TCP Window Dynamic

![Graph showing TCP window dynamics with thresholds and timeout points.](image)
• 1982: TCP/IP “Version 3” officially split TCP and IP into two protocol layers
• 1988: the “slow-start” scheme was added as TCP-Tahoe Version
• 199?: the “fast recovery” schemes was added as TCP-Reno Version
• 1995: TCP-Vegas Version
Fast Recovery

• Time-out period often relatively long:
  – long delay before resending lost packet

• Detect lost segments via duplicate ACKs.
  – Sender often sends many segments back-to-back
  – If segment is lost, there will likely be many duplicate ACKs.

If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:

  **fast retransmit:** resend segment before timer expires
Figure 3.37 Resending a segment after triple duplicate ACK
event: ACK received, with ACK field value of y

if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
        start timer
}
else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
        resend segment with sequence number y
    }
}

a duplicate ACK for already ACKed segment

fast retransmit
**Fast retransmit algorithm:**

**event:** ACK received, with ACK field value of y

```java
if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
        start timer
}
else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
        resend segment with sequence number y
    }
}
```

- a duplicate ACK for already ACKed segment
- fast retransmit
TCP Flow Control

• receive side of TCP connection has a receive buffer:

  app process may be slow at reading from buffer

  speed-matching service: matching the send rate to the receiving app’s drain rate

  flow control
  sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP Flow control: how it works

(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
  = $\text{RcvWindow}$
  = $\text{RcvBuffer} - ([\text{LastByteRcvd} - \text{LastByteRead}])$

Rcvr advertises spare room by including value of $\text{RcvWindow}$ in segments

Sender limits unACKed data to $\text{RcvWindow}$ guarantees receive buffer doesn’t overflow
**TCP Connection Management**

**Recall:** TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)

- **client:** connection initiator

```java
Socket clientSocket = new Socket("hostname","port number");
```

- **server:** contacted by client

```java
Socket connectionSocket = welcomeSocket.accept();
```

**Three way handshake:**

- **Step 1:** client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

- **Step 2:** server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

- **Step 3:** client receives SYNACK, replies with ACK segment, which may contain data.
**Closing a connection:**

client closes socket:
```java
clientSocket.close();
```

**Step 1:** client end
system sends TCP FIN control segment to server.

**Step 2:** server
receives FIN, replies with ACK. Closes connection, sends FIN.
Step 3: client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle