Now we address issues of flow and congestion control. We'll do this in the context of TCP since TCP's approach to congestion/flow control are the dominant ones.

First we need to give an introduction to TCP.

- Fundamental transport layer services
  - Multiplexing/Demultiplexing
  - Error detection
  - Reliable data delivery
  - Pipelining
  - Flow control
  - Congestion control

- Internet transport protocols
  - UDP
  - TCP
TCP is stream-oriented (as opposed to UDP which is a datagram protocol). You think about it as you do reading/writing to a file. The model is that the application sends data into a socket and TCP eventually sends the data. When TCP actually transmits the data is not specified in the TCP spec. TCP may send multiple segments for a single socket write call. A TCP configuration parameter is the maximum segment size (MSS) which is the largest application payload that can be in a segment.

Both sender and receiver have two buffers. Remember that each endpoint is both a sender and a receiver. (TCP connections are full-duplex.)

TCP provides only a point-to-point service — No multicast.

Segments are divided into two parts: the header and the payload. The TCP spec describes the contents and format of the header.

Outgoing sequence numbers are counts of the amount of data sent. ACKs are piggybacked on data packets. If an ACK needs to be sent and the receiver has no data to send, a segment with a null payload will be sent. (There is no separate packet type for an ACK.)

The URG and PSH bits are generally not used. The receiver's window size is the number of bytes the receiver is willing to accept in sequence beyond the current ACK number. Thus the receiver's window won't increase until the ACK number advances.
Starting sequence numbers are chosen at random to avoid a potential problem with packets with same source/destination/port information (e.g., a packet from an old connection) being delivered to an endpoint.

Outgoing sequence number is the index of the first byte in the payload.

ACKs carry the sequence number of (effectively) the last byte received in-order.

- Cumulative ACKs.
- ACKs are piggybacked with data.
- But if there is no data to send, eventually a pure ACK is generated. Note that packet still has a sequence number.

Note that there is 1 packet format (no ACK packets).

Two choices for dealing with out-of-order segments:

- Buffer (more efficient but complicates the code).
- Discard (simple — this is what we assume here).

TCP Sequence Numbers and ACKs

Telnet example

- Sequence numbers:
  - Byte stream "index" of the first byte in the segment's payload
- ACKs:
  - Sequence number of next byte expected from the other side
  - ACKs are cumulative
- How does receiver handle out-of-order segments?
  - TCP spec doesn’t say, it’s up to the implementor

TCP Reliable Data Transfer (RTD)

- TCP creates RDT service on top of IP’s unreliable service
- Pipelined segments
- Cumulative ACKs
- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - timeout events
  - duplicate ACKs
- Initially consider simplified TCP sender:
  - ignore flow control, congestion control (they determine the window size dynamically)
Consider a simplified sender. Assume data flows in only one direction and ignore flow control for now.

Retransmission in the case of 3 duplicate ACKs is called a fast retransmit (because it occurs before the timer expires).

— Why wait for 3 duplicate ACKs rather than 2? (Because out-of-order arrival is possible and we don't want to retransmit segments that are simply late.)

— More in next lecture…

Note that either the window will advance, or we retransmit segment y. — That is, both these actions can't happen at the same time. — So losses stall further sending of data.

TCP is a Go-Back-N protocol; hence the only state the sender needs to maintain is…

— the base sequence number of its current window and
— the next sequence number.

In the case of data being received from the application, the length of the payload is \( \text{Min (MSS, data passed in on call)} \).

Comment:
• SendBase – 1 = last cumulatively ACKed byte

Example:
• SendBase-1 = 71; ACK = 73, so the receiver wants 73+
• ACK > SendBase, so new data (SeqNum = 72) is ACKed
TCP is a Go-Back-N protocol and hence the only state the sender needs to maintain is...

- The base sequence number of its current window and the next sequence number.

In the case of data being received from the application, the length of the payload is 
\[
\text{Min (MSS, data passed in on call)}
\]

In the ACK processing, \(\text{sendbase}\) is the index of the bottom of the sender's window. Everything with sequence numbers less than \(\text{sendbase}\) have already been ACKed.

- If an ACK arrives with sequence number less than \(\text{sendbase}\) it must be a duplicate and out-of-order ACK (and in this case is ignored).
Bytes 92-99 are sent in the first segment.

—

Byte 100 is the next byte expected by the receiver.

Note that in the case of the lost ACK, the receiver gets a duplicate segment which it discards (its sequence number is less than the receiver's window base).

—

But the receiver still ACKs the segment (so the sender will stop resending it).

In the second example, the sender now has a larger window and can send multiple segments back-to-back.

Here the use of cumulative ACKs ameliorates the effects of a lost ACK.
RTT will vary (often dramatically) over the life of a connection. (Remember the ping experiments?)

— RTT varies as congestion varies.

— Potentially routes can change during a connection.

— Servers (or clients) can become congested.

In computing the RTT the sender will ignore retransmissions and cumulatively ACKed segments.

The EWMA calculation is also called a low-pass filter in the signal processing literature.

— It filters out high-frequency noise in the signal.

Estimating round-trip-time

\[
\text{EstimatedRTT} = (1-x)\times\text{EstimatedRTT} + x\times\text{SampleRTT}
\]

\[
\text{Timeout} = \text{EstimatedRTT} + 4\times\text{Deviation}
\]

\[
\text{Deviation} = (1-x)\times\text{Deviation} + x\times|\text{SampleRTT} - \text{EstimatedRTT}|
\]

— The estimated RTT is an exponential weighted moving average (EWMA)

  » Computes a “smooth” average

  » Influence of a given sample decreases exponentially fast

  » Typical value of \( x \) is 0.125

— Timeout is \( \text{EstimatedRTT} \) plus “safety margin”

— Large variation in \( \text{EstimatedRTT} \) results in a larger safety margin
**Fast Retransmit**

- Time-out period often relatively long:
  - long delay before resending lost packet
  - timer granularity ~ 10-200 ms
  - minimum RTO = 200-1200 ms

- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If a segment is lost, there will likely be many duplicate ACKs (one for each segment that arrives safely)

- If sender receives 3 ACKs for the same data, it assumes that the segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires
  - Why wait for 3?
Recall that we are considering only unidirectional data flow.

- The receiver's state machine here has only 1 state (wait for data).

**Case 1:** Everything is normal (and the receiver has nothing to send to the sender).

Wait up to 200 ms until an ACK is sent. Why?

- We want to avoid cluttering up the network with really small packets.
  (Leads to under-utilization of resources if switches allocate resources on a packet basis.)

- Won't this delay the sender's sending more data?
  (No! We assume the sender's window is large enough to have multiple segments in-flight.)

- Won't this lead to a premature timeout?
  (It could but only under stylized circumstances — 1-way data transfer, the last segment in a window, and an odd number of segments in the window.)

---

**TCP Flow Control**

**Window control**

- Flow control is the problem of ensuring the receiver is not overwhelmed.  
  » The receiver can become overwhelmed if the application reads too slow or the sender transmits too fast.

- The receiver’s window represents its remaining buffer capacity.

- The window advances as the application reads received data.
The size of the receiver's window changes over time as bytes are read and arrive.  

- $RcvWindow$ is just a subset of the receiver's sliding window.
- Every segment going from the receiver to the sender (e.g., ACKs), contains the value of the current receiver's window size.

Note that a tricky boundary condition occurs when the receiver's window is 0 bytes. If out-of-order data has been received, $RcvWindow$ starts with the highest numbered byte received.

- The receiver explicitly informs the sender of the amount of free buffer space in $RcvBuffer$.
- The sender keeps the amount of transmitted, unACKed data less than most recently received $RcvWindow$.

- The goal is to ensure: $LastByteRcvd - LastByteRead \leq RcvBuffer$
- Sender is sent:
  
  $$RcvWindow = RcvBuffer - (LastByteRcvd - LastByteRead)$$
In the figure, the sender’s window is considered to be full even though there is more buffer space for untransmitted packets (the application can accept more data from the application).

The sender’s window is limited by the receiver’s window. If the sender’s buffer is smaller than the receiver’s window then the connection is limited by the rate at which the sender can push out data.

Remember that the client (by definition) is the initiator of a connection and the server is the acceptor of a connection.

Why do we care about the 3-way handshake? Because it represents an important part of the overhead of a TCP connection. Which is critical for short-lived connections like many web transfers.

TCP endpoints establish a “connection” before exchanging data segments:

- client: connection initiator
  
  ```java
  Socket clientSocket = new Socket("hostname", "port number")
  ```

- server: contacted by client
  
  ```java
  Socket connectionSocket = welcomeSocket.accept()
  ```
Recall that the client is the end-system that initiates the connection.

One implication of this is the fact that the client typically will have data to send as soon as the connection is created.

If this is the case then the final ACK is piggybacked with the first data segment from client to server.

Either side can close a connection.

This example assumes that the client closes the connection.

In the timed wait state the client is waiting until it is certain that all of the connection's segments have been delivered.

The duration of the timed wait is implementation dependent but is typically 30, 60, or 120 seconds.

What if both sides close the connection at the same time?

Client sends FIN segment to server

Server receives FIN, replies with ACK

Client receives FIN, replies with ACK

Client enters “timed wait” state

Client will ACK any received FIN
Assume the starting state is **closed**.

The lifecycle starts with the client application initiating a TCP connection.

Data segments are exchanged during the **established** state.

The client application then attempts to close the connection.

These state diagrams assume no errors occur.

---

Errors add significantly to the complexity of these diagrams.