Transport Layer

- this layer supports two protocols: UDP and TCP.

- **UDP (User Datagram Protocol):**
  supports connectionless communication within applications

- **TCP (Transmission Control Protocol):**
  supports connection communication within applications
Appl. Servers on Top of UDP 38

- appl. server aps is deployed in well-known host sh on top of well-known low-numbered UDP port i

- aps is always running in sh

- initially aps allocates UDP port i and any UDP socket ss and binds ss to i and adds entry (port = i, socket = ss) to UDP socket table in sh

- when shrcvs a UDP segment whose src is (ch, j) and whose dst is (sh, i), sh checks its socket table and adds rcvd segment to in-bff of ss

- later, aps rcvs the rcvd segment, prepares a reply whose src is (sh, i) and whose dst is (ch, j), and sends the reply as a UDP segment to (ch, j)
Appl. Clients on Top of UDP

- appl. client apc is deployed in any client host ch on top of any high-numbered UDP port j

- apc runs, only when needed, in any ch

- when apc runs, it allocates any UDP port j and any UDP socket cs, binds cs to j, and adds entry (port=j, socket=cs) to UDP socket table in ch

- now apc can send a UDP segment to the appl. server aps by making the src of this segment (ch,j) and its dst (sh,i)

- later, apc rcvs a reply to its sent segment from the in-bff of socket cs and removes the entry (port=j, socket=cs) from the socket table of ch.
UDP Segments

1. Each UDP segment has five fields:
   
   ```
   UDP
   { headers }
   ```
   
   - Source UDP port number
   - Destination UDP port number
   - Length of segment
   - Checksum of segment
   - Application message

2. Length of UDP segment is number of bytes in segment (UDP header + appl. msg)

3. Checksum of UDP segment is used to detect (most) end-to-end corruptions in the segment
Objectives of UDP

- addressing of applications running on top of UDP

- end-to-end detection of (most) corruptions in UDP segments

- "best effort" data transfer
Objectives of TCP

- addressing of applications running on top of TCP
- end-to-end detection of (most) corruptions in TCP segments
- connection management (i.e. establishment, removal and reset)
- control of seq's and ack's in TCP segments
- reliable data transfer
- flow control
- congestion control
TCP Connection Establishment 43

- **Server sh:**
  
  add entry \((src = \text{any}, dst = (sh, i), ss)\) to TCP socket table of \(sh\). \((ss\) is the welcome socket\)

- **Client ch1:**
  
  add entry \((src = (sh, i), dst = (ch1, j1), cs1)\) to TCP socket table of \(ch1\). Send SYN seg \((src = (ch1, j1), dst = (sh, i))\)

- **Server sh:**
  
  rcv SYN seg and add it to in-bff of socket \(ss\) in \(sh\). Add entry \((src = (ch1, j1), dst = (sh, i), ss1)\) to TCP socket table of \(sh\). Send SYN-ACK seg \((src = (sh, i), dst = (ch1, j1))\)

- **Client ch1:**
  
  rcv SYN-ACK seg and add it to in-bff of socket \(cs1\) in \(ch1\). Send ACK seg \((src = (ch1, j1), dst = (sh, i))\)

- **Server sh:**
  
  rcv ACK seg and add it to in-bff of socket \(ss1\) in \(sh\). Send DATA seg \((src = (sh, i), dst = (ch1, j1))\)
client and server wait 30 seconds of no activity before they release port j1 and sockets cs1 and ss1 and update their TCP socket tables
if server sh rcvs a TCP seg
(src = (ch1, j1), dst = (sh, i)) and does not
find an entry
(src = (ch1, j1), dst = (sh, i), ss1)
in the TCP socket table of sh

then sh sends back a RST seg
(src = (sh, i), dst = (ch1, j1))
for simplicity, consider TCP connection between two sides: called src and dst. Side src sends Data seg's to dst and side dst sends Ack seg's to src

each Data seg has, say 100, data bytes, and the data bytes in the Data seg's constitute a byte stream. Consecutive bytes in this stream have consecutive seq#'s. The first seq# in the stream, say 5, is selected at random by src

each Data seg has seq#, which is the seq# of the first data byte in the seg. Thus, the first two Data seg's sent from src to dst are:

dst ← (Data, 5), (Data, 105), ... src
Ack#'s

- each Ack seg sent from dst to src has an ack# which is the seq# of the next data byte that dst expects to rcv from src

- example:

```
+------------------+--------+
|                   |        |
| dst               | src    |
|                   |        |
| Data, 5           |        |
|                   |        |
| Ack, 105          |        |
|                   |        |
| Data, 105         |        |
|                   |        |
| Ack, 205          |        |
```

seq# of the first Data seg is 5
Ack# of the first Ack seg is 105
Examples of Reliable Data Transfer 48

- **Example 1:**
  - Data, 5
  - Data, 5
  - Ack, 105

- **Example 2:**
  - Data, 5
  - Ack, 105
  - No Ack Timeout
  - Repeat Seg
  - Data, 5
  - Ack, 105
  - No Ack Timeout
Sliding Window Protocol

- $x, x+1, \ldots$ seq#'s of data bytes in stream
- $na$: seq# of next data byte to be acked by dst
- $ns$: """""""" sent by src
- $w$: max number of data bytes that have been sent by src and not yet ackd by dst
- $nd$: seq# of next data byte to be read by appli.
- $nr$: """""""" rcvd by dst

![Diagram]

- Next, we discuss how src keeps track of $w$
Flow Control

- Side dst has circular bff to store rcvd data bytes from src until these bytes are rcvd by the application:

- Side dst keeps track of the rcv window wr as follows:
  \[ wr := (bff \text{ size}) - (nr - nd) \text{ Bytes} \]
  Side dst includes the latest wr in every Ack seg that dst sends to src

- when side src rcvs an Ack seg with wr from side dst, then src computes w as follows:
  \[ w := \max (wr, 1) \text{ Bytes} \]

- when wr=0 in dst, then w=1 in src and side src can continue to send segments with 1 Byte of data each.
Congestion Control

- Side src keeps track of window size \( w \) as follows:
  \[
  w := \min(\max(w_r, 1), w_c) \text{ Bytes}
  \]
  where \( w_c \) is congestion window computed by src through alternating phases of slow start and congestion avoidance.
- Slow start:
  1. Initially, \( w_c := 1 \) MSS where MSS is max seg size 1500 Bytes
  2. As long as no sent seg is lost, then \( w_c := 2 \times w_c \) every RTT seconds
  3. When a sent seg is lost, then \( w_c := 1 \) MSS, and Step 2 is repeated until value of \( w_c \) is half of its value when seg loss is detected, and proceed to Step 1 in congestion avoidance phase.
- Congestion Avoidance:
  1. As long as no sent seg is lost, then \( w_c := w_c + MSS \) Bytes every RTT sec.
  2. When a sent seg is lost, then proceed to Step 3 in slow start phase.
TCP Segments

each TCP segment has following fields:

* src TCP port (in host that sent seg)
* dst TCP port (in host that rcvd seg)

* seq # of seg
* ack # of seg

* flags: SYN, ACK, FIN, RST

* rcv window wr of host that sent seg

* TCP checksum

* data bytes -- optional