TCP Retransmission and Flow Control (Part II)

Presenter: Assist.Prof.Dr. Fatih ABUT

What TCP does for you?

- Stream-based in-order delivery
  - Segments are ordered according to sequence numbers
  - Only consecutive bytes are delivered
- Reliability
  - Missing segments are detected (ACK is missing) and retransmitted
- Flow control
  - Receiver is protected against overload (window based)
- Congestion control
  - Network is protected against overload (window based)
  - Protocol tries to fill available capacity
- Connection handling
  - Explicit establishment + teardown
- Full-duplex communication
  - e.g., an ACK can be a data segment at the same time (piggybacking)

TCP Retransmission Strategy

- TCP relies on positive acknowledgements
  - Retransmission on timeout
- Timer associated with each segment as it is sent
- If timer expires before acknowledgement, sender must retransmit

- TCP uses an *adaptive retransmission algorithm* because internet delays are so variable
- *Round trip time* of each connection is recomputed every time an acknowledgment arrives
- Timeout value is adjusted accordingly

Retransmit Timeout Mechanism (1)

- RTO timer value difficult to determine:
  - too high $\Rightarrow$ bad in case of msg-loss!
  - too short $\Rightarrow$ risk of false alarms!
  - General consensus: too short is worse than too long; use conservative estimate

- Calculation: measure RTT (Seg# ... ACK#)

- Original suggestion in RFC 793 (1981)
  - Exponentially Weighed Moving Average (EWMA)

  - **$SRTT_{new} = \alpha * SRTT_{old} + (1-\alpha) * RTT_{current}$**
  - **RTO = min(UBOUND, max(LBOUND, β * SRTT))**

SRTT: Smoothed Round Trip Time
α: Smoothing factor (typically 0.8-0.9)
β: Variance factor (typically 2)

Retransmit Timeout Mechanism (2)

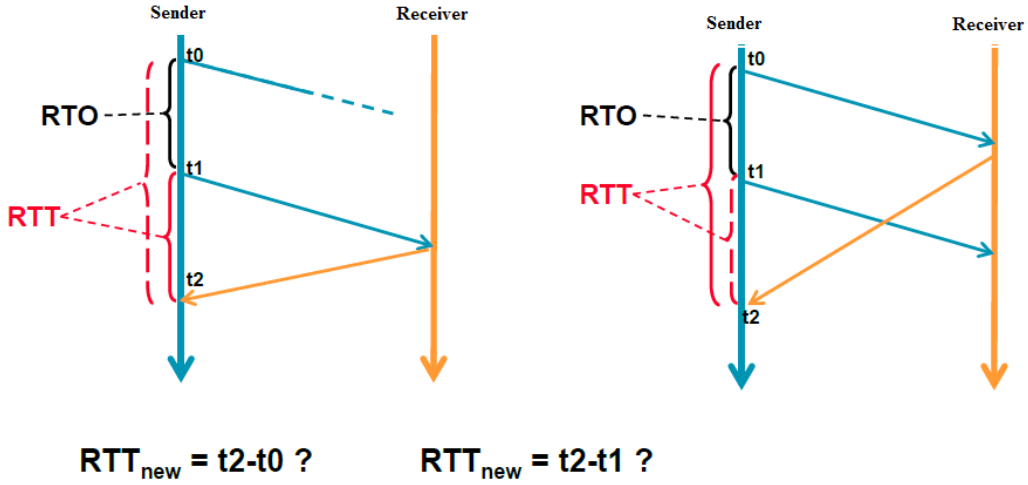
- Depending on variation, this RTO may be too small or too large; thus, final algorithm includes deviation
- Key observation:
  - Original smoothed RTT can't keep up with wide/rapid variations in RTT

- Jacobson/Karel's Retransmission Timeout (1988)

  - $\mathbf{SRTT_{new} = \alpha * SRTT_{old} + (1-\alpha) * RTT_{current}}$
  - $\mathbf{SDEV_{new} = \beta * SDEV_{old} + (1 - \beta) * [abs(RTT_{current} - SRTT_{new})]}$
  - $\mathbf{RTO_{new} = SRTT_{new} + \gamma * SDEV_{new}}$

SDEV: Smoothed deviation
β: Smoothing factor for standard deviation (typically 0.8-0.9)
γ: Adjustment factor (typically 4)

RTO calculation

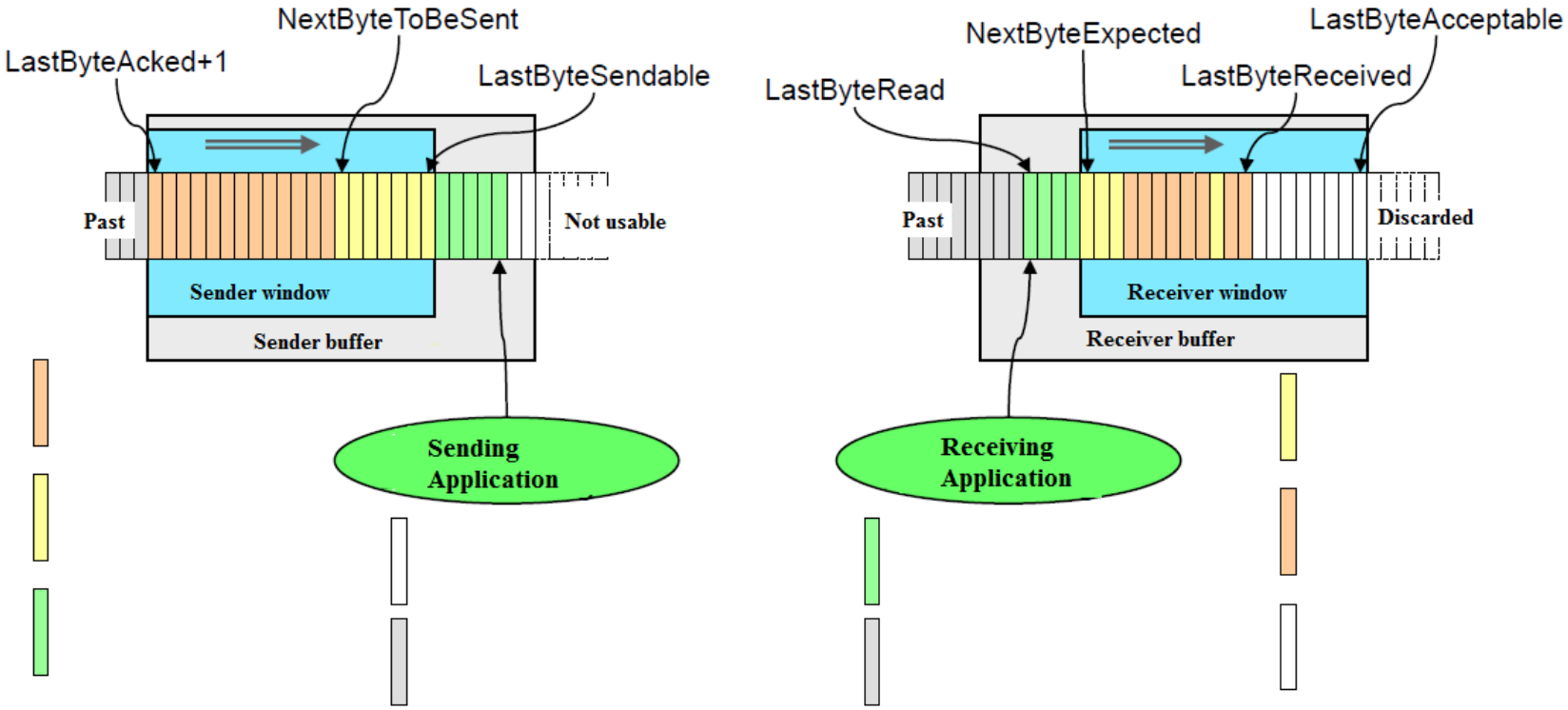


- Problem: retransmission ambiguity
  - Segment #1 sent, no ACK received → segment #1 retransmitted
  - Incoming ACK #2: cannot distinguish whether original or retransmitted segment #1 was ACKed
  - Thus, cannot reliably calculate RTO!

- Solution [Karn/Partridge]: ignore RTT values from retransmits
  - Problem: RTT calculation especially important when loss occurs; sampling theorem suggests that RTT samples should be taken more often

- Solution: Timestamps option
  - Sender writes current time into packet header (option)
  - Receiver reflects value
  - At sender, when ACK arrives, RTT = (current time) - (value carried in option)

Sliding Window Management



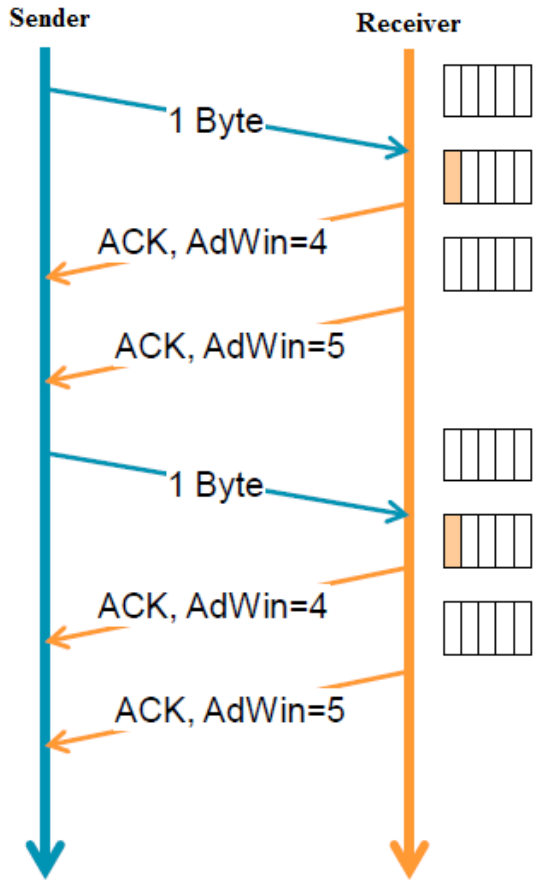
- Receiver "grants" credit (receiver window, rwnd)
  - sender restricts sent data with window

- Receiver buffer not specified
  - i.e. receiver may buffer reordered segments (i.e. with gaps)

Small Packet Problem

➢ **Sending data in very small segments**

- Syndrome created by the Sender
  - Sending application program creates data slowly (e.g. 1 byte at a time)
  - Wait and collect data to send in a larger block
  - How long should the sending TCP wait?
  - Solution: Nagle's algorithm
  - Nagle's algorithm takes into account (1) the speed of the application program that creates the data, and (2) the speed of the network that transports the data



Nagle's Algorithm (1)

- If there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data
  - If we wait too long, then we hurt interactive applications
  - If we don't wait long enough, then we risk 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 (2)

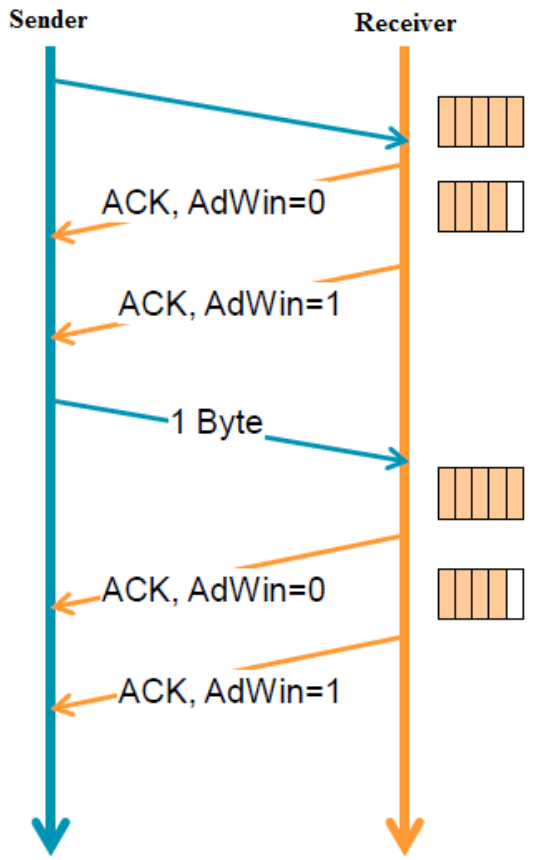
- We could use a clock-based timer, for example one that fires every 100 ms
- Nagle introduced an elegant self-clocking solution
- Key Idea
  - As long as TCP has any data in flight, the sender will eventually receive an ACK
  - This ACK can be treated like a timer firing, triggering the transmission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

Nagle's Algorithm (3)

```
When the application produces data to send
        if both the available data and the window ≥ MSS
                send a full segment
        else /* window < MSS */
                if there is unACKed data in flight
                        buffer the new data until an ACK arrives
                else
                        send all the new data now
```

Silly Window Syndrome

- Syndrome created by the Receiver
  - Receiving application program consumes data slowly (e.g. 1 byte at a time)
  - The receiving TCP announces a window size of 1 byte. The sending TCP sends only 1 byte.
  - Solution: Sending an ACK but announcing a window size of zero until there is enough space to accommodate a segment of max. size or until half of the buffer is empty



Fast Retransmit (1)

- Coarse timeouts remained a problem, and Fast retransmit was added with TCP Tahoe.
- Since the receiver responds every time a packet arrives, this implies the sender will see duplicate ACKs.
- Basic Idea: *use **duplicate ACKs** to signal lost packet.*

Fast Retransmit

Upon receipt of *three* duplicate ACKs, the TCP Sender retransmits the lost packet.

Fast Retransmit (2)

- Generally, fast retransmit eliminates about half the coarse-grain timeouts.
- This yields roughly a 20% improvement in throughput.

Fast Retransmit

Based on three duplicate ACKs