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
  - Connection management
  - Flow control
- Principles of congestion control
- TCP congestion control
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**

---

RFCs: 793, 1122, 1323, 2018, 2581

- **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

- **Congestion controlled:**
  - Sender will not overwhelm network
TCP segment structure

- **source port #**
- **dest port #**
- **sequence number**
- **acknowledgement number**
- **rcvr window size**
- **ptr urgent data**
- **URG: urgent data** (generally not used)
- **ACK: ACK #** valid
- **PSH: push data now** (generally not used)
- **RST, SYN, FIN:** connection estab (setup, teardown commands)
- **Internet checksum** (as in UDP)

Options (variable length)

- **application data** (variable length)

checksum

counting by bytes of data (not segments!)

# bytes rcvr willing to accept
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
  - Connection management
  - Flow control
- 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 implementer

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 the one segment that caused timeout
- Restart timer

Ack rcvd:
- If acknowledges previously unacked segments
  - Update what is known to be acked
  - Restart timer if there are outstanding segments
TCP: reliable data transfer

Simplified sender, assuming

- One way data transfer
- No flow, congestion control

Wait for event

- Event: data received from application above
  Create, send segment

- Event: timer timeout
  Retransmit segment

- Event: ACK received, with ACK # y
  ACK processing
TCP sender (simplified)

sendbase = initial_sequence number
nextseqnum = initial_sequence number

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

SendBase = 100

Host B

Seq=92, 8 bytes data

Ack=100

Seq=100, 20 bytes data

SendBase = 100

SendBase = 120

premature timeout

time

timeout

Seq=92 timeout

Seq=92, 8 bytes data

loss

X

lost ACK scenario

host A

host B

time

SendBase = 120

Seq=92 timeout

Seq=92, 8 bytes data

Seq=100, 20 bytes data

SendBase = 100

ACK=100

ACK=120

ACK=120

SendBase = 120

ACK=120

ACK=100

ACK=100
TCP retransmission scenarios (2.)

- Host A: Seq = 92, 8 bytes data, ACK = 100
- Host B: Seq = 100, 20 bytes data, ACK = 120

Cumulative ACK scenario

SendBase = 120

loss

timeout
**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 − α)* EstimatedRTT + α * SampleRTT

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of α: 0.125

- Key observation:
  - At high loads round trip variance is high
Example RTT estimation

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

<table>
<thead>
<tr>
<th>time (seconds)</th>
<th>RTT (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SampleRTT</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>Estimated RTT</td>
</tr>
</tbody>
</table>

SampleRTT

Estimated RTT
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:

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

- Accounts for retransmission ambiguity
  - If a segment has been retransmitted: Don’t count RTT sample on ACKs for this segment

- If retransmission timer expires
  - Double retransmission TimeoutInterval
  - Do not use RTT estimate to calculate TimeoutInterval until successful retransmission

- Timer restarted (not due to timeout)
  - Reuse RTT estimate
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
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:

event: ACK received, with ACK field value of y
  if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
      restart 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
Delayed ACK

- It is inefficient to send too many ACK only packets
- Why?
  - No data => >40 Bytes for 1 byte of information
- Goal:
  - Wait for additional data to piggy bag ACK on data pkt.
- Implementation
  - Try to not ACK every packet but only ever second
  - Wait for at most 200ms
  - ACK any out of order data
## 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><strong>Delayed ACK.</strong> 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 <em>duplicate ACK</em>, 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>
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
  - Connection management
  - Flow control
- 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
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:

```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.
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

[Diagram showing a sequence of packets exchanged between a sender and receiver, including FIN, FIN-ACK, Data write, and Data ack packets.]

TCP connection management (cont.)

TCP client lifecycle

- **CLOSED**: 
  - Client application initiates a TCP connection
  - send SYN

- **SYN_SENT**: 
  - receive SYN & ACK
  - send ACK

- **ESTABLISHED**: 
  - Client application initiates close connection
  - send FIN

- **FIN_WAIT_1**: 
  - receive ACK
  - send nothing

- **FIN_WAIT_2**: 
  - receive FIN
  - send ACK

- **TIME_WAIT**: 
  - wait 30 seconds
TCP connection management (cont.)

TCP server lifecycle

- **CLOSED**: server application creates a listen socket
- **LISTEN**: receive SYN send SYN & ACK
- **SYN_RCVD**: receive ACK send nothing
- **ESTABLISHED**: receive FIN send ACK
- **CLOSE_WAIT**: send FIN
- **LAST_ACK**: receive ACK send nothing
Detecting half-open connections

<table>
<thead>
<tr>
<th></th>
<th>TCP A</th>
<th>TCP B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(CRASH)</td>
<td>(send 300, receive 100)</td>
</tr>
<tr>
<td>2.</td>
<td>CLOSED</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>3.</td>
<td>SYN-SENT (\rightarrow) &lt;SEQ=400&gt;&lt;CTL=SYN&gt;</td>
<td>(??)</td>
</tr>
<tr>
<td>4.</td>
<td>(??)</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>5.</td>
<td>SYN-SENT (\rightarrow) &lt;SEQ=100&gt;&lt;CTL=RST&gt;</td>
<td>(Abort!!)</td>
</tr>
<tr>
<td>6.</td>
<td>SYN-SENT</td>
<td>CLOSED</td>
</tr>
<tr>
<td>7.</td>
<td>SYN-SENT (\rightarrow) &lt;SEQ=400&gt;&lt;CTL=SYN&gt;</td>
<td></td>
</tr>
</tbody>
</table>
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
  - Connection management
  - Flow control
- Principles of congestion control
- TCP congestion control
TCP flow control

- Receive side of TCP connection has a receive buffer:
  - Speed-matching service: match 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
  - App process may be slow at reading from buffer
TCP flow control: How it works

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

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

- Rcvr advertises spare room by including value of \textit{RcvWindow} in segments

- Sender limits unACKed data to \textit{RcvWindow}
  
  - Guarantees receive buffer doesn’t overflow
TCP flow control: How it works (2.)

- 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
Window flow control: Sender side

- Sent and acked
- Sent but not acked
- Not yet sent
- Next to be sent
Window flow control: Receiver side

Receive buffer

Acked but not delivered to user

Not yet acked

window
TCP persist

What happens if window is 0?
- Receiver updates window (i.e., sends ACK with new window size) 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
Observed TCP problems

- Too many small packets
  - Silly window syndrome
  - Nagel’s algorithm
- Initial sequence number selection
- Amount of state maintained
Silly window syndrome

Problem: (Clark, 1982)
- If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets

Solution
- Receiver must not advertise small window increases
- Increase window by \( \min(MSS, \text{RecvBuffer}/2) \)
Nagel’s algorithm

- Small packet problem:
  - Don’t want to send a 41 byte packet for each keystroke
  - How long to wait for more data?

- Solution:
  - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged
Why is selecting ISN important?

- Suppose machine X selects ISN based on predictable sequence
- Fred has .rhosts to allow login to X from Y
- Evil Ed attacks
  - Disables host Y – denial of service attack
  - Make a bunch of connections to host X
  - Determine ISN pattern and guess next ISN
  - Fake pkt1: [<src Y><dst X>, guessed ISN]
  - Fake pkt2: desired command
Time Wait issues

- Web servers not clients close connection first
  - Established $\rightarrow$ Fin-Waits $\rightarrow$ Time-Wait $\rightarrow$ Closed
  - Why would this be a problem?

- Time-Wait state lasts for $2 \times MSL$
  - $MSL$ is should be $120$ seconds (is often $60$s)
  - Servers often have order of magnitude more connections in Time-Wait
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