

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

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*

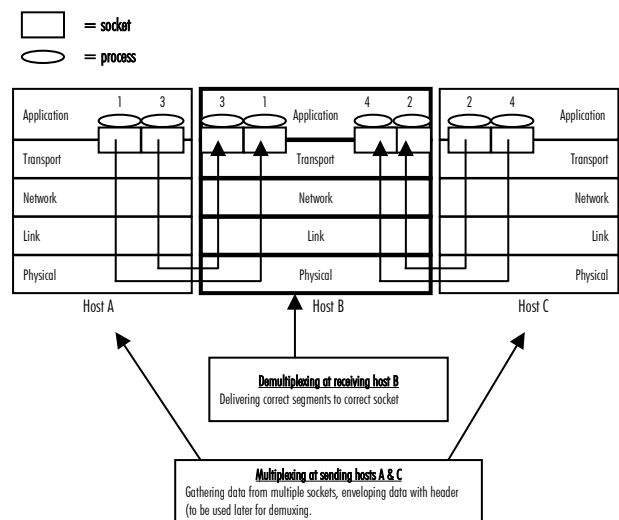
UDP

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

SERVICES NOT AVAILABLE TO BOTH:

- *Delay guarantees*
- *Bandwidth guarantees*

Multiplexing and Demultiplexing

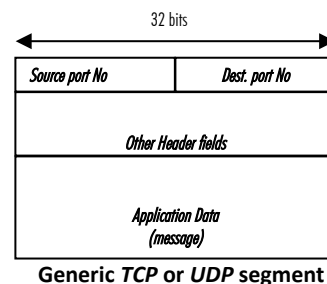


How demultiplexing works?

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 protocols)

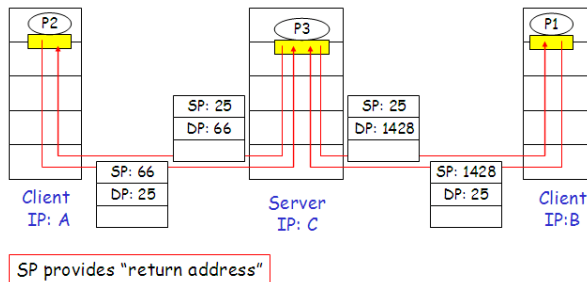
Host uses **IP addresses & Port Numbers** to direct segment to appropriate socket.



Connectionless demultiplexing: UDP

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

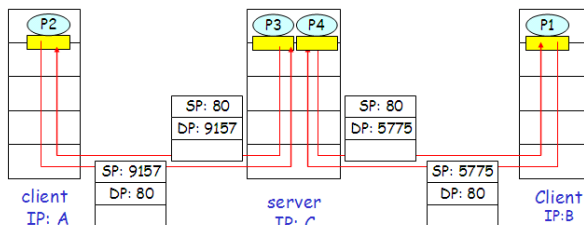
- ❖ When host receives **UDP segment**: checks **destination port number** in segment. directs UDP segment to socket with that **port number**.
- ❖ **IP datagrams** with different source **IP addresses** and/or **source port numbers** directed to same socket.



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]

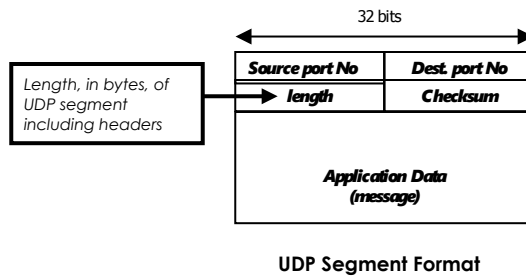
- ❖ “no frills” Internet Transport Protocol
- ❖ “best effort” service, UDP segments may be:
 - lost
 - delivered out of order to applications
- ❖ *connectionless-oriented*:
 - no handshaking between parties
 - each UDP segment handled independently of others

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 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

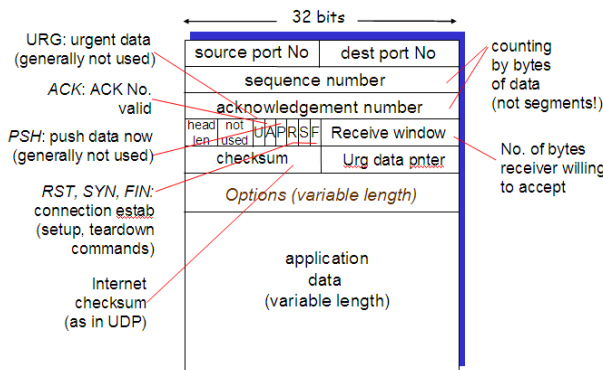
- ❖ **Sequence Nos.:**
 - byte stream “number” of first byte in segment’s data.
- ❖ **ACK Nos.:**
 - Sequence No. of next byte expected from other side
 - cumulative ACK

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

TCP Segment Format

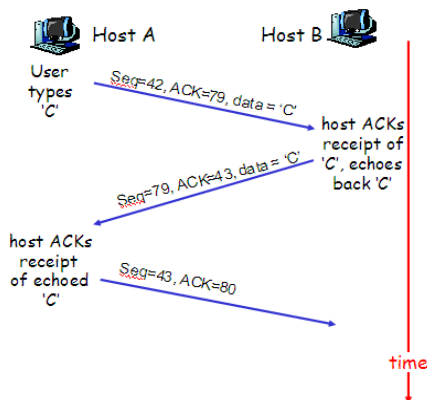


Reliable Data Transfer

- ❖ 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 Sequence Nos. and ACK Nos.

Simple Telnet Scenario:



TCP Sender Events

- ❖ Data received from Applications:
 - 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
- ❖ Timeout:
 - retransmit segment that caused timeout
 - restart timer
- ❖ ACK received:
 - If acknowledges previously unACKed segments
 - update what is known to be ACKed
 - start timer if there are any outstanding segments

