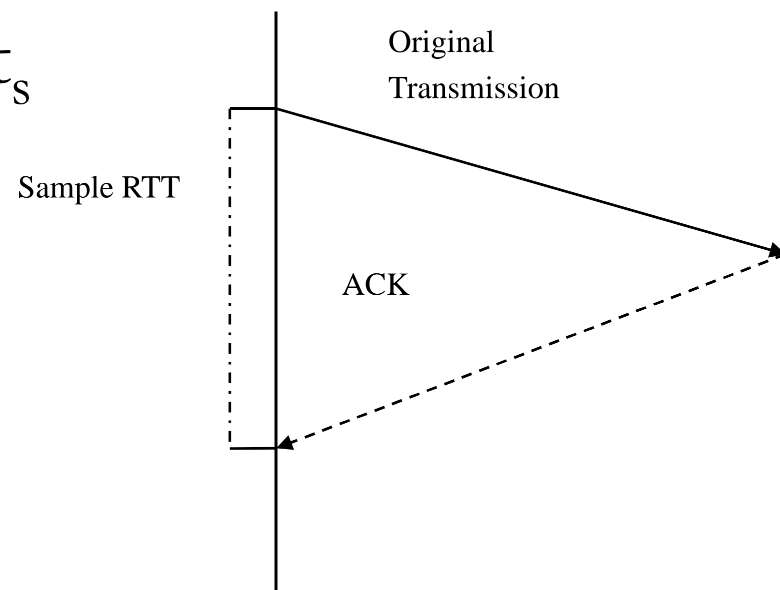
- TCP achieves reliability by retransmitting segments after a Timeout

- Choosing the value of the Timeout
  - Set time out as a function as RTT
  - If too small, retransmit unnecessarily
  - If too large, poor throughput
  - Make this adaptive, to respond to changing congestion delays in Internet

Original Algorithm

- Keep a running average of RTT and compute TimeOut as a function of this RTT.
  - Send packet and keep timestamp t_s .
  - When ACK arrives, record timestamp t_a .

$$\text{SampleRTT} = t_a - t_s$$



Original Algorithm

- Compute a weighted average:

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$

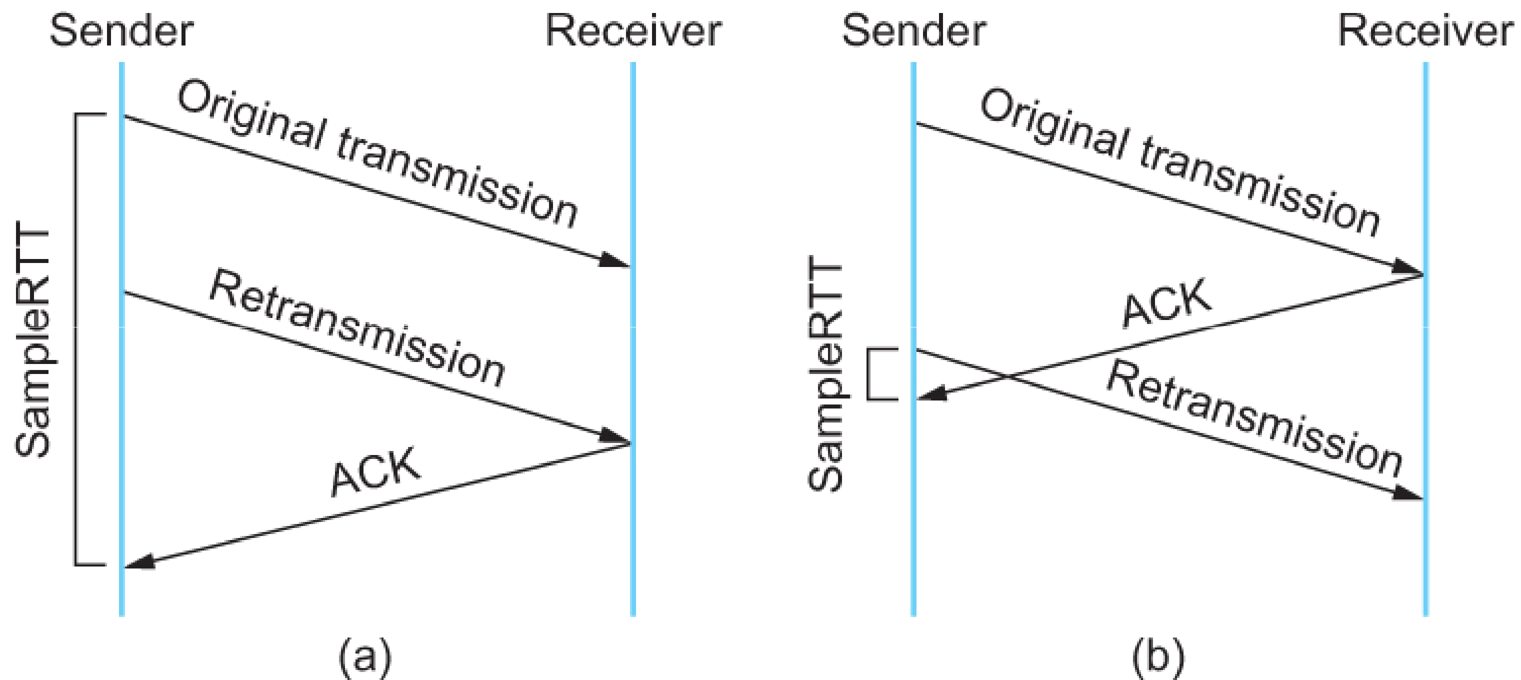
- Original TCP spec: α in range (0.8,0.9)

$$TimeOut = 2 \times EstimatedRTT$$

Original Algorithm

- Flaw in the original algorithm
  - ACK does not really acknowledge a transmission
    - It actually acknowledges the receipt of data
  - When a segment is retransmitted and then an ACK arrives at the sender
    - It is impossible to decide if this ACK should be associated with the first or the second transmission for calculating RTTs

Original Algorithm



Associating the ACK with (a) original transmission versus (b) retransmission

Karn/Partridge Algorithm

1. Do not measure *SampleRTT* when sending packet more than once.
2. For each retransmission, set *TimeOut* to *double* the last *TimeOut*.

   { Note – this is a form of exponential backoff based on the believe that the lost packet is due to congestion.}

Jacobson/Karels Algorithm

- The problem with the original algorithm is that it did not take into account the variance of SampleRTT.

$$Difference = SampleRTT - EstimatedRTT$$
$$EstimatedRTT = EstimatedRTT + (\delta \times Difference)$$
$$Deviation = \delta\ (|Difference| - Deviation)$$

where δ is a fraction between 0 and 1.

Jacobson/Karels Algorithm

- TCP computes timeout using both the mean and variance of RTT

$$TimeOut = \mu \times EstimatedRTT + \Phi \times Deviation$$

where based on experience $\mu = 1$ and $\Phi = 4$.