Networks and Distributed Systems

Lecture 15 – TCP sliding window

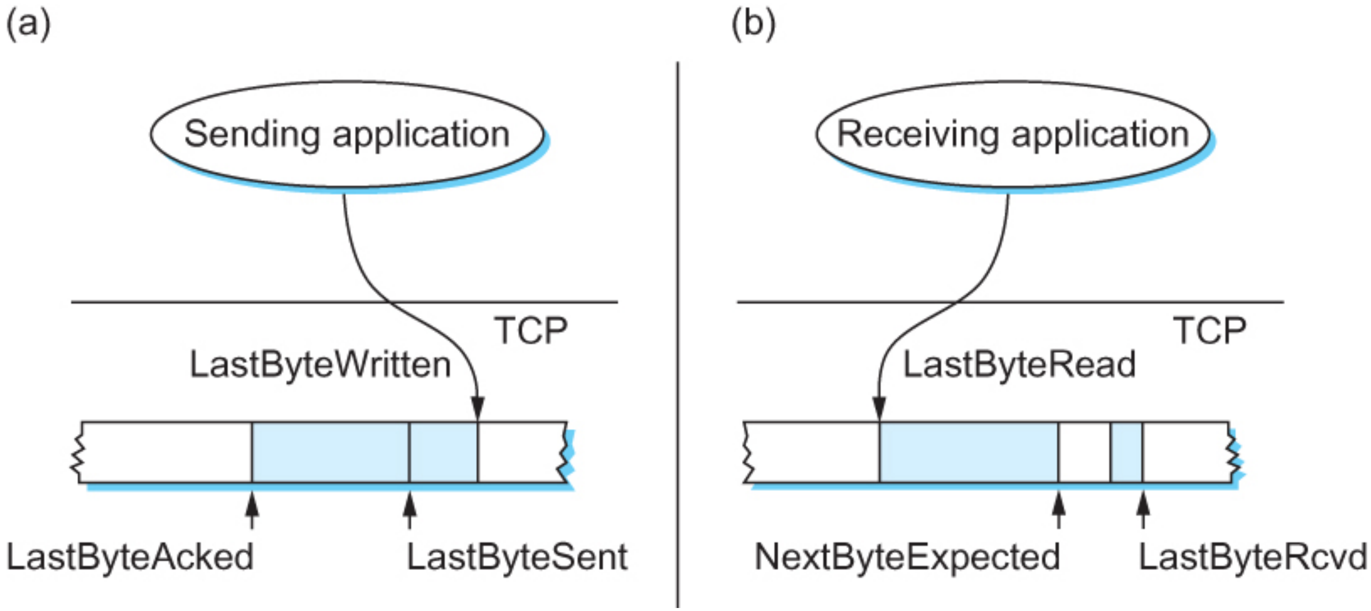
Outline

- Simple Demultiplexer (UDP)
- Reliable Byte Stream (TCP)

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.

Sliding Window Revisited



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

TCP Sliding Window

- Sending Side
  - LastByteAcked ≤ LastByteSent
  - LastByteSent ≤ LastByteWritten
- Receiving Side
  - LastByteRead < NextByteExpected
  - NextByteExpected ≤ LastByteRcvd + 1

Receiver's Advertised Window

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

TCP Flow Control

- 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!

TCP Flow Control

- TCP receiver advertises:

AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - LastByteRead)

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

TCP Flow Control

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

LastByteSent – LastByteAcked <= AdvertisedWindow

- or use EffectiveWindow:

EffectiveWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)

TCP Flow Control

- If the sending process tries to write y bytes to TCP,
- But:

  (LastByteWritten − LastByteAcked) + y > MaxSendBuffer

Then TCP blocks the sending process and does not allow it to generate more data.

TCP Flow Control

- LastByteRcvd − LastByteRead ≤ MaxRcvBuffer
- AdvertisedWindow = MaxRcvBuffer − ((NextByteExpected − 1) − LastByteRead)
- LastByteSent − LastByteAcked ≤ AdvertisedWindow
- EffectiveWindow = AdvertisedWindow − (LastByteSent − LastByteAcked)
- LastByteWritten − LastByteAcked ≤ MaxSendBuffer
- If the sending process tries to write y bytes to TCP, but (LastByteWritten − LastByteAcked) + y > MaxSendBuffer then TCP blocks the sending process and does not allow it to generate more data.

Protecting against Wraparound

- SequenceNum: 32 bits longs
- AdvertisedWindow: 16 bits long
  - TCP has satisfied the requirement of the sliding
  - window algorithm that is the sequence number
  - space be twice as big as the window size
  - $2^{32} >> 2 \times 2^{16}$

Protecting against Wraparound

- Relevance of the 32-bit sequence number space
  - The sequence number used on a given connection might wraparound
  - A byte with sequence number *x* could be sent at one time, and then at a later time a second byte with the same sequence number *x* could be sent
  - Packets cannot survive in the Internet for longer than the **MSL**
  - **MSL** is set to 120 sec
  - We need to make sure that the sequence number does not wrap around within a 120-second period of time
  - Depends on how fast data can be transmitted over the Internet

Protecting against Wraparound

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-12 (622 Mbps)	55 seconds
OC-48 (2.5 Gbps)	14 seconds

Time until 32-bit sequence number space wraps around.

Keeping the Pipe Full

- 16-bit AdvertisedWindow field must be big enough to allow the sender to keep the pipe full
- Clearly the receiver is free not to open the window as large as the AdvertisedWindow field allows
- If the receiver has enough buffer space
  - The window needs to be opened far enough to allow a full
  - delay × bandwidth product's worth of data
  - Assuming an RTT of 100 ms

Keeping the Pipe Full

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-12 (622 Mbps)	7.4 MB
OC-48 (2.5 Gbps)	29.6 MB

Required window size for 100-ms RTT.

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

Triggering Transmission

- What factors governs this decision
  - Ignore flow control: window is wide open, as would be the case when the connection starts
  - TCP has three mechanism to trigger the transmission of a segment
    - 1) TCP maintains a variable MSS and sends a segment as soon as it has collected MSS bytes from the sending process
      - MSS is usually set to the size of the largest segment TCP can send without causing local IP to fragment.
      - MSS: MTU of directly connected network – (TCP header + and IP header)
    - 2) Sending process has explicitly asked TCP to send it
      - TCP supports push operation
    - 3) When a timer fires
      - Resulting segment contains as many bytes as are currently buffered for transmission

Silly Window Syndrome

- If you think of a TCP stream as a conveyer belt with "full" containers (data segments) going in one direction and empty containers (ACKs) going in the reverse direction, then MSS-sized segments correspond to large containers and 1-byte segments correspond to very small containers.
- If the sender aggressively fills an empty container as soon as it arrives, then any small container introduced into the system remains in the system indefinitely.
- That is, it is immediately filled and emptied at each end, and never coalesced with adjacent containers to create larger containers.

Silly Window Syndrome

Silly Window Syndrome

Nagle's Algorithm

- 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
- But how long?
- If we wait too long, then we hurt interactive applications like Telnet
- 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

- 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 ≥ MSS
        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

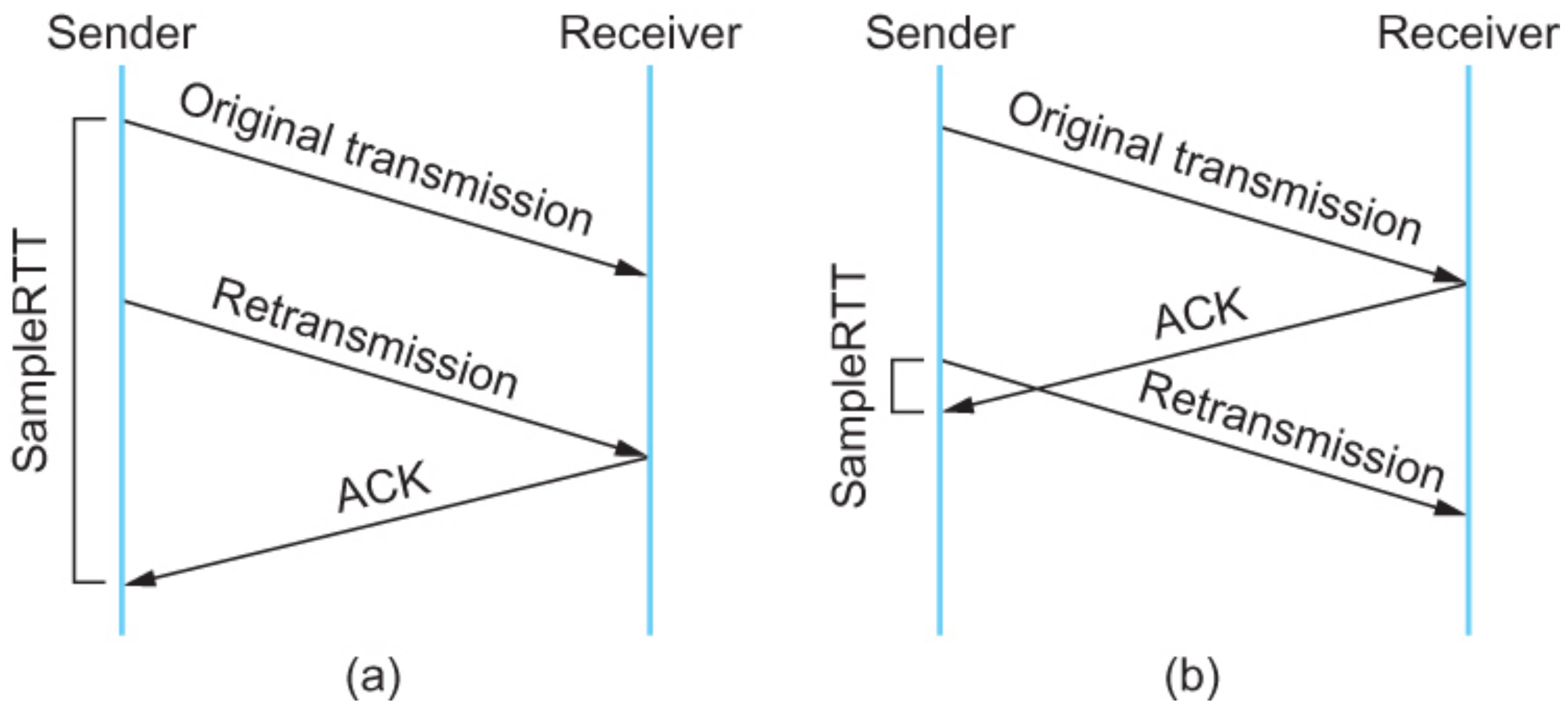
- Original Algorithm
  - Measure `SampleRTT` for each segment/ ACK pair
  - Compute weighted average of RTT
    - `EstRTT` = α x `EstRTT` + (1 - α)x `SampleRTT`
    - α between 0.8 and 0.9
  - Set timeout based on `EstRTT`
    - `TimeOut` = 2 x `EstRTT`

MORGAN KAUFMANN

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



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

Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission

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

$$SRTT = \alpha\ SRTT + (1 - \alpha)\ R$$

$$RTTVAR = \beta\ RTTVAR + (1 - \beta)\ |SRTT - R|$$

$$RTO = SRTT + 4 \times RTTVAR$$

MORGAN KAUFMANN

Jacobson/Karels Algorithm

- Difference = SampleRTT − EstimatedRTT
- EstimatedRTT = EstimatedRTT + (× Difference)
- Deviation = Deviation + (|Difference| − Deviation)
- TimeOut = μ × EstimatedRTT + × Deviation
  - where based on experience, μ is typically set to 1 and is set to 4. Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the deviation term to dominate the calculation.

Summary

- We have discussed how to convert host-to-host packet delivery service to process-to-process communication channel.
- We have discussed UDP
- We have discussed TCP