Transport Layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- 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 (exchange of control msgs) init’s sender, receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver
TCP segment structure

- 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)

Transport Layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control
TCP reliable data transfer

- 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 duplicate acks
  - Ignore flow control, congestion control
  - One way dataflow

TCP seq. #’s and ACKs

**Seq. #’s:**
- Byte stream “number” of first byte in segment’s data

**ACKs:**
- Seq # of next byte expected from other side
- Cumulative ACK

**Q:** How receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor

User types ‘C’

Host A

Seq=42, ACK=79, data = ‘C’

host ACKs receipt of echoed ‘C’, echoes back ‘C’

Seq=79, ACK=43, data = ‘C’

host ACKs receipt of echoed ‘C’

Seq=43, ACK=80

simple telnet scenario
TCP sender events:

**Data rcvd from app:**
- Create segment with seq #
- Seq # is byte-stream number of first data byte in segment
- Start timer if not already running (think of timer as for oldest unacked segment)
- Expiration interval: TimeOutInterval

**Timeout:**
- Retransmit segment that caused timeout
- Restart timer

**Ack rcvd:**
- If acknowledges previously unacked segments
  - Update what is known to be acked
  - Start timer if there are outstanding segments

TCP: reliable data transfer

- **event:** data received from application above
  - create, send segment
- **wait for event**
- **event:** timer timeout
  - retransmit segment
- **event:** ACK received, with ACK # y
  - ACK processing

Simplified sender, assuming
- One way data transfer
- No flow, congestion control
TCP sender (simplified)

```plaintext
loop (forever) {
    switch(event)
    event: data received from application above
        create TCP segment with sequence number nextseqnum
        if (timer currently not running) start timer
        pass segment to IP
        nextseqnum = nextseqnum + length(data)
    event: timer timeout
        retransmit not-yet-acknowledged segment with
        smallest sequence number
        restart timer
    event: ACK received, with ACK field value of y
        if (y > sendbase) { /* cumulative ACK of all data up to y */
            sendbase = y
            if (currently not-yet-acknowledged segments) {
                restart timer
            }
        }
} /* end of loop forever */
```

TCP retransmission scenarios

- **Host A**: Seq=100, 20 bytes data
  - **Host B**: Seq=92, 8 bytes data
  - **Lost ACK scenario**: SendBase = 100, ACK=100, lost ACK
  - **Premature timeout**: SendBase = 120, Ack=120

- **Host A**: Seq=100
  - **Host B**: Seq=92, 8 bytes data
  - **Timeout scenario**: SendBase = 120, Ack=120
  - **Loss scenario**: SendBase = 100, ACK=100
TCP retransmission scenarios (2.)

TCP round trip time and timeout

**Q:** How to set TCP timeout value?
- Longer than RTT
  - Note: RTT will vary
- 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, cumulatively ACKed segments
- **SampleRTT** will vary, want estimated RTT “smoother”
  - Use several recent measurements, not just current **SampleRTT**
TCP round trip time and timeout

EstimatedRTT = (1 - \(\alpha\)) \times \text{EstimatedRTT} + \(\alpha\) \times \text{SampleRTT}

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of \(\alpha\): 0.125

Key observation:
- At high loads round trip variance is high

Example RTT estimation

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr
TCP round trip time and timeout

Setting the timeout

- EstimatedRTT plus "safety margin"
  - Large variation in EstimatedRTT → larger safety margin
- First estimate of how much SampleRTT deviates from EstimatedRTT:
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \\
  \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]

(typically, $\beta = 0.25$)

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
Karn’s RTT Estimator

- Accounts for retransmission ambiguity
- If a segment has been retransmitted:
  - Don’t count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
  - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet
Timer Granularity

- Many TCP implementations set RTO in multiples of 200,500,1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient

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>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>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>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>Segment 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

- 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:

```c
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
    }
```

Duplicate ACK for already ACKed segment  Fast retransmit
Transport layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

TCP flow control

- Receive side of TCP connection has a receive buffer:
  - Flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - Speed-matching service: match the send rate to the receiving app’s drain rate
- App process may be slow at reading from buffer
TCP flow control: How it works

- TCP is a sliding window protocol
  - For window size $n$, can send up to $n$ bytes without receiving an acknowledgement
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - Congestion control now limits this

(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
Window flow control: Send Side

Window flow control: Receive Side
TCP persist

- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?

- TCP Persist state
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK even if it can’t store the packet

Transport layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control
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();
  ```

Connection establishment

- **Use 3-way handshake**

```
A
SYN + Seq A  SYN+ACK-A + Seq B
ACK-B

B
```
Sequence number selection

- Why not simply chose 0?
- Must avoid overlap with earlier incarnation

TCP connection: Three way handshake

**Step 1:** client end system sends TCP SYN control segment to server
  - Specifies initial seq #
  - Specifies initial window #

**Step 2:** server end system receives SYN, replies with SYNACK control segment
  - ACKs received SYN
  - Allocates buffers
  - Specifies server-> receiver initial seq. #
  - Specifies initial window #

**Step 3:** client system receives SYNACK, replies with ACK segment which may contain data
TCP connection management (2.)

Closing a connection:
client closes socket:
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.

TCP connection management (3.)

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.
Tear-down packet exchange

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIN</td>
<td>FIN-ACK</td>
</tr>
<tr>
<td>FIN-ACK</td>
<td>Data write</td>
</tr>
<tr>
<td>FIN</td>
<td>Data ack</td>
</tr>
<tr>
<td>FIN-ACK</td>
<td></td>
</tr>
</tbody>
</table>

TCP connection management (cont.)

TCP client lifecycle

- CLOSED
  - client application initiates a TCP connection
  - send SYN

- TIME_WAIT
  - wait 30 seconds
  - receive FIN, send ACK

- SYN_SENT
  - receive SYN & ACK, send ACK

- ESTABLISHED
  - client application initiates close connection
  - send FIN

- FIN_WAIT_2
  - receive ACK, send nothing

- FIN_WAIT_1
  - send FIN
TCP connection management (cont.)

TCP server lifecycle

- CLOSED:
  - receive ACK
  - send nothing

- LISTEN:
  - server application creates a listen socket
  - receive SYN
  - send SYN & ACK

- LAST_ACK:
  - send FIN

- CLOSE_WAIT:
  - receive FIN
  - send ACK

- ESTABLISHED:
  - receive ACK
  - send nothing