Transport layer services: provide logical communication between app processes on different hosts.

Send side breaks app messages into segments and passes to network layer, Receive side reassembles message and passes to app layer

difference between network and transport layer: network layer is communication between hosts, transport layer is communication between processes

UDP and TCP transport protocol availiable

12 kids sending 12 letters analogy: processes -> kids, app messages -> letters in envelopes, hosts -> houses, transport protocol -> parents, network protocol -> postal service

TCP: reliable, in-order delivery with congestion control, flow control, and connection setup

UDP: unreliable, unordered delivery. Best effort

Services not available: delay and bandwidth guarantees

Multiplexing: Gathering data from multiple sockets, enveloping data with header (used for demultiplexing)

Demultiplexing: Delivering received segments to correct sockets

How demultiplexing works: Host receives datagram with source and destination IP, datagram contains transport layer segment containing source and destination port number. Uses source and destination IP and port numbers to deliver to correct sockets.

Connectionless demultiplexing uses only destination IP and port number. UDP socket is identified by two-tuple. IP datagrams with different source IP and/or port numbers are delivered to same socket. DatagramSocket serverSocket = new DatagramSocket(6428);

Connection-oriented demux is 4-tuple: source IP, source port number, destination IP, destination port number. Server host may support many simultaneous TCP sockets. Web servers have sockets for each connecting client, non-persistent HTTP will have different socket for each request

UDP advantages: no connection needed, simple, no congestion control (quicker)

UDP segments may be lost or delivered out of order to app

UDP used for streaming multimedia because it is loss tolerant and data needs to be transmitted quickly. UDP also used for DNS and SNMP.

Possible to add packet retrieval reliability at application layer

UDP checksum used to detect errors

Sender sends checksum to receiver, the receiver computes the checksum value then compares it to received checksum value. If they are the same there is no error, if they are different there is an error

sending and receiving processes are on application layer, the reliable channel in which they communicate is in the transport layer

rdt\_send() -> reliable data transfer protocol (send side) <-> udt\_send() <-> unreliable channel <-> rdt\_recieve() <-> reliable data transfer protocol (receive side) -> deliver\_data()

Sender sends data into underlying channel, receiver reads data from underlying channel, underlying channel is reliable

ACK: receiver tells sender that packet was received ok

NAK: receiver tells sender that packet had errors

sender retransmits packet if ACK/NAK corrupts. Each packet has sequence number and duplicate packets with same sequence number are discarded

rdt2.2 same as rdt2.1 but uses only ACK, does not use NAK. Dupliate ACK at sender works same as NAK

rdt3.0 sets countdown value, if no ACK is received in set amount of time then packet is retransmitted. If ACK is just delayed receiver will discard duplicate packet with same sequence number

rdt3.0 performance stinks, dtrans = packet size/data transmission rate

Usender = dtrans/RTT + dtrans

pipelining: sender allows multiple yet-to-be-acknowledged packets

two pipelining protocols: go-Back-N, selective repeat

Usender = (num packets \* dtrans)/ RTT + dtrans

Go-Back-N: sends multiple packets in one window and each get scanned and acked. Packets that don’t receive ack and receive multiple previous ACKS get resent

Selective repeat: receiver individually acknowledges all correctly received pkts, sender only resneds pkts with no ACK

TCP is one sender, one receiver. in-order byte stream, pipelined, sender will not overwhelm receiver, bi-directional data flow in same connection, MSS: maximum segment size, connection-oriented

EstimateRTT = (1-a)\*estimatedRTT + a\*sampleRTT, a = .125 usually

DevRTT = (1-B)\*DevRTT + B = .25 usually

cumuluative acks, single retransmission timer, retransmissions by timeout and duplicate acks

fast retransmit: If sender receives 3 acks for same data, it assumes it is lost

Three way handshake: client sends TCP syn segment to server with initial sequence num and no data, server receives SYN and replies with SYNACK and allocates buffers, client receives SYNACK, replies with ACK segment which may contain data

Congestion: too many sources sending too much data too fast for network to handle. Results in lost packets and long delays

one router, infinite buffers. No retransmission causes congestion

one router, finite buffers. Sender retransmission of lost packet cause of congestion.

When packet dropped, any upstream transmission capacity used for that packet was wasted

two approaches to congestion control: end-end congestion control, network-assisted congestion control

end-end: no feedback from network, congestion inferred from end-system observed loss, delay, approach taken by TCP

network-assisted: routers provide feedback to end systems, single bit indicating congestion

ABR (available bit rate): if sender’s path is congested, use minimum guaranteed rate. If path is not congested, use available bandwidth

RM (resource management): sends bits in RM cell (network assisted), NI bit: no increase in rate (mild congestion) CI bit: congestion indicated

decentralized: each TCP sender sets its own rate based on feedback, ACK segment received for network not congested so increase sending rate. Lost segment, assume network is congested and decrease sending rate

when cwnd > ssthresh, grow cwnd linearly. Increase cwnd by 1 MSS per RTT.   
cwnd += MSS/cwnd

when cwnd < ssthresh, sender in slow-start phase, window grows exponentially. When cwnd >= ssthresh, sendor is in congestion-avoidance phase, grow window linearly. When triple duplicate Ack occurs, set ssthresh to cwnd/2 and set cwnd to ~ssthresh. When timeout occurs, set ssthresh to cwnd/2, set cwnd to 1 MSS.

Window = window/RTT. After loss Window = Window/2\*RTT

rate = R/K, R = bandwidith, K = TCP sessions when all using same bottleneck