## The Transport Layer

Link Layer

* Transport between adjacent network elements

Network layer

* Transport between hosts and end points

Transport layer

* Transport processes

Logical communication

* Transport layer doesn’t actually connect directly to other process, but it seems like it does from the transport layer’s perspective

Runs on end systems, not routers

* Sending: Break data from apps into segments, pass to network layer
* Receiving: Reassemble segments into messages, pass to apps

Two transport protocols:

* TCP
  + Reliable, in order
* UDP
  + Unreliable, unordered

## 

## Sockets

UDP

* Assigned to random port not currently in use for UDP
* Can then be bound to specific port
* When sending packet through socket, transport header appended to it
  + Source port
    - Automatically determined port socket is bound to
  + Dest port
    - Port the receiving process is listening on
    - Specified by program / user
* Fully defined as (dest\_addr, dest\_port)
  + Need to include source\_port to allow response back
  + If two clients both send to same dest\_addr, dest\_port, data will be sent to same process

TCP Sockets

* Adds complexity
* Defined with a four-tuple: source\_addr, source\_port, dest\_addr, dest\_port
  + All used when demultiplexing TCP data
  + How each client is directed to unique socket
* Server has “welcoming socket” waiting for connection requests
* Client creates a socket and sends connection request to the server
  + Server notes request and creates a new socket, assigned to random, unused port
  + Transport layer takes note of new port and maps TCP socket to port assigned to new socket
  + Unlike UDP, if two clients send data, both get different sockets

# UDP

Used for:

* Finer app-level control over what data is sent and when
  + Immediately packages segments and passes them to network layer
  + TCP can delay sending and waste time resending dropped packets
* No connection establishment
  + Skips extra RTT required
* No connection state
  + Minimizes memory and computation overhead
* Small header overhead
  + 7 vs 20 bytes

Basic Error Checking with UDP checksum

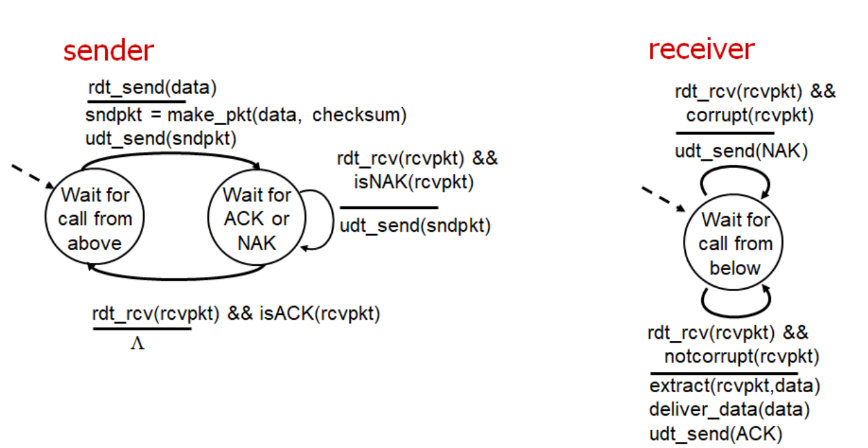
* Send performs 1 complement of sum of all 16-bit words in segment and puts in checksum field
* Receiver also sums all 16-bit words and adds them to checksum\
* If any 0s result, error occurred

# Reliable Data Transfer

RDT 1.0 – Reliable Transfer over Reliable Channel

* Assume underlying channel is perfectly reliable
  + No bit errors
  + No packet loss
* Separate FSms for sender and receiver

RDT 2.0 – Handle bit errors in underlying channel

* Channel may flip bits
  + Include checksum to detect
* How to recover from errors:
  + ACK: Receiver tells sender the packet was received okay
  + NAK: Receiver tells sender packet had errors

RDT 2.1 – Handle bit errors in ACK/NAK

* 2.0 has a flaw, ACK/NAK can be corrupted
* Solution: Add a sequence number field to each packet
  + Discard duplicate
* Stop and wait
  + Sender waits for receiver to respond

RDT 2.2 – Remove NAK, only use ACK

* Simplifies protocol, lays ground for future
* Sends an ACK that includes sequence number of packet being acknowledged

RDT 3.0 – Handle lost packets in underlying channel

* Adds retransmission timer, resends packet if no ACK before timer expires
  + Results in duplicate packets, but sequence numbers prevent sender from thinking duplicate is new
* Works, terrible performance
  + Stop and wait
* Improved with pipelining:
  + Multiple yet-to-be-ACK’d packets
  + Adds additional valid sequence numbers
  + Buffers at sender and/or receiver

Pipeline Protocols

* Go-Back-N
  + Sender have up to N unACKed packets
  + Receiver send cumulative ACK
    - Don’t ack new packets if earlier gap
  + Sender has timer for oldest unacked packet
    - Resends all unACKed packets when timer expires
* Selective Repeat
  + Up to N unACKed packets in pipeline
  + Individual ACK for each packet
  + Sender maintains timer for each unACKed packet, when timer expires transmit that packet

Summary:

* Checksums → detect if pkt is corrupted
* ACK messages → sender knows pkt received OK
* Sequence numbers → sender/rcvr can keep track of pkts
* Timers → sender can recover from a lost pkt
* Pipelining → increases ability to utilize available resources
* Windowing → manage ability to keep track of un-ACK’d pkts

# TCP

Connection Oriented

* Three-way handshaking procedure
* “Logical”, software-based, no dedicated resources
* Only hosts know about connecting, intermediate elements don’t implement transport layer

Full-duplex

* Data flows both directions
* Both hosts send data and receive ACKs

Point-to-point

* Connections always run between two, and only two hosts
* No multicasting

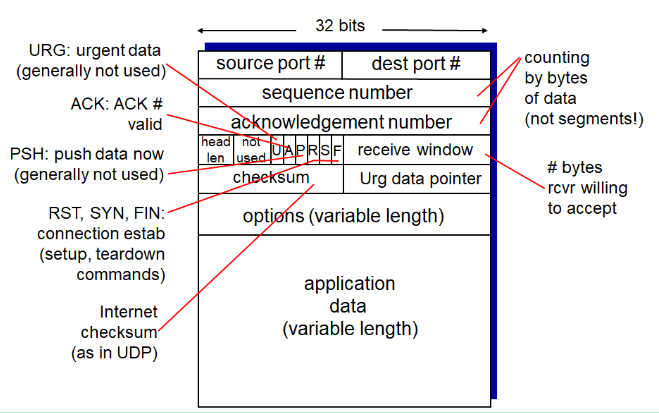
Reliable, in-order byte stream

* No data lost or corrupted
* Data delivered in order it was sent
* TCP doesn’t detected “message boundaries”

Pipelined

* Window size adjusted based on congestion and flow control
  + Sender will not overwhelm receiver

Handshaking process

* Client issues connection request to server
  + Special TCP segment, no data
* Server responds
  + Special TCP segment., no data
* Client completes handshake
  + Special TCP segment, can have data
* Several parameters set up, including reserving memory for send and receive buffers

Delimiters

* String indicating end of piece of data
* HTTP:
  + \r\n = end of parameter
  + : = separates header name from value
  + \r\n\r\n = end of header
  + Split on delimiter character to find boundary data

Size of TCP segments

* Determined by sender’s MTU
  + Maximum Transmission Unit
  + Largest li1 nk-layer packet than can be set
  + Ethernet and PPP have MTU of 1500 bytes
  + All data must fit inside
* Maximum Segment Size = MTU – TCP/IP Header – TCP Header
  + 1500 – 40 – 20 = 1440 MSS

Seq and ACK #s in TCP

* Refer to bytes in data
* Views packets as stream of bytes:
  + Seq # = Number of first byte in segment being sent
  + ACK # = Next byte the receiver expects from the sender
  + Uses cumulative ACK
* Starting number is random , not zero

RTT Estimation

* Samples RTT by computing time between segment and ACK
  + Only one at time
  + Not for retransmitted segments
* EstimatedRTT = (1 – a ) \* EstimatedRTT + a\*SampleRTT
  + A = 1/8

RTT Variability – Weighted moving average of difference between SampleRTT and EstimatedRTT

* DevRTT = (1 – B) \* DevRTT + B \* |SampleRTT – EstimatedRTT|
* B = 0.25

Timeout Interval:

* Timeout = EstimatedRTT + [margin]
* Increase margin when DevRTT is high, decrease when low
* 1 second:
  + Timeout = EstimatedRTT + 4 \* DevRTT