Transport Protocols
Reliability
Sliding Window Revisited

• TCP’s variant of the sliding window algorithm, which serves several purposes:
  – (1) it guarantees the reliable delivery of data,
  – (2) it ensures that data is delivered in order, and
  – (3) it enforces flow control between the sender and the receiver.
Solution: Pipelining via Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called *window*
Buffering on Sender and Receiver

- Sender needs to buffer data so that if data is lost, it can be resent
- Receiver needs to buffer data so that if data is received out of order, it can be held until all packets are received
  - Flow control
- How can we prevent sender overflowing receiver’s buffer?
  - Receiver tells sender its buffer size during connection setup
- How can we insure reliability in pipelined transmissions?
  - Go-Back-N
    - Send all N unACKed packets when a loss is signaled
    - Inefficient
  - Selective repeat
    - Only send specifically unACKed packets
    - A bit trickier to implement
Sliding Window Revisited

Sending side

- \( \text{LastByteAcked} \leq \text{LastByteSent} \)
- \( \text{LastByteSent} \leq \text{LastByteWritten} \)

buffer bytes between \( \text{LastByteAcked} \) and \( \text{LastByteWritten} \)

Receiving side

- \( \text{LastByteRead} < \text{NextByteExpected} \)
- \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)

buffer bytes between \( \text{NextByteRead} \) and \( \text{LastByteRcvd} \)
Flow Control in TCP

- Send buffer size: **MaxSendBuffer**
- Receive buffer size: **MaxRcvBuffer**

Receiving side
- \( \text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer} \)
- \( \text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{NextByteExpected} - 1) - \text{LastByteRead} \)

Sending side
- \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow} \)
- \( \text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked}) \)
- \( \text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer} \)
  - block sender if \((\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSenderBuffer}\)

- Always send ACK in response to arriving data segment
- Persist sending one byte seg. when \( \text{AdvertisedWindow} = 0 \)
  - Keep soliciting ACKs, eventually window opens up
Triggering Transmission

• How does TCP decide to transmit a segment?
  – TCP supports a byte stream abstraction
  – Application programs write bytes into streams
  – It is up to TCP to decide that it has enough bytes to send a segment

• TCP uses “self clocking”
  – Use ACKs as an implicit timer

• ACK info tells if there is enough space
Nagle’s Algorithm

• 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

When the application produces data to send

if both the available data and the window $\geq$ MSS // either at
startup or when an ACK arrives

send a full segment

else

if there is unACKed data in flight

buffer the new data until an ACK arrives

else

send all the new data now
Adaptive Retransmission

• Original Algorithm
  – Measure **SampleRTT** for each segment/ACK pair
  – Compute weighted average of RTT
    • \( \text{EstRTT} = \alpha \times \text{EstRTT} + (1 - \alpha) \times \text{SampleRTT} \)
      - \( \alpha \) between 0.8 and 0.9
  • Set timeout based on \( \text{EstRTT} \)
    • \( \text{TimeOut} = 2 \times \text{EstRTT} \)
Original Algorithm

• Problem
  – 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
Karn/Partridge Algorithm for RTO

- Two degenerate cases with timeouts and RTT measurements
  - Solution: Do not sample RTT when retransmitting
- After each retransmission, set next RTO to be double the value of the last
  - Exponential backoff is well known control theory method
  - Loss is most likely caused by congestion so be careful
Karn/Partridge Algorithm

• Karn-Partridge algorithm was an improvement over the original approach, but it does not eliminate congestion

• We need to understand how timeout is related to congestion
  – If you timeout too soon, you may unnecessarily retransmit a segment which adds load to the network
Karn/Partridge Algorithm

• Main problem with the original computation is that it does not take variance of Sample RTTs into consideration.

• If the variance among Sample RTTs is small
  – Then the Estimated RTT can be better trusted
  – There is no need to multiply this by 2 to compute the timeout
Karn/Partridge Algorithm

• On the other hand, a large variance in the samples suggest that timeout value should not be tightly coupled to the Estimated RTT

• Jacobson/Karels proposed a new scheme for TCP retransmission
Jacobson/ Karels Algorithm

- In late ’80s, Internet was suffering from *congestion collapse*
- New Calculations for average RTT – Jacobson ’88
- Variance is not considered when setting timeout value
  - If variance is small, we could set RTO = EstRTT
  - If variance is large, we may need to set RTO > 2 x EstRTT
- New algorithm calculates both variance and mean for RTT
- \( \text{Diff} = \text{sampleRTT} - \text{EstRTT} \)
- \( \text{EstRTT} = \text{EstRTT} + (d \times \text{Diff}) \)
- \( \text{Dev} = \text{Dev} + d \times (|\text{Diff}| - \text{Dev}) \)
  - Initially settings for \( \text{EstRTT} \) and \( \text{Dev} \) will be given to you
  - where \( d \) is a factor between 0 and 1
  - typical value is 0.125
• \textbf{TimeOut} = \mu \times \text{EstRTT} + \phi \times \text{Dev}
  
  – where \( \mu = 1 \) and \( \phi = 4 \)

• When variance is small, TimeOut is close to EstRTT
• When variance is large Dev dominates the calculation

• Another benefit of this mechanism is that it is very efficient to implement in code (does not require floating point)

• Notes
  – algorithm only as good as granularity of clock (500ms on Unix)
  – accurate timeout mechanism important to congestion control (later)

• These issues have been studied and dealt with in new RFC’s for RTO calculation.

• TCP RENO uses Jacobson/Karels