Transport Layer

- Transport Layer Services and Protocols
- Multiplexing and De-multiplexing
- Connectionless Transport: UDP
- Connection-Oriented Transfer: TCP
  a. Segment Structure
  b. Reliable Data Transfer
  c. Flow Control
  d. Connection Management
- Principles of Congestion Control

Transport Layer Services and Protocols

Transport Layer Services provide ***logical communication*** between application processes running on different host.

Transport layer Protocols run on **end-systems**:

Sending side: breaks ***Application Layer messages*** into ***segments***, passes them to ***Network Layer***

Receiving side: reassembles ***segments*** into ***messages***, passes them to ***Application Layer.***

Transport layer Protocols available to applications:

Internet: ***TCP*** and ***UDP***

Transport Layer v/s Network Layer

Network Layer provides ***logical communication*** between hosts.

Transport Layer provides ***logical communication*** between processes relies on, enhances, **network layer** services.

Household Analogy: e.g. *12 kids sending letters amongst themselves.*

- processes = kids
- application messages = letters in envelopes
- hosts = house numbers
- Transport Layer protocol = Ann and Bill (identified & unique processes.)
- Network Layer protocol = postal service

Internet Transport-Layer Services

TCP

- ***reliable, in-order delivery***
- ***congestion control***
- ***flow control, error control***
- ***connection setup***

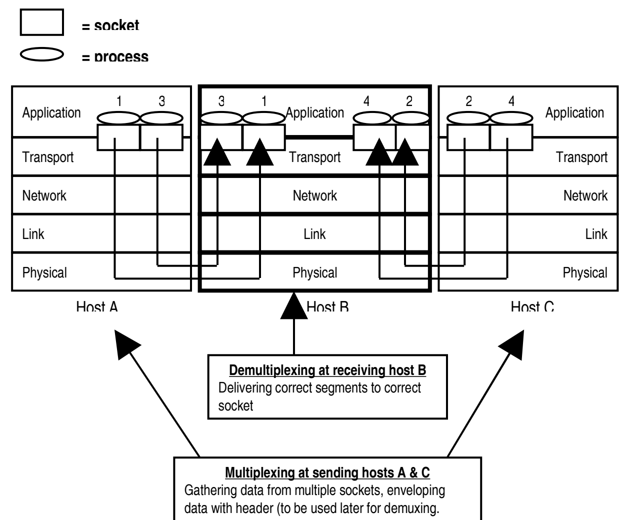
UDP

- ***no-frills (best-effort service)***
- ***unreliable, unordered delivery***

SERVICES NOT AVAILABLE TO BOTH:

- ***Delay guarantees***
- ***Bandwidth guarantees***

Multiplexing and Demultiplexing

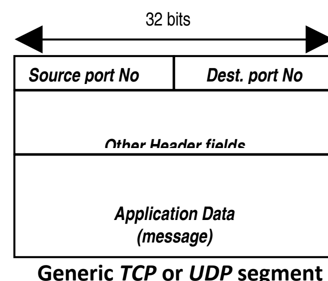


How demultiplexing works?

Host receives ***IP datagrams*** (will learn later)

1. each datagram has source ***IP address***, destination ***IP address. (will learn later)***
2. each datagram carries ***1 Transport Layer segment***
3. each segment has ***source and destination port numbers***. (***recall: well-known port numbers for specific application ptotocols***)

Host uses **IP addresses & Port Numbers** to direct segment to appropriate socket.

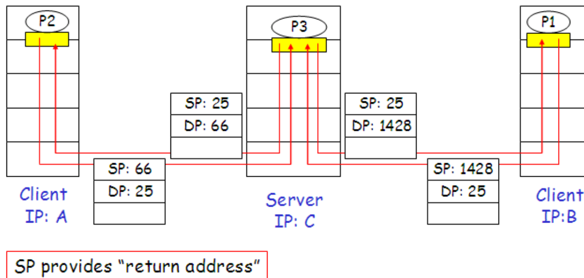


Generic *TCP* or *UDP* segment

Connectionless demultiplexing: UDP

UDP socket identified by two-parameters: (***dest IP address***, ***dest port number***)

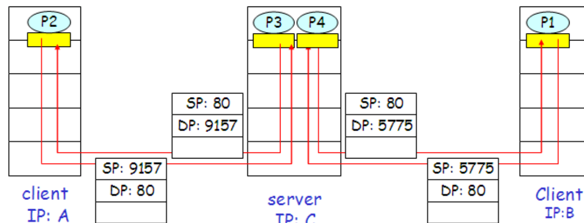
- When host receives ***UDP segment***:
  checks ***destination port number*** in segment.
  directs UDP segment to socket with that ***port number.***
- ***IP datagrams*** with different source ***IP addresses*** and/or ***source port numbers*** directed to same socket.



Connection-oriented demultiplexing: TCP

TCP socket identified by four parameters: (***source IP address***, ***source Port number***, ***dest IP address***, ***dest port number***)

- Receiving host uses all four values to direct segment to appropriate socket.
- Server host may support many simultaneous **TCP** sockets:
  - each socket identified by its own 4 parameters.
- Web servers have different sockets for each connecting client
  - *Non-persistent HTTP* will have different socket for each request.



Connectionless Transport: UDP [RFC 768]

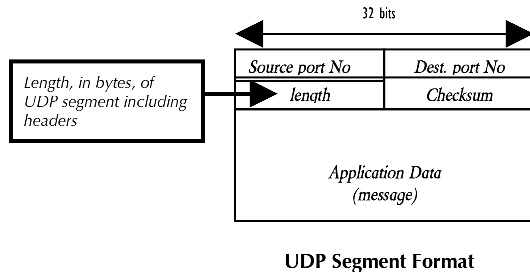
- "no frills" Internet Transport Protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to applications
- *connectionless-oriented:*
  - no handshaking between parties
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

Where do we use UDP?

- often used for ***streaming multimedia*** applications
  - **loss tolerant**
  - **rate-sensitive**
- other UDP uses
  - **DNS (*Domain Name Service*)**
  - **SNMP (*Simple Network Management Protocol*)**
- How to achieve "reliable" transfer over UDP?
  - **add reliability at application layer.**
  - **Application-specific error recovery!**



UDP Segment Format

UDP Checksum

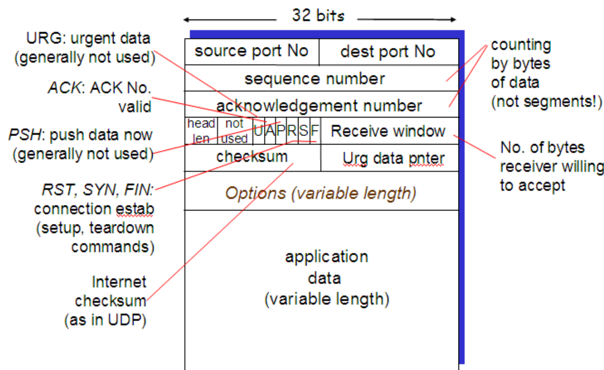
- ***Goal:*** detect "errors" (flipped bits) in transmitted segment
- ***Sender:***
  - treat segment contents as **sequence of 16-bit integers.**
  - **checksum**: addition (1's complement sum) of segment contents
  - sender puts ***checksum value*** into ***UDP checksum field***
- ***Receiver:***
  - compute ***checksum*** of received segment
  - check if computed ***checksum*** equals *checksum field value*:
    NO - error detected
    YES - no error detected. *But maybe errors nonetheless?* More later ….

Connection-Oriented Transport: TCP [RFC 793]

Transmission Control Protocol

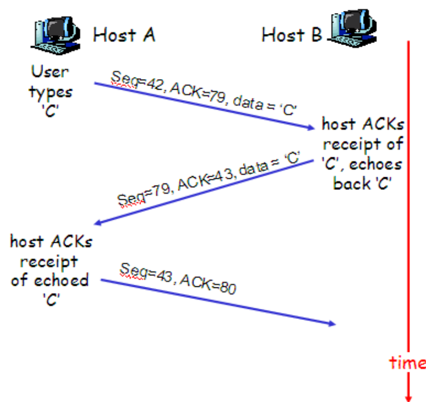
- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order *byte stream:***
  - no "message boundaries"
- **pipelined:**
  - TCP congestion and flow control set window size
- **send & receive buffers**
- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- **connection-oriented:**
  - handshaking (exchange of control messages) initialise sender, receiver states before data exchange
- **flow controlled:**
  - sender will not overwhelm receiver

TCP Segment Format



TCP Sequence Nos. and ACK Nos.

Simple Telnet Scenario:



- **Sequence Nos.:**
  - byte stream "number" of first byte in segment's data.
- **ACK Nos.:**
  - Sequence No. of next byte expected from other side
  - cumulative ACK

TCP Round Trip Time (RTT) and Timeout

Question: *How to set TCP timeout value?*

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

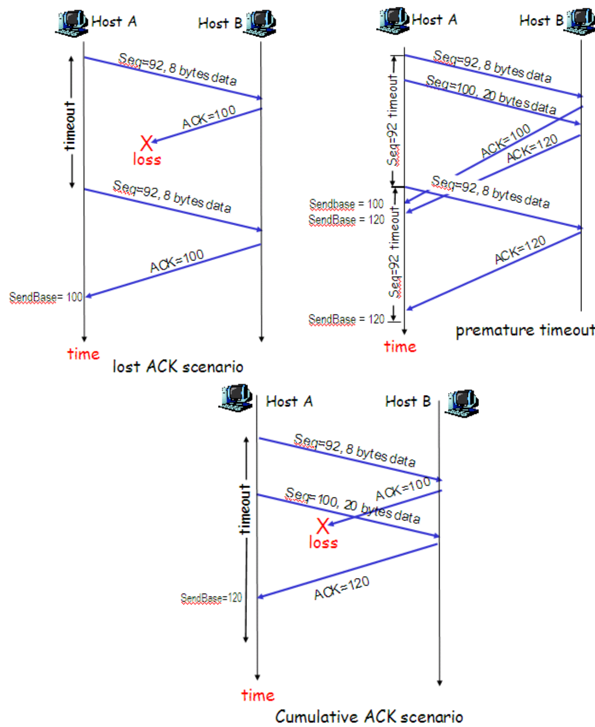
Reliable Data Transfer

- TCP creates *reliable data transfer* 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

TCP Sender Events

- <u>Data received from Applications:</u>
  - Create segment with sequence nos.
  - Sequence no. 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
- <u>Timeout:</u>
  - retransmit segment that caused timeout
  - restart timer
- <u>ACK received:</u>
  - If acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are any outstanding segments

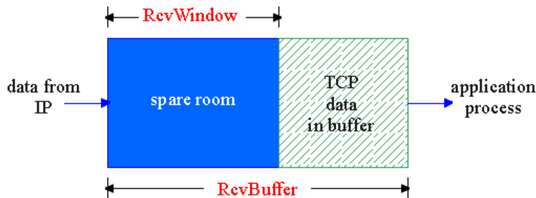
3 TCP Transmission Scenarios



TCP Flow Control

Reason: Sender won't overflow receiver's buffer by transmitting too much, too fast.

- ❖ receive side of TCP connection has a receive buffer:



Application process may be slow at reading from buffer
Speed-matching service: matching the send rate to the receiving application's drain rate.

How it works?

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

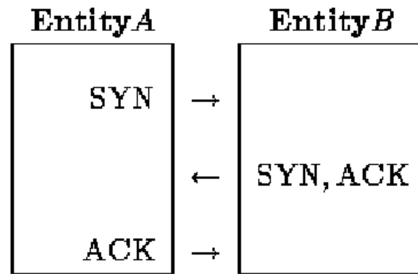
- ❖ **spare room in buffer**
  = RcvWindow
  = RcvBuffer-[LastByteRcvd - LastByteRead]
- ❖ Reciever advertises spare room by including value of RcvWindow in segments
- ❖ Sender limits unACKed data to RcvWindow
- ❖ guarantees receive buffer doesn't overflow

TCP Connection Management

- ❖ TCP sender, receiver establish "connection" before exchanging data segments
- ❖ initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)

Opening a connection (Three-Way Handshake)

- ❖ Step 1: client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data
- ❖ Step 2: server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #
- ❖ Step 3: client receives SYNACK, replies with ACK segment, which may contain data



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.

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.