Elements of Transport Protocols

- Transport protocol similar to data link protocols
- Both do error control and flow control
- However, significant differences exist
Elements of Transport Protocols

<table>
<thead>
<tr>
<th></th>
<th>Data link layer</th>
<th>Transport layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>directly via physical channel</td>
<td>over the entire network</td>
</tr>
<tr>
<td>Addressing</td>
<td>no need to specify address.</td>
<td>explicit addressing of destination is required</td>
</tr>
<tr>
<td></td>
<td>Just select outgoing line</td>
<td></td>
</tr>
<tr>
<td>Connection establishment</td>
<td>over a wire is simple</td>
<td>more complicated</td>
</tr>
<tr>
<td>Delay</td>
<td>frame either arrives or lost, not stored and delayed</td>
<td>packets might be stored for seconds and delivered later</td>
</tr>
<tr>
<td>Buffering and flow control</td>
<td>simpler</td>
<td>more complicated; large and dynamic number of simultaneous connections</td>
</tr>
</tbody>
</table>

Addressing

- Specify which host process to connect to
- TSAP: Transport Service Access Point
- In TCP, UDP, called **ports**
- Analogy: NSAP. Example: IP address
- Client or server app attaches to TSAP
- Connections run through NSAP
- TSAP to distinguish endpoints sharing NSAP
Addressing

How host1 process knows TSAP # at host2?

- Stable TSAP # listed in well-known places
  - e.g. `/etc/services` in UNIX list permanent #'s
  - mail server: 25, web server: 80

- Alternatively, special process: **portmapper**
  - listen on well-known TSAP
  - user conn to portmapper, specify service name
  - portmapper sends back TSAP address
  - similar to directory service (دليل الهاتف)
Addressing

- Many server processes used only rarely
- Waste if each process listen to TSAP all day
- Instead, use initial connection protocol
- Spec process server listens all known TSAP
- Act as proxy for lightly used servers
- e.g. inted, xinetd on UNIX
Connection Establishment

- Sounds easy; surprisingly tricky!
- Just send REQUEST, wait for ACCEPTED?
- Can lose, delay, corrupt, duplicate packets
- Duplicate may transfer bank money again!
- Protocols must work correct all cases
- Implemented efficiently in common cases
- Main problem is delayed duplicates
- Cannot prevent; must deal with (reject)

Solutions for delayed duplicates

- Not reuse transport address (TSAP)
  - difficult to connect to process
- Give each connection unique ID
  - seq # chosen by initiating party
  - update table listing obsolete connections
  - check new connections against table
- requires maintain certain amount of history
- if machine crashes, no longer identify old con
Connection Establishment

- To simplify problem, restrict packet lifetime
  - restricted network design: prevent looping
  - hop counter in each packet: -1 at each hop
  - timestamp in each packet: clock must be synced
- Must also guarantee ACKs are dead
- Assume a value $T$ of max packet lifetime
- $T$ sec after packet sent, sure traces are gone
- In the Internet, $T$ is usually 120 seconds

New method with packet lifetime bounded

- Label segments with seq # not reused in $T$
- $T$ and packet rate determine size of seq #s
- 1 packet w given seq # may be outstanding
- Duplicates may still occur, but discarded dst
- Not possible to have delayed duplicate old packet with same seq # accepted at dest
Connection Establishment

How to deal with losing memory after crash?

- Each host has time-of-day clock
  - clocks at different host need not be synced
  - binary counter increments uniform intervals
  - no. of bits must be ≥ of seq #
  - clock must be running even if host goes down
- Initial seq # (k-bits) ← low k-bits of clock
- Seq space must be so large
  - by time # wrap, old pkts w same # are long gone

Clock method work within connection
- Host don’t remember # across connections
- Can’t know if CONN REQUEST with initial seq # is a duplicate of a recent connection
- To solve this, use three-way handshake
- Check with other peer that con req is current
- Used in TCP, with 32-bit seq #
- Clock not used in TCP; attacker can predict
Connection Establishment

Normal Procedure

- H1 chooses initial s# x
- H2 replies
  - ACKs x
  - announce own s# y
- H1 replies
  - ACKs y
  - with 1st data segment

Abnormal situations

- Delayed duplicate CR
  - H2 sends ACK to H1
  - H1 rejects
  - H2 knows it was tricked

- Worst case
  DD CR, old ACK floating
  - H2 gets delayed CR, replies
  - H1 rejects
  - H2 gets old DATA, discards (z received instead of y)
Connection Release

- Easier than establish
- However, some pitfalls

- Asymmetric release
  - each con term separately
  - abrupt; may cause data loss
  - better protocol needed

Symmetric release

- Each direction is released independently
- Can receive data after sending DISCONNECT
- H1: I am done, are you done too?
- H2: I am done too, goodbye
- Two-army problem: unreliable channel
Connection Release

- **Two-army problem**
  - each blue army < white army, but together are larger
  - need to sync attack
  - however, only com channel is the valley (unreliable)
  - 3-way handshake? B1 can’t know ACK arrived
  - making 4-way handshake doesn’t help either

![Diagram of two armies and communication](image)

Connection Release

- Let each side independently decide its done
- Easier to solve

**Normal release sequence**
- H1 send DR, start timer
- H2 responds with DR
- when H1 recv DR, release
- when H2 recv ACK, release
Connection Release

Error cases, handled by timers, retransmissions

- Final ACK lost: Host 2 times out
- Lost DR: H1 starts over
- Extreme: Many lost DRs both release after N

Protocol usually suffices; can fail in theory
after N lost attempts; half open connection
Not allowing give up, can go on forever
To kill half open connections, automatically disconnect if no received segments in X sec
Must have timer reset after each segment
Send dummy segments to keep con alive
TCP normally does symmetric close, with each side independently close ½ con w FIN
Multiplexing

- Transport, network sharing can either be:
  - Multiplexing: connections share a network address
  - Inverse multiplexing: addresses share a connection