Summary of Reliable Data Transfer
- Checksums help us detect errors
  - ACKs and NAKs help us deal with errors
- If ACK/NAK has errors sender retransmits
  - Sequence numbers help us deal with duplicates
- We can replace NAKs with ACKs of the last packet
  - ACK must specify the sequence number of the packet being acknowledged
- We can handle packet loss by using timers
  - Retransmit packets if no ACK is received and timer expires
- We can send out many packets and then wait for ACKs for them - pipelining vs. stop-and-wait
  - Need to handle out-of-order arrival at the receiver
- Receiver can ACK last in-order received packet (go-back-N) or each received packet (selective repeat)

TCP Header
- 32 bits
  - source port #: destination port #
  - sequence number
  - acknowledgment number
  - urgent
  - options (variable length)
  - application data (variable length)
- counting by bytes of data (not segments)
- # bytes receiver is willing to accept
- URG: urgent data (generally not used)
- ACK: ACK # valid
- PSH: push data now (generally not used)
- RST, SYN, FIN: connection establishment (setup, teardown commands)
- Internet checksum (as in UDP)

TCP Sequence Numbers and ACKs
- Sequence numbers:
  - Byte stream "number" of first byte in segment's data
- ACKs:
  - ACK number specifies the sequence number of next byte expected from other side
  - Cumulative ACK
- TCP spec doesn't specify how to handle out-of-order segments

TCP Round Trip Time and Timeout
- Q: How to set TCP timeout value?
  - Longer than RTT
  - But RTT varies
  - Too short: premature timeout
  - Unnecessary retransmissions
  - Too long: slow reaction to segment loss
- Q: How to estimate RTT?
  - SampleRTT: measured time from segment transmission until ACK receipt
  - Ignore retransmissions
  - SampleRTT will vary, want estimated RTT "smoother"
  - Average several recent measurements, not just current SampleRTT

TCP Round Trip Time and Timeout
- EstimatedRTT_new = (1-\alpha) \cdot EstimatedRTT_old + \alpha \cdot SampleRTT
  - Exponential weighted moving average
  - Influence of past sample decreases exponentially fast
  - Typical value: \alpha = 0.125

TCP: Overview
- Point-to-point:
  - One sender, one receiver
- Reliable, in-order byte stream:
  - No "message boundaries"
  - Pipelined
  - TCP congestion and flow control set window size
  - Send & receive buffers
- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-oriented:
  - Handshaking initializes sender, receiver state before data exchange
  - Flow controlled:
    - Sender will not overwhelm receiver
TCP Reliable Data Transfer
- TCP creates reliable data transfer 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 duplicate ACKs
  - Ignore flow control, congestion control

TCP Sender Events:
- Data rcvd from app:
  - Create segment with sequence number
  - Sequence number is byte-stream number of first data byte in segment
  - Start timer if not already running (think of timer as for oldest unacknowledged segment)
- Timeout:
  - Retransmit segment that caused timeout (oldest unacknowledged)
  - Restart timer
- ACK rcvd:
  - If acknowledges previously unacknowledged segments
  - Update what is known to be asked
  - Start timer if there are outstanding segments

TCP: Retransmission Scenarios
- Comment:  
  - SendBase: 1: last cumulatively acknowledged byte.
  - Examples:
    - SendBase = 72; y=73, so the rcv wants 73 and more; y > SendBase, so that new data is asked
    - Ack = y-1 or y + 1

TCP Round Trip Time and Timeout
- Setting the timeout:
  - EstimatedRTT plus “safety margin”
  - Large variation in EstimatedRTT -> longer safety margin
  - First estimate of how much SampleRTT deviates from EstimatedRTT:
    
    \[ \text{EstimatedRTT} = (1-\beta) \times \text{EstimatedRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}| \]

    (typically, \( \beta = 0.25 \))
- Then set timeout interval:
  - \( \text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \)
### TCP Retransmission Scenarios

**Good feature:** even if we lose some ACKs we sometimes don’t need to resend them since later ACKs may carry enough information.

![Diagram showing TCP Retransmission Scenarios](image)

**TCP ACK Generation** [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segments that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

### Fast Retransmit

- **Optimization**
  - 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

### Fast Retransmit Algorithm:

```java
if (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

### TCP Flow Control

- **Receiver’s side of TCP connection has a receive buffer:**
  - Flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - Application process may be slow at reading from buffer

- **Speed-matching service:** matching the send rate to the receiving application’s drain rate

- **Receiver advertises spare room by including value of RcvWindow in segments**
- **Sender limits unACKed data to RcvWindow**
- **Guarantees receive buffer doesn’t overflow**

### TCP Flow Control Diagram

- ![Diagram showing TCP Flow Control](image)

<table>
<thead>
<tr>
<th>Data from IP</th>
<th>socket</th>
<th>TCP data in buffer</th>
<th>application process</th>
</tr>
</thead>
<tbody>
<tr>
<td>RcvWindow</td>
<td></td>
<td>RcvBuffer</td>
<td>LastByteRcvd - LastByteRead</td>
</tr>
</tbody>
</table>
TCP Connection Management

- Establish "connection" before exchanging data segments
- Initialize TCP variables:
  - Sequence numbers
  - Buffers, flow control info
  - Client connection initiated
  - Socket clientSocket = new Socket("hostname", "port number");
- Server contacted by client
  - Socket connectionSocket = welcomeSocket.accept();

Three way handshake:

1. Client host sends TCP SYN segment to server
   - Specifies initial sequence number
   - No data
2. Server host receives SYN, replies with SYNACK segment
   - Server allocates buffers
   - Specifies server initial sequence number
3. Client receives SYNACK, replies with ACK segment, which may contain data

TCP Client Lifecycle

1. TIME_WAIT
   - close
   - wait 30 seconds
   - receive FIN
   - respond with ACK
2. SYN_SENT
   - send SYN
   - receive SYN & ACK
3. FIN_WAIT_1
   - send ACK
   - receive FIN
   - close connection
4. ESTABLISHED
   - server application creates a listen socket
5. LISTEN
   - receive ACK
   - send nothing
   - listen
6. CLOSED
   - client application initiates a TCP connection
   - send FIN
   - receive FIN
   - close connection

TCP Server Lifecycle

- Close connection
- ServerSocket.close();
- ServerSocket.accept();
- ServerSocket.accept();
- ServerSocket.close();