Transport Layer Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
  - no one to many multicasts
- connection-oriented:
  - processes must handshake before sending data
  - three-way handshake: (exchange of control msgs) initializes sender, receiver state before data exchange
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers:
  - set-aside during the 3-way handshaking

- full duplex data:
  - bi-directional data flow in same connection at the same time
- flow controlled:
  - sender will not overwhelm receiver

Transport Layer 3-1
Transport Layer 3-2
TCP: Overview - cont

- **Maximum Segment Size (MSS):**
  - Defined as the maximum amount of application-layer data in the TCP segment.
  - TCP grabs data in chunks from the send buffer where the maximum chunk size is called MSS. TCP segment contains TCP header and MSS.
  - MSS is set by determining the largest link layer frame (Maximum Transmission Unit or MTU) that can be sent by the local host.
  - MSS is set so that an MSS put into an IP datagram will fit into a single link layer frame. Common values of MTU is 1460 bytes, 536 bytes and 512 bytes.

- **TCP sequence #s:**
  - Both sides randomly choose initial seq #s (other than 0) to prevent receiving segments of older connections that were using the same ports.
  - TCP views data as unordered structured stream of bytes so seq #s are over the stream of bytes.
  - File size of 500,000 bytes and MSS=1,000 bytes, segment seq #s are: 0, 1000, 2000, etc.

- **TCP acknowledgement #s:**
  - Uses cumulative acks: TCP only acks bytes up to the first missing byte in the stream. TCP RFCs do not address how to handle out-of-order segments.
  - ACK # field has the next byte offset that the sender or receiver is expecting.
**Seq Numbers and Ack Numbers**

- Suppose a data stream of size 500,000 bytes, MSS is 1,000 bytes; the first byte of the data stream is numbered zero.
  - Seq number of the segments:
    - 1st seg: 0; 2nd seg: 1000; 3rd seg: 2000, ...

- Ack number:
  - Assume host A is sending seg to host B. Because TCP is full-duplex, A may be receiving data from B simultaneously.
  - Ack number that host B puts in its seg is the seq number of the next byte B is expecting from A
    - B has received all bytes numbered 0 through 535 from A. If B is about to send a segment to host A. The ack number in its segment should 536

**TCP seq. #’s and ACKs - Telnet example**

- Telnet uses "echo back" to ensure characters seen by user already been received and processed at server.
- Assume starting seq #s are 42 and 79 for client and server respectively.
- After connection is established, client is waiting for byte 79 and server for byte 42.

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

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

![Diagram of Telnet example](image-url)
**TCP Round Trip Time and Timeout**

Q: how to set TCP timeout value? (timer management)
- based on RTT
- 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 (handed to IP) until ACK receipt
  - ignore retransmissions
- SampleRTT will vary from segment to segment, want estimated RTT
  - "smoother"
    - average several recent measurements, not just current SampleRTT
- TCP maintains an average called EstimatedRTT to use it to calculate the timeout value

**TCP Round Trip Time (RTT) and Timeout**

\[
\text{EstimatedRTT} = (1 - \alpha) \times \text{priorEstimatedRTT} + \alpha \times \text{currentSampleRTT}
\]

- Exponential Weighted Moving Average (EWMA)
- Puts more weight on recent samples rather than old ones
- Influence of past sample decreases exponentially fast
- Typical value: \(\alpha = 0.125\)
- Formula becomes:
  \[
  \text{EstimatedRTT} = 0.875 \times \text{priorEstimatedRTT} + 0.125 \times \text{currentSampleRTT}
  \]

Why TCP ignores retransmissions when calculating SampleRTT:
Suppose source sends packet P1, the timer for P1 expires, and the source then sends P2, a new copy of the same packet. Further suppose the source measures SampleRTT for P2 (the retransmitted packet) and that shortly after transmitting P2 an acknowledgment for P1 arrives. The source will mistakenly take this acknowledgment as an acknowledgment for P2 and calculate an incorrect value of SampleRTT.
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:

$$\text{DevRTT} = (1-\beta)\text{DevRTT} + \beta|\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4\text{DevRTT}$$
TCP: conn-oriented transport

- segment structure
- RTT Estimation and Timeout
- reliable data transfer
- flow control
- connection management

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer as multiple timers require considerable overhead

- 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 seq #
  - seq # is byte-stream number of first data byte in segment
  - start timer if not already running for some other segment (think of timer as for oldest unacknowledged segment)

- **expiration interval:**
  - TimeOutInterval

- **timeout:**
  - retransmit segment that caused timeout
  - restart timer

- **Ack rcvd:**
  - a valid ACK field (cumulative ACK) acknowledges previously unacknowledged segments:
    - update expected ACK #
    - restart timer if there are currently unacknowledged segments

---

**TCP sender (simplified)**

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

```c
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
    start timer
    event: ACK received, with ACK field value of y
    if (y > SendBase) {
      SendBase = y
      if (there are currently not-yet-acknowledged segments)
        start timer
    }
  }

} /* end of loop forever */
```

Comment:

- SendBase-1: last cumulatively ack'ed byte

Example:

- SendBase-1 = 71;
  - y = 73, so the rcvr wants 73+;
  - y > SendBase, so that new data is acked
TCP: retransmission scenarios

**Host A**
Seq=92, 8 bytes data
ACK=100

**Host B**
Seq=92, 8 bytes data
ACK=100

**SendBase** = 100

**Cumulative ACK scenario**

- ** Lost ACK scenario **

TCP retransmission scenarios (more)

- Doubling the timeout value technique is used in TCP implementations. The timeout value is doubled for every retransmission since the timeout could have occurred because the network is congested. (the intervals grow exponentially after each retransmission and reset after either of the two other events)
**TCP ACK generation policy** [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 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>

*leaves buffering of out-of-order segments open*

**Fast Retransmit**

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Dup Ack is an ack that reacknowledges the receipt of an acknowledged segment
  - 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 last ACKed segment was lost:
  - sender performs fast retransmit: resend segment before that segment’s timer expires
  - algorithm comes as a result of 15 years TCP experience!
Fast retransmit algorithm:

event: ACK received, with ACK field value of $y$
  if ($y > \text{SendBase}$) {
    \text{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 $y$
    }
  }

Is TCP a GBN or SR protocol?

- TCP can buffer out-of-order segments (like SR).
- TCP has a proposed RFC called selective acknowledgement to selectively acknowledge out-of-order segments and save on retransmissions (like SR).
- TCP sender need only maintain smallest seq # of a transmitted but unacknowledged byte and the seq # of next byte to be sent (like GBN).
- TCP is hybrid between GBN and SR.
TCP: conn-oriented transport

- segment structure
- RTT Estimation and Timeout
- reliable data transfer
- flow control
- connection management

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: matching 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

(Suppose TCP receiver discards out-of-order segments)
- sender maintains variable called receive window
- spare room in buffer = RcvWindow = RcvBuffer - (LastByteRcvd - LastByteRead)
- TCP is not allowed to overflow the allocated buffer (LastByteRcvd - LastByteRead <= RcvBuffer)

- Rcvr advertises spare room by including value of RcvWindow in segments
- RcvWindow = RcvBuffer at the start of transmission
- Sender limits unACKed data to RcvWindow
  - sender keeps track of UnAcked data size = (LastByteSent - LastByteAcked)
  - UnAcked data size <= RcvWindow
- When Receiver RcvWindow = 0, Sender does not block but rather sends 1 byte segments that are acknowledged by receiver until RcvWindow becomes bigger.

TCP: conn-oriented transport

- segment structure
- RTT Estimation and Timeout
- reliable data transfer
- flow control
- connection management
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 (SYN bit=1) to server
- specifies initial seq # (client_isn)
- no data

**Step 2:** server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. # (server_isn), with ACK # = client_isn+1

**Step 3:** client receives SYNACK, replies with ACK # = server_isn+1, which may contain data
TCP Connection Management - disconnecting

Closing a connection:

client closes socket:

clientSocket.close();

Step 1: client end system sends TCP FIN control segment (FIN bit=1) to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN=1.

TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs where typical wait is 30 sec. All resources and ports are released.

Step 4: server receives ACK. Connection closed.
TCP Connection Management (cont)

TCP Summary

- TCP Properties:
  - point to point, connection-oriented, full-duplex, reliable
- TCP Segment Structure
- How TCP sequence and acknowledgement #s are assigned
- How does TCP measure the timeout value needed for retransmissions using EstimatedRTT and DevRTT
- TCP retransmission scenarios, ACK generation and fast retransmit
- How does TCP Flow Control work
- TCP Connection Management: 3-segments exchanged to connect and 4-segments exchanged to disconnect