EP2120 Internetworking

IK2218 Protocols and Principles of the Internet

Transport Layer
UDP, TCP and beyond

Lecture 8

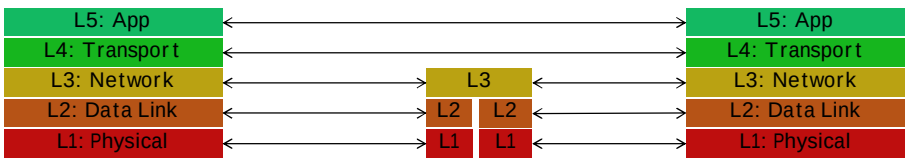
György Dán
KTH/EE/LCN

Literature:

Forouzan, TCP/IP Protocol Suite

(3ed Ch 11,12.1-12.4)(4ed Ch 13,14,15.1-15.4)

Transport Layer



Transport Layer

Purpose:

Logical process-to-process communication

- Implemented in “end systems” only
- Provides service to the applications (processes)
  - Implements service models widely used by applications
- Uses the services of the network layer
  - IP datagram service
- Several transport layer protocols coexist in the Internet

Transport Layer Service Models

- Connectionless
- Unreliable
- Message oriented
- Simplex/Semi-duplex

- Connection-oriented
- Reliable
- Byte-stream
- Full-duplex

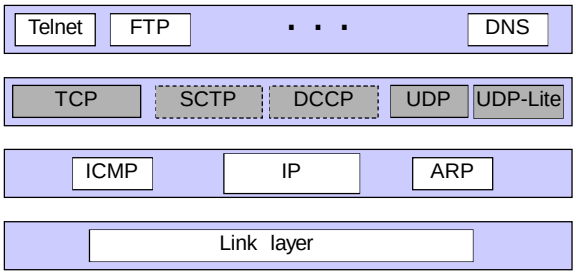
TCP/IP Transport Layer Protocols

Two main transport layer protocols

- UDP – User Datagram Protocol
  - Connectionless unreliable service
- TCP – Transmission Control Protocol
  - Connection-oriented reliable full-duplex bytestream service w. congestion control

Others, not widely available yet

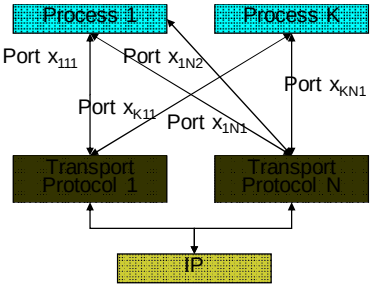
- SCTP, DCCP, UDP-Lite



UDP, TCP

Transport layer Mux-Demux

- Protocol port
  - Source/destination of a transport layer message
    - Processes send to and listen at ports
  - Identified by an integer (16 bits)
    - Implicitly identifies the process that uses the port
- Socket
  - IP address AND port number
- Each message must carry
  - destination socket
  - sender socket
  - protocol identifier



Three Groups of Port Numbers

Range	Name	Purpose
0 .. 1023	Well-known ports	•Assigned and controlled by IANA •Normally used for servers/services –e.g., 80 for HTTP, 443 for HTTPS
1024 .. 49151	Registered ports	• Registered by IANA – non-exclusive • Normally used for application services – e.g., 1862/tcp MySQL Cluster Manager Agent
49152 .. 65535	Dynamic ports	•Ephemeral, can be used by any process •Normally used by client processes

See: http://www.iana.org/assignments/port-numbers

UDP

User Datagram Protocol - RFC 768

User Datagram Protocol

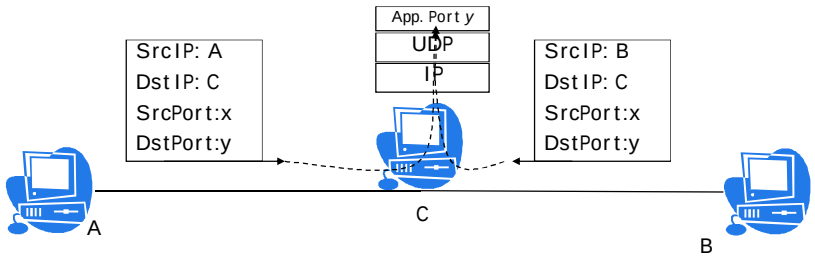
Best-effort datagram service

- Connectionless message-oriented unreliable
- UDP functions
  - Error detection
  - Multiplexing/demultiplexing

  "No frills", "bare bones" transport layer protocol
- Applications using UDP
  - DNS, DHCP, SNMP, NFS, VoIP, etc.
  - UDP is a base that you can build your own protocols on

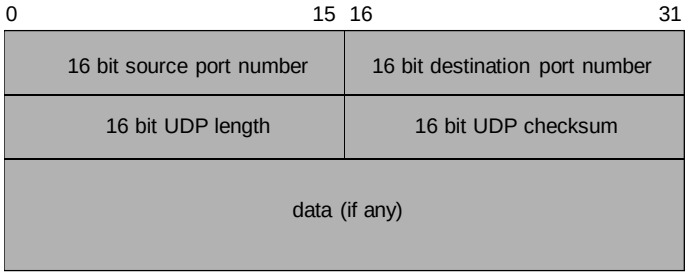
Multiplexing/Demultiplexing

- Communication between application processes
  - Port number identifies receiving process
- UDP data delivery based on
  - Receiver port number
  - Receiver IP address

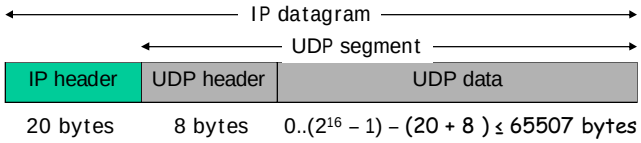


UDP Segment Format

- 8 byte header + app. layer data

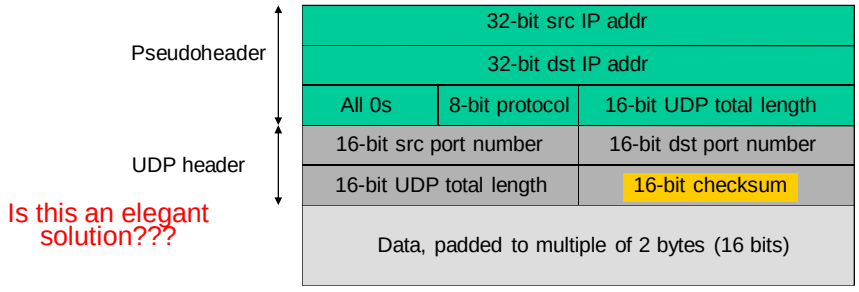


- UDP length field is redundant
  - IP could pass this info to UDP



Error detection and Pseudo-header

- UDP checksum (one's complement)
  - Application data, UDP header, pseudoheader, and pad byte (if needed)
  - Pseudoheader and pad byte not transmitted, only used for computation
  - Optional in IPv4, mandatory IPv6
- Purpose of the pseudo-header
  - Double-check that packet arrived to intended IP address
  - Check that IP delivered the packet to the correct protocol (UDP/TCP)



Is this an elegant solution???

Maximum UDP Segment Size

- Theoretical limit
  - IPv4: 65,507 bytes of payload
    - 65,535 bytes (IPv4) - 20 bytes IP header - 8 bytes of UDP header
  - IPv6: 65,527 bytes of payload
    - 65535 bytes payload – 8 bytes UDP header
- Practical limit
  - Sockets API limits size of send and receive buffer (~8 kbytes)
    - can be changed
  - TCP/IP implementation
    - Various limits (even with loopback interface) – see Stevens "Unix network programming"
  - Avoid IP fragmentation (why?)
    - min IPv4 host MTU=576 bytes → 512 bytes or less of data to ensure delivery
      - DNS, TFTP, BOOTP, and SNMP

Quiz

- An application sends "ABC", then "DEF" through UDP port x on Host B to port y on Host A. At the same time, an application sends "GHI", then "JKL" through UDP port z on Host B to port y on Host A. The application process on host A will receive
  - a) ABC, DEF, GHI, JKL
  - b) ACB, DEF
  - c) GHI, ABC
  - d) ABC, DEF or GHI, JKL

UDP, TCP

UDP Summary

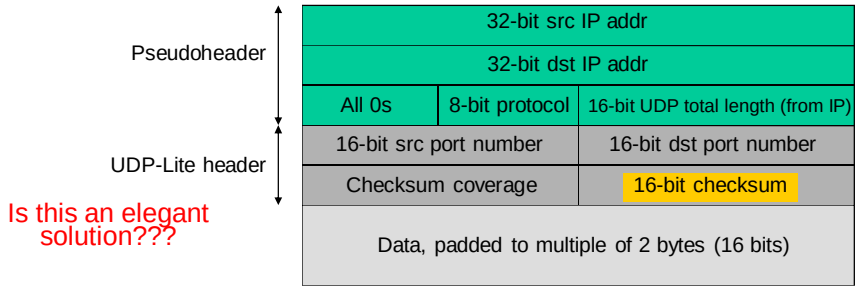
- Transport Layer Basics
  - Transport layer: inter-process communication
  - Port numbers for mux/demux
- UDP
  - Simple connectionless message-oriented unreliable protocol
  - Basic error detection

Continue

- TCP

UDP-Lite (rfc3828)

- Multimedia applications may benefit from receiving corrupted data
  - e.g., bit errors (wireless links)
- UDP-Lite checksum (one's complement using pseudoheader)
  - Coverage specified in header (min. 8 bytes)
  - Pseudoheader and pad byte not transmitted, only used for computation
- Complication
  - Link layer complexity increases (Cross-layer information needed)
    - Link layer should not discard the datagram if CRC fails
    - Link layer checks UDP-Lite checksum to verify header is correct



Is this an elegant solution???

TCP

Transmission Control Protocol
(RFCs 793,2581,2018,3390,...)

Why TCP?

- rfc793
  - "Computer communication systems are playing an increasingly important role in military, government, and civilian environments. This document focuses its attention primarily on military computer communication requirements, especially robustness in the presence of communication unreliability and availability in the presence of congestion, but many of these problems are found in the civilian and government sector as well. "
- Reliable data transfer useful service abstraction for many applications
  - TELNET (virtual terminal), FTP (file transfers), SMTP (email), HTTP (Web), Video streaming!

TCP service model

- Connection-oriented
  - Not a virtual circuit
- Between exactly two end-points
  - Broadcast and multicast are not applicable to TCP (use UDP)
- Full duplex
- Reliable and in-order
  - Delivery is not guaranteed but reception is known
- Byte stream service
  - A stream of 8-bit bytes is transmitted over the TCP connection
  - No record markers inserted by TCP
    - The receiver can *NOT* tell what sizes the individual writes were at the sender

TCP functions

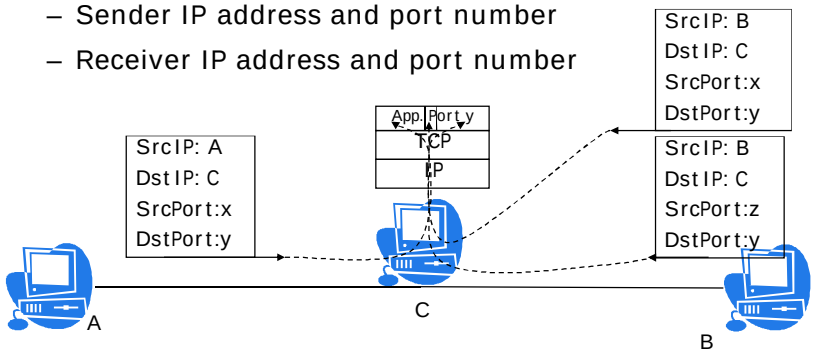
- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Retransmission scheme
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

TCP functions

- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Retransmission scheme
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

Multiplexing/Demultiplexing

- Communication between application processes
  - Port numbers as in UDP
  - But…
- TCP connection identification
  - Sender IP address and port number
  - Receiver IP address and port number

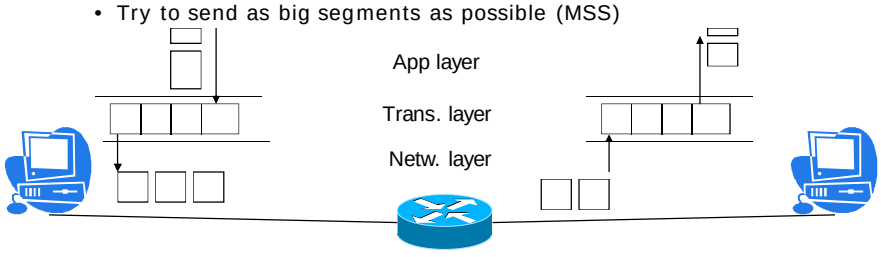


TCP functions

- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Retransmission scheme
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

From byte stream to datagram

- Data arrive and are delivered as byte stream
  - How should the data be sent over the network?
    - Network layer datagram size limitations
    - Network efficiency vs. latency
  - Need to identify every byte ⇒ sequence number
- Send and receive buffers for data
  - For each TCP connection
  - Used for segmentation
    - Try to send as big segments as possible (MSS)

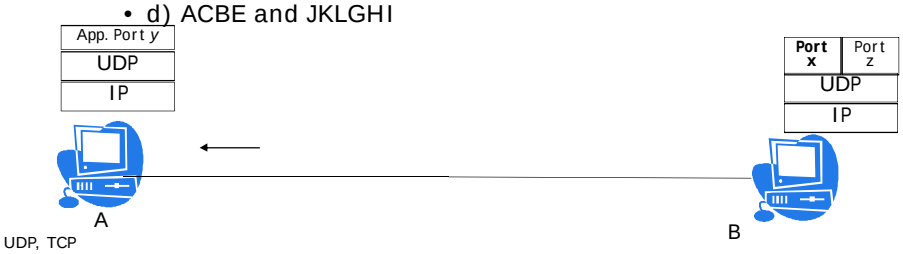


Maximum Segment Size - MSS

- The largest chunk of data TCP will send to the other side
  - Can be announced in the options field of the TCP header during connection establishment
- If not announced, a default value is assumed
  - 576 bytes host MTU requirement in IPv4 : 536 bytes
  - 1280 bytes MTU requirement in IPV6: 1220 bytes
- Large MSS means
  - Less overhead (headers)
  - Less segments to take care of (will see later)
    - Until fragmentation occurs (Path MTU discovery)
  - Potentially more delay

Quiz

- An application sends “ABC”, then “DEF” through TCP port x on Host B to port y on Host A. At the same time, an application sends “GHI”, then ”JKL” through TCP port z on Host B to port y on Host A. The application process on host A will receive
  - a) ABC, DEF and GHI, JKL
  - b) ABCDEF and GHIJKL
  - c) ABCD and GHIJKL
  - d) ACBE and JKLGHI



UDP, TCP

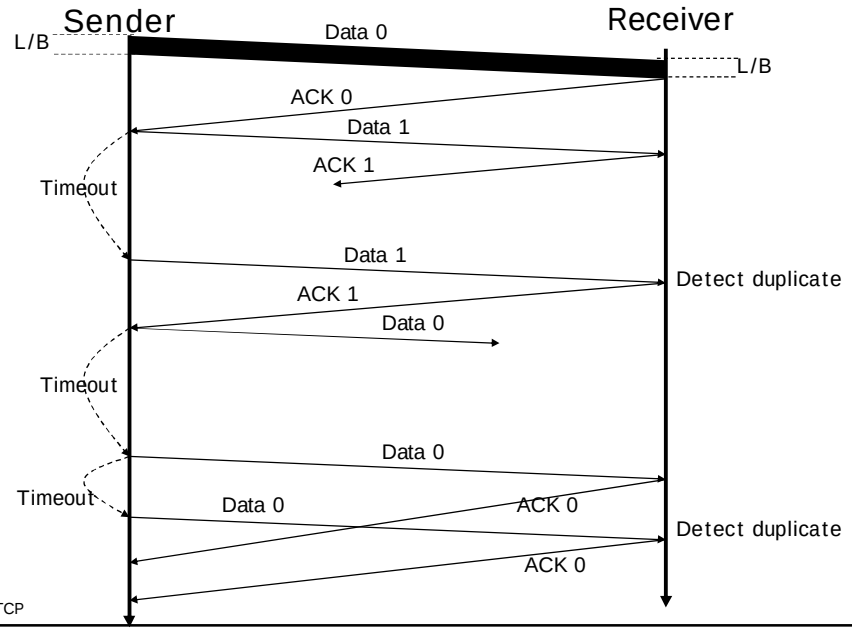
TCP functions

- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Reliable transmission over unreliable channel
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

Reliable Communications

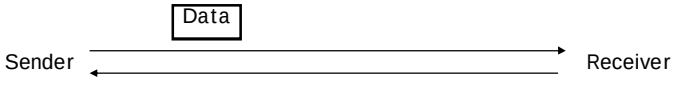
- Provide reliable transmission over error prone channel
- Error-prone channel
  - Noise ⇒ Bit error ⇒ Packet corruption ⇒ Packet drop
  - Congestion ⇒ Packet drop
- Automatic Repeat Request (ARQ)
  - ACK
  - Negative ACK (NACK)
- Flavors
  - Stop-and-wait
  - Go-back-N
  - Selective-Repeat

Stop-and-Wait



Stop-and-Wait Pros/Cons

- Simple operation
  - m=1 bit counter to detect duplicate data and ACK
  - No buffering needed in receiver
- Poor performance

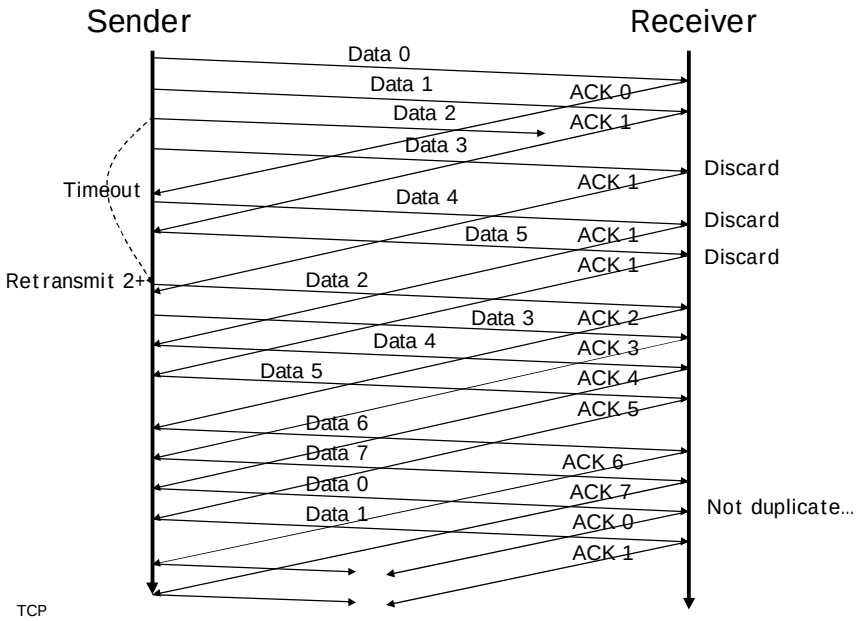


  - Link bitrate: B
  - Packet length: L
  - Round trip time: RTT
  - Example: Stockholm-NY (1000bytes,10Mbps,100ms RTT)

$$U_{SnW} = \frac{L/B}{RTT + L/B}$$

$$U_{SnW} \approx \frac{10^3 * 8/10^7}{0.1 + 10^3 * 8/10^7} \approx 0.007$$

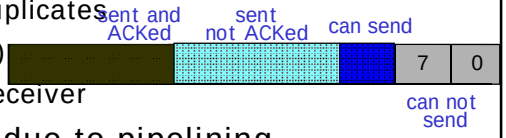
Go-Back-N



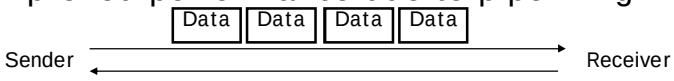
TCP

Go-Back-N Pros/Cons

- Relatively simple operation
  - *m* bit counter to avoid duplicates
  - Sender window (N≤2^m-1)
  - No buffering needed in receiver



- Improved performance due to pipelining



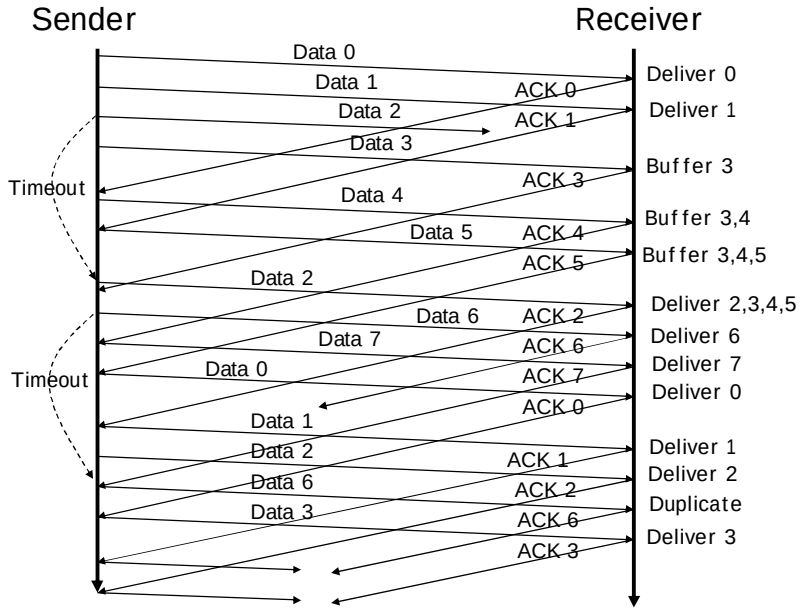
  - Link bitrate: B
  - Packet length: L
  - Round trip time: RTT
  - Example: Stockholm-NY, N=10

$$U_{GBN} = \min(1, \frac{N*L/B}{RTT + L/B})$$

$$U_{GBN} \approx \frac{10*10^3*8/10^7}{0.1+10^3*8/10^7} \approx 0.08$$

TCP

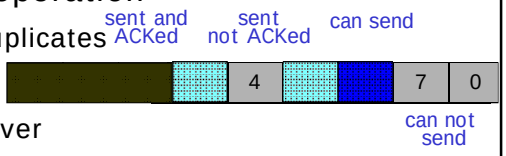
Selective Repeat-N



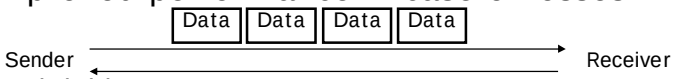
TCP

Selective Repeat Pros/Cons

- Slightly more complex operation
  - *m* bit counter to avoid duplicates
  - Sender window ($N \leq 2^{m-1}$)
  - Buffering needed in receiver



- Improved performance in case of losses



  - Link bitrate: B
  - Packet length: L
  - Round trip time: RTT
  - Example: Stockholm-NY, N=10

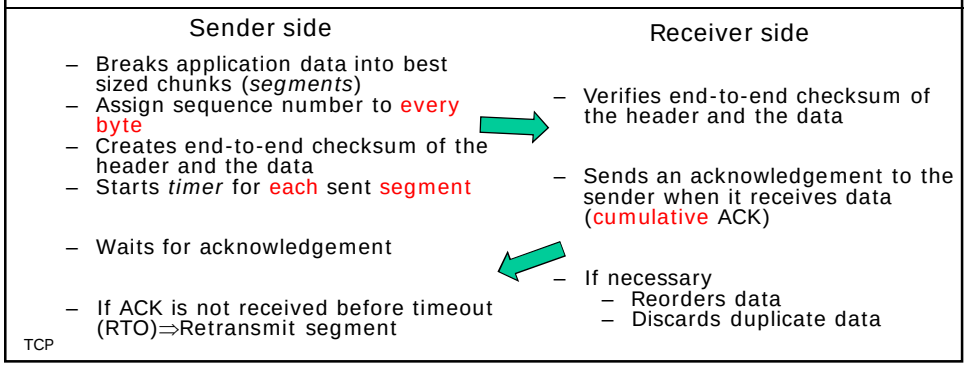
$$U_{SRN} = \min(1, \frac{N * L/B}{RTT + L/B})$$

$$U_{SRN} \approx \frac{10 * 10^3 * 8/10^7}{0.1 + 10^3 * 8/10^7} \approx 0.08$$

TCP

Reliability in TCP

- Purpose
  - Provide reliable transmission over datagram channel
- Solution
  - Go-Back-N with cumulative ACK and receiver side buffering
    - does not discard received data (if not duplicate)



TCP

Retransmission Timer

- Retransmission Time-Out (RTO)
  - Time to wait for the ACK of a segment
  - Cannot be fixed
    - for all connections (e.g., different RTTs)
    - for all segments in a connection (e.g., varying network delay)
- RTO calculated dynamically
  - Based on the measured round-trip time (RTT)
- RTO updated after retransmission
  - Binary Exponential backoff
    - Double the RTO for each retransmission
    - Required for stability (not in RFC 793, but in RFC2988)

TCP

(Wrong) Calculation of the RTO

- Original proposal (RFC 793)
  - Measure *RTT (mRTT[k])*
  - Exponential smoothed average (a=0.9)

$$sRTT[k+1] = a\ sRTT[k] + (1-a)\ mRTT[k]$$

  - RTO = 2 * sRTT[k+1]
    - Works up to 30% link load – above it RTT varies more!
    - Underestimating the RTT causes unnecessary retransmissions!
- This is one way how you should not do it!

Calculation of the RTO

- Method used today (RFC 2988, Van Jacobson '88)
  - Update estimated RTT variation

$$RTTvar[k+1] = (1-\beta)RTTvar[k] + \beta\,|sRTT[k] - mRTT[k]|$$

  - Update estimated RTT mean

$$sRTT[k+1] = \alpha\ sRTT[k] + (1-\alpha)\ mRTT[k]$$

  - Update RTO based on estimated mean *and* variation

$$RTO = sRTT[k+1] + 4RTTvar[k+1]$$

- How do you measure *mRTT*?
  - Time between segment is sent and ACK is received (?)

Karn's Algorithm

- Scenario
  - Segment not ACKed and is retransmitted
  - When ACK is received the sender does not know if
    - ACK was sent for the original segment
    - ACK was sent for the retransmitted segment
- Solutions
  - Use the TCP timestamp option (see later)
  - Karn's algorithm
    - Don't consider the RTT of a retransmitted segment
    - Don't update RTT value until you get an ACK without need for retransmission

Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", in Proc. of ACM SIGCOMM 87.

TCP functions

- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Retransmission scheme
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

TCP functions

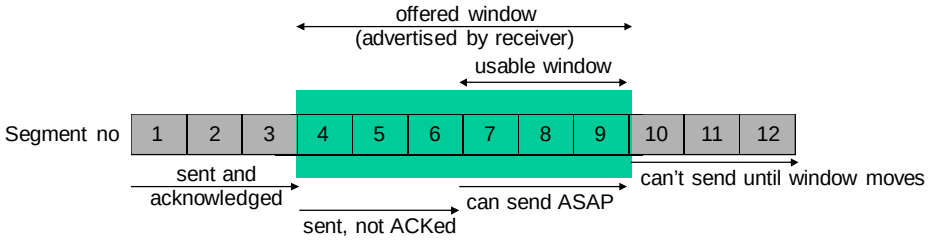
- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Retransmission scheme
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

Flow Control in TCP

- Ensure that receiver does not get overwhelmed with data sent by the sender
  - Control the amount of data the sender can send before receiving an acknowledgement from the receiver
  - Should not be too slow
    - do not let sender send 1 byte and wait for acknowledgement
  - In-band vs. Out-of-band
- TCP uses a *sliding window* protocol
  - Sending and the receiving TCP use the window to control the amount of unacknowledged data in the network
  - For each TCP connection in each direction (duplex)

TCP Sliding Windows

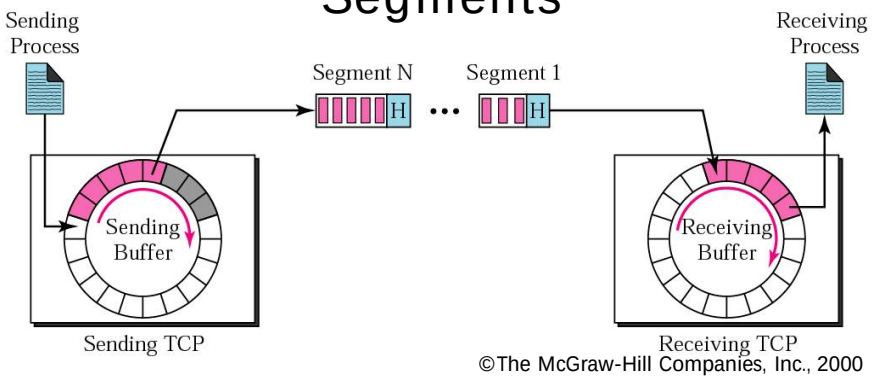
- Receiver: *offered window* (sent with ACK)
  - data that it can receive beyond what it ACKed
  - can send an ACK with an offered window of 0!
    - later it sends *window update* with offered window size > 0
- Sender: *usable window*
  - data it can send immediately



Persistence Timer

- Scenario:
  - Receiver announces RWND=0 ⇒ sender stops sending and waits for an ACK with RWND>0
  - The ACK gets lost ⇒ deadlock
    - Sender waits for ACK and RWND>0
    - Receiver waits for data
- Solution:
  - Sending TCP sets *persistence timer*
  - Periodically sends *window probes* to find out if window size has increased, e.g., every 60 seconds

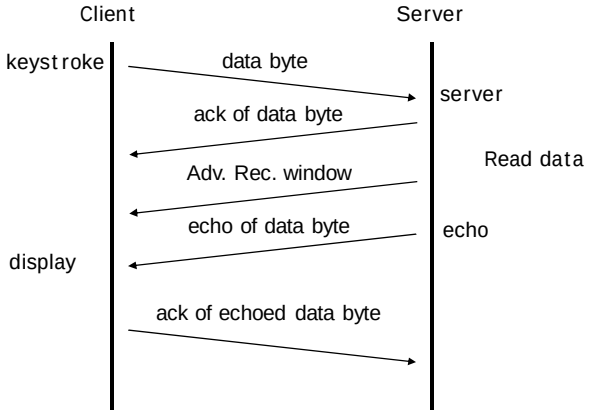
Byte streams, Buffers, and Segments



- White area: empty locations, ready to be filled
- Gray area: bytes that have been sent but not ACKed
- Colored area:
  - bytes to be sent by the sending TCP
  - bytes to be delivered by the receiving TCP to the receiving process

TCP Interactive Data Flow

In an *ssh* session, each individual keystroke can generate a packet



Delayed ACKs

- Problem
  - Too many ACKs sent
- Solution
  - Do not send ACK immediately (Host Requirements RFC1122)
  - Send at least one ACK per two full sized segments (MSS)
  - The delay must be less than 500ms
  - Typically wait up to ~200ms before sending ACK
- Advantages:
  - ACK traffic is reduced
    - Less processing in sender!
    - Less network traffic
  - Increased chance that data can be piggy-backed on the ACK
- Important
  - Retransmission timer should not go off because of delayed ACKs

Silly window sindrome
Sender initiated

- Problem: Slow sender application
  - E.g., 1 byte application layer data sent for each keystroke
  - 1 datagram (41 bytes) for each 1 byte of user data ("tinygram")
    - not a problem on LANs
    - adds to the congestion on WANs
- Solution: Nagle's Algorithm
  - TCP connection can have only one unacknowledged tinygram
    - Data are accumulated while waiting
    - Sent as one segment when the ACK arrives, or when maximum segment size can be filled
  - The faster ACKs come, the more often data are sent
    - On slow WANs fewer segments are sent than on fast LANs
    - Rarely invoked on a LAN
      - Round trip time (RTT) ~1ms - to generate data faster than this would require typing faster than 60 characters per second!

Silly window sindrome
Receiver initiated

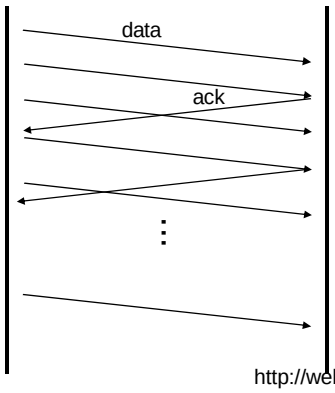
- Problem: Slow receiver application
  - Sender sends big segments (e.g., 1KB)
  - Receiving application consumes data slowly (e.g., 1 byte at a time)
  - Receiving TCP buffer gets full
  - Receiver window size announced for each byte read from the buffer
- Solution: Clark's solution
  - Do not advertise small increments of the rwnd size
  - Increment at least by MSS or ½ receiver buffer size

TCP Bulk Data Flow

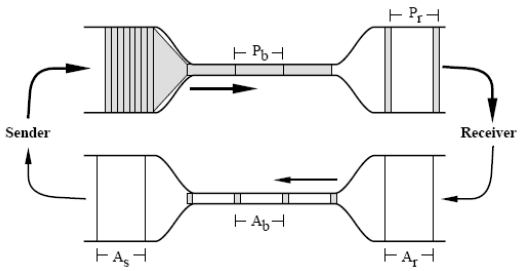
Data transfer: full MSS

Typically: ACK every other segment

Self-clocking



Self-clocking



Source: V. Jacobson, "Congestion avoidance and control", In Proc. of SIGCOMM '88

http://web.eecs.utk.edu/~dunigan/tcptour/javis/tcp_slidwin.html

Bandwidth-Delay Product

- The ”capacity” of the ”pipe”
  capacity(bits) = bandwidth(bits/sec) x RTT(sec)
- The receiver advertised window should be higher...

Example:
Connection across the US:
T1: capacity = 1.544Mbit/s x 60ms ≈ 11,5 KB
STM-4: capacity = 622Mbit/s x 60ms ≈ 4.66 MB

- TCP limitation: ”window size” field 16 bits → Max 65535 bytes
  - ”window scale” option for ”Long Fat Pipes”

Notes on Flow Control

Some points about TCP’s sliding windows

- The src does not have to send data worth a full window
- Window size can be increased/decreased by the receiver
  - But the sender’s usable window should not shrink
- The receiver can send an ACK at any time

- The sender window size is controlled by the receiver window size
- Typical sizes of the receiver buffer are 4K, 8K, or 16K
  - Can be changed through the programming interface (e.g., the *socket API*)
- The *actual* window size can be smaller than the *receiver window size* if there is *congestion* in the network

TCP functions

- Multiplexing/demultiplexing
- Segmentation
  - Byte stream to segment translation
- Error control
  - Retransmission scheme
- Flow control
  - Adapt to the receiver's capabilities
- Connection Management
  - Establishment/tear down
- Congestion control
  - Adapt to network conditions

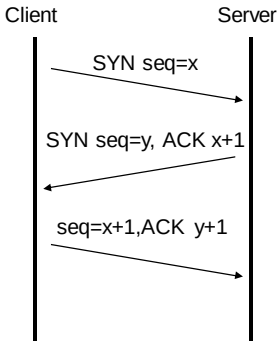
Vanilla Connection Establishment

- Design a connection establishment protocol…
  - …over a datagram network layer protocol
  - …agree on sequence numbers

C.A. Sunshine, Y.K. Dalal, "Connection Management in Transport Protocols", Computer Networks, Vol 2, 1978, pp. 454-473

TCP Connection Establishment

TCP uses a three-way handshake to establish a connection



3-way handshake

3-way handshake:

- Guarantees both sides ready to transfer data
- Allows both sides to agree on initial sequence numbers, MSS, window size

Initial sequence number (ISN) must be chosen so that each incarnation of a specific TCP connection between two end-points has a different ISN.

Normally, client performs *active open*, server performs *passive ope*
Alternative: simultaneous open

How can you attack this?

Keepalive Timer

Avoid TCP connections to exist forever

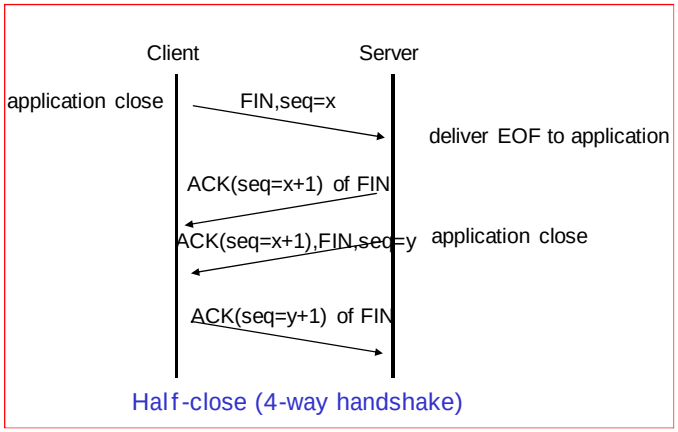
- Scenario
  - Client connects to server, transmits data, then crashes
  - TCP connection would continue to exist
    - Consumes server resources!
- Solution
  - TCP gets a keepalive time-out after some time (typical: 2 hours)
  - After keepalive time-out, server probes the client
    - E.g. 10 probes, each 75s
  - If client does not respond, the connection is torn down
    - Reset (RST)

Vanilla Connection Teardown

- Design a connection teardown protocol…
  - …over a datagram network layer protocol

C.A. Sunshine, Y.K. Dalal, "Connection Management in Transport Protocols", Computer Networks, Vol 2, 1978, pp. 454-473

TCP Connection Teardown



Half-close (4-way handshake)

4 segments needed to terminate a connection

3-way handshake is allowed as well

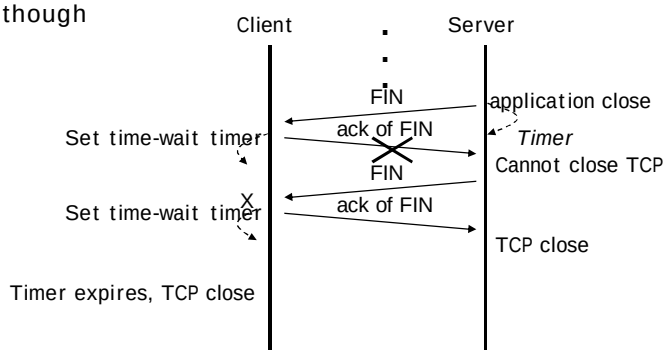
Simultaneous close looks the same

Normally, client performs *active close* and server performs *passive close*

Time-Wait Timer

Connection termination

- The connection is held in TIME-WAIT state
  - allows duplicate FIN segments to arrive at the destination (send ACK)
- TWT is usually twice the maximum lifetime of a segment
  - Maximum Segment Lifetime = 0,5...2 mins (←IP TTL limit)
- Port should not be reused within this time – might be reopened though



TCP functions

- Multiplexing/demultiplexing
- Error control
  - Retransmission scheme
- Connection Management
  - Establishment/tear down
- Segmentation
  - Byte stream to segment translation
- Flow control
  - Adapt to the receiver's capabilities
- Congestion control
  - Adapt to network conditions

Why congestion control?

- Number of hosts connected to the Internet
  - Aug.1981: 213 Oct.1990: 313 thnd Jul.2009: ~681 million
- Number of users of the Internet
  - 1990: ~2 million 2011: ~2.3 billion

http://www.isc.org
http://www.internetworldstats.com

- Internet is a shared resource with limited capacity
  - Tragedy of the commons?
  - Need for resource sharing
    - Fairness

Congestion

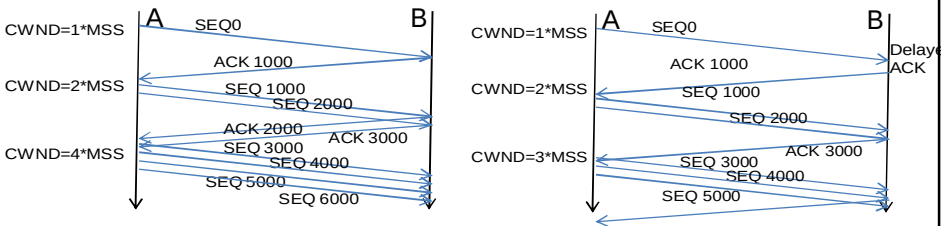
- Assumption until now
  - Receiver is only bottleneck
  - Network can deliver the data as fast as sent by the sender
- Assumption is invalid
  - If router receives packets faster than it can process
    - packets get dropped (congestion)
    - packets have to be stored in a buffer to decrease drop rate
      - EP2200 - Queueing theory
  - Delay and missing ACKs cause retransmission
  - Retransmission will add to congestion → network collapses
    - Congestion collapse (e.g., Oct 86 : 40kbps → 40 bps, 400yard link)
- The *window size* must depend on the *network's* state as well!
  - rfc793 does not include congestion control ☺

TCP Congestion Control

- Origin
  - Introduced by Van Jacobson (1988)
  - Based on analysis of actual traffic, and control theory
- Congestion Window maintained by the sender
  - Updated in response to losses and/or delay
- Sender maintains 2 window sizes:
  - Receiver-advertised window (RWND)
  - Congestion window (CWND)
- Actual window size = min(RWND, CWND)
- Two phases
  - Slow start
  - Congestion avoidance → Additive increase Multiplicative decrease (AIMD)

Slow Start phase (rfc5681)

- All TCP implementations *must* implement slow start
- Operation
  - At beginning of connection, CWND ⇐ 1 MSS
  - For each ACK: `CWND += min(N,MSS)` rfc5681
    - N is the new data (in bytes) acknowledged by ACK
- Not slow: window size grows exponentially
  - Note the interaction with delayed ACKs...!

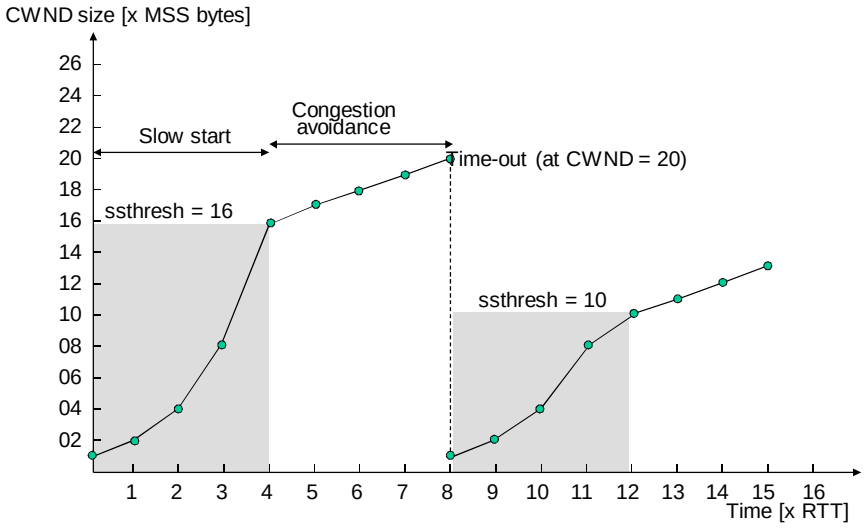


- Stop using *slow start* when CWND > *ssthresh*

Congestion Avoidance phase

- Slower than exponential growth
  - Increase CWND additively
    - On every acked byte CWND+=MSS/CWND (+1 MSS per RTT)
  - Linear growth continues until
    - Receiver-advertised window size is reached, or
    - ACK timeout occurs → TCP assumes congestion
- When congestion occurs (TCP Tahoe)
  - *ssthresh* = ½ CWND
  - CWND = 1 MSS ⇒Enter slow-start phase
- *Choice of ssthresh*
  - Initial value 65535 bytes
  - Minimum value 512 bytes

Slow-start and Congestion Avoidance



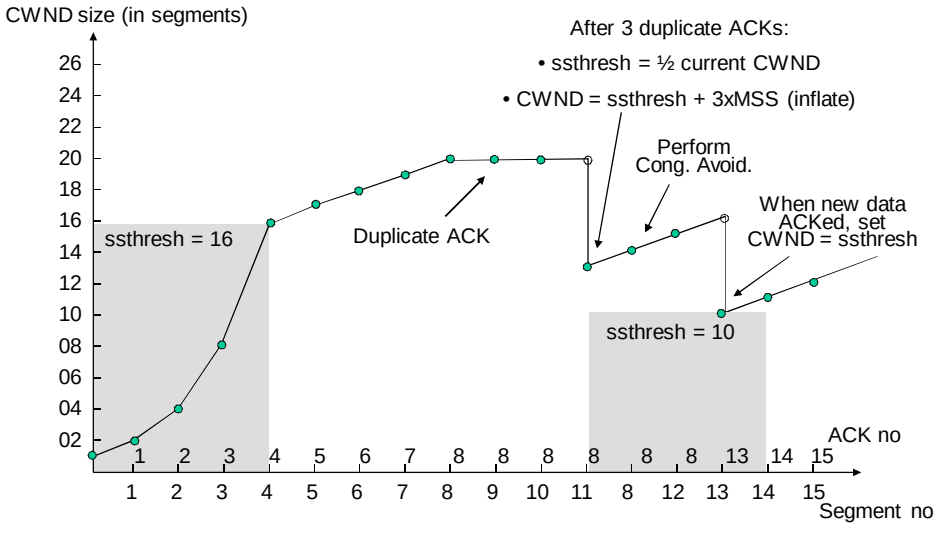
Fast Retransmit and Fast Recovery

- Problem
  - Out of order segments can trigger *slow start*
  - Loss of one segment triggers slow start
- Solution (TCP Reno)
  - Generate immediate ACK (a duplicate ACK) when an out-of-order segment is received
    - lost segment, or
    - reordering of segments

    (but data is still flowing between the two ends!!)
  - If 3 (or more) duplicate ACKs are received in a row:
    - Retransmit immediately (before time-out) - *Fast Retransmit*
    - Do congestion avoidance (*not* slow start) - *Fast Recovery*

Rfc 2581, originallly V. Jacobson, "Modified TCP Congestion Avoidance Algorithm," end2end-interest mailing list, April 30, 1990. ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail.

Fast Retransmit and Recovery cont'd



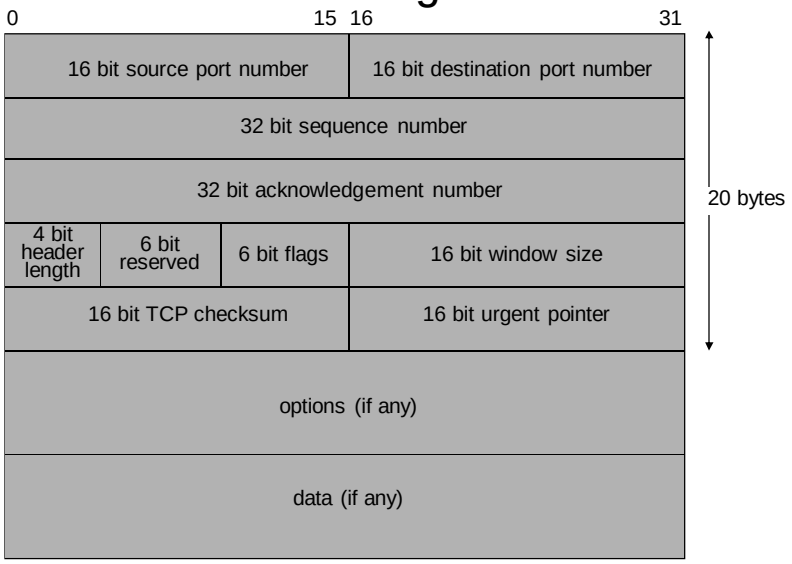
TCP Flavors
Different congestion control algorithms...

- Tahoe: Van Jacobson, 1988
  - Congestion avoidance and control
- Reno: Van Jacobson, 1990
  - Modified congestion avoidance algorithm
  - Fast retransmit and fast recovery
  - Reactive algorithm (loss based)
- Vegas: Brakmo, Peterson, 1994
  - New techniques for congestion detection and avoidance
  - Proactive algorithm (delay measurement based)
- Westwood: Casetti et. al, 2000
  - Adaptation to high-speed wired/wireless environment
  - Deals with large bandwidth-delay product and packet loss (leaky pipes)
- And a few others…probably more coming…

TCP functions

- Multiplexing/demultiplexing
- Error control
  - Retransmission scheme
- Connection Management
  - Establishment/tear down
- Segmentation
  - Byte stream to segment translation
- Flow control
  - Adapt to the receiver's capabilities
- Congestion control
  - Adapt to network conditions

TCP Segment



TCP Header Fields

- *Src and Dst port numbers (16 bits)*
  - Identify sending/receiving process.
  - Identify the TCP connection together with src/dst IP addresses
    - called *socket pair*
- *Sequence number* (32 bits)
  - Sequence number of first byte of this segment (Wraps around at 2^{32}-1)
- *Acknowledgement number* (32 bits)
  - Next sequence number that the receiver (sender of the ACK) is ready to receive
- *Header length*
  - Length of the header in 32-bit words, required field bcs options field is variable.
  - Max header length is 60 bytes ($4*(2^4 - 1) = 60$)
- *Reserved*
  - for future use

TCP Header Fields cont'd

- *Flags* (8 bits)
  - Determine the purpose and vallid contents of the segment (e.g., SYN,FIN,ACK, ,...)
- *Window size*
  - The number of bytes that the receiver is willing to accept
  - Starts with the byte specified in the acknowledgement number
- *TCP Checksum*
  - Covers header and data, uses pseudo header (like in UDP)
  - Mandatory (unlike in UDP)
- *Urgent pointer*
  - Location of emergency data in the payload (e.g., end of session)
- *Options*
  - Used by TCP to negotiate certain features between the end-points (e.g., maximum segment size (MSS), window scale, SACK)

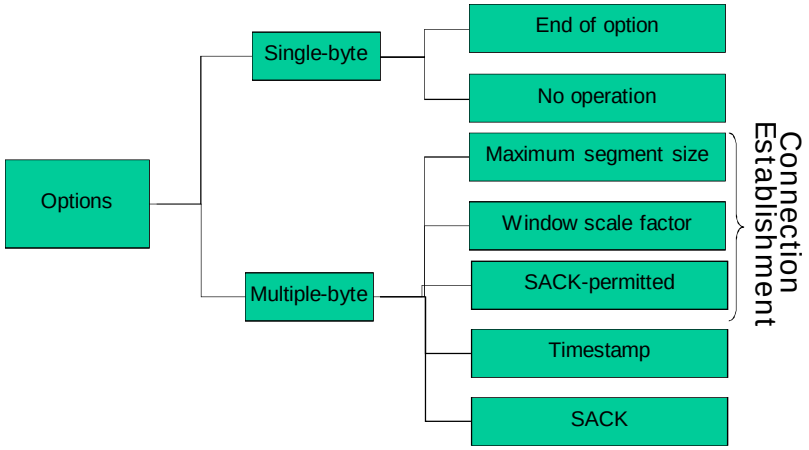
TCP Flags

TCP header contains 8 flag bits

- URG - urgent pointer valid
  - set when sender wants the receiver to read a piece of data urgently and possibly out of order
- PSH – push data to application
  - receiver should immediately pass the data to the application (buffers should be emptied, note the difference compared to URG)
- ACK - acknowledgement number valid
- RST - reset the connection
- SYN - synchronize sequence numbers to initiate connection
- FIN - sender has finished sending data
- ECE, CWR (rfc3168) – support for explicit congestion notification

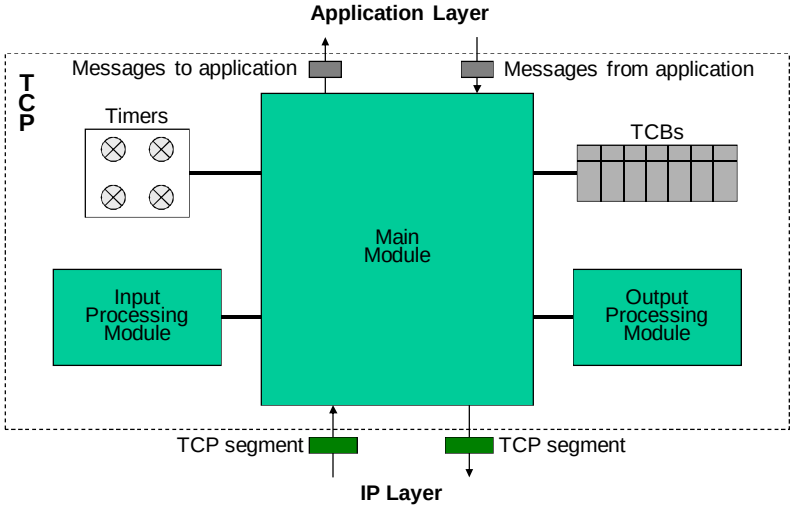
TCP Options

- Up to 40 bytes long
- Used to convey additional information to the destination

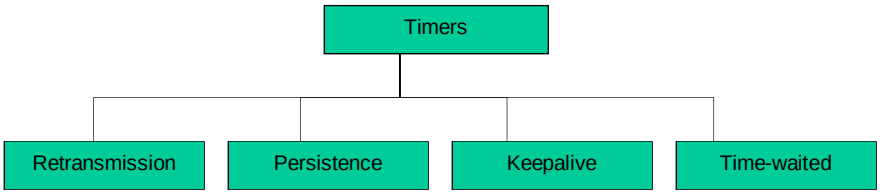


TCP Package

Very complex protocol, tens of thousands of lines of code!



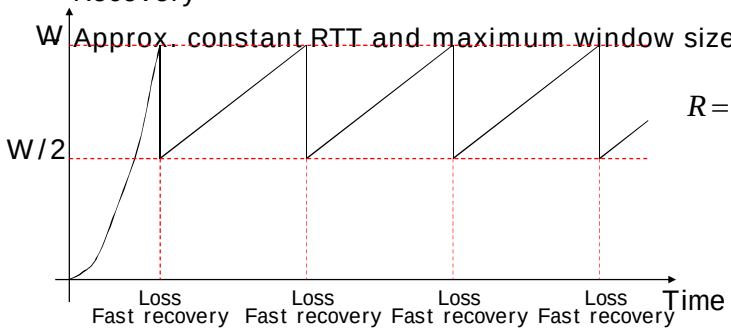
Timers



TCP uses four timers, already discussed

TCP Throughput Model

- Very simple approximation
  - Congestion avoidance phase only, using Fast Recovery



Approx. constant RTT and maximum window size (W)

$$R = MSS \frac{3}{4} \frac{W}{RTT}$$

TCP Throughput

- Many models around – active field of research
  - Padhye et al. TCP Reno, approximate formula, packet rate

$$B(p) = \min\left(\frac{W}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0\left(\min\left(1, 3\sqrt{\frac{3bp}{8}}\right) p(1+32p^2)\right)}\right)$$

    - RTT: round trip time
    - b: delayed ACKs
    - p: ~loss probability
    - T_0: RTO

$$B(p) = \frac{1}{RTT}\sqrt{\frac{3}{2bp}} + o(1/\sqrt{p})$$

- Fairness?

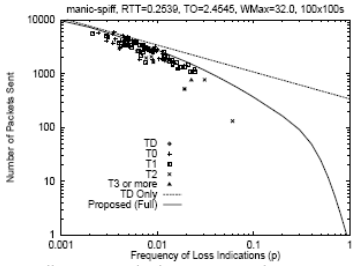
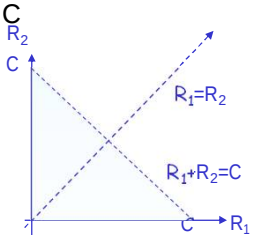
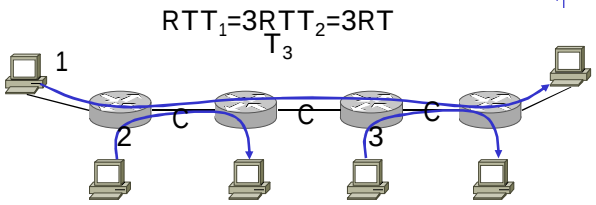


Fig 15 in J. Padhye, V. Firoiu, D. Towsley, J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", in Proc. ACM SIGCOMM 98

Fairness and TCP

- Allocation of the bandwidth among competing flows
- TCP fairness
  - Identical flows on a link with capacity C
    - Equal allocation
  - More complex scenarios
    - What is fair?





$$\frac{R_1}{R_2} \approx \frac{RTT_2}{RTT_1}$$

$$R_1 = R_2$$

TCP over wireless

- Problem
  - TCP interprets loss as a sign of congestion
  - In wireless networks loss can be due to
    - Fading, handover, shadowing, interference + congestion
- Solutions
  - e2e: classify loss events based on loss patterns
  - Proxy for TCP
  - Link layer retransmissions

TCP Summary

- TCP service
  - Reliable full-duplex byte stream service
- Very complex protocol
  - Connection Management
  - Reliability
  - Segmentation
  - Flow control
  - Congestion control

Other Transport Layer Protocols

There are other transport layer protocols than UDP and TCP! (remember UDP-Lite?)

- SCTP (Stream Control Transmission Protocol)
  - Reliable, congestion control (like TCP)
  - Message-based
  - Multiple streams in a connection
  - Multi-homing: more than one pair of end-points
  - Typically used for signaling over IP
- DCCP (Datagram Congestion Control Protocol)
  - Unreliable, message-based (like UDP)
  - Congestion control - TFRC
  - Flow-based semantics of TCP, but timely delivery instead of in-order delivery and reliability
  - Suitable for applications such as streaming media

Transport Layer Summary

Protocol/Service or function	UDP	DCCP	SCTP	TCP
Stream/Message oriented	Message	Message	Message	Bytestream
Reliable	No	No	Yes	Yes
Flow control	No	No	Yes	Yes
In-order delivery	No	No	Yes	Yes
Congestion control	No	Yes	Yes	Yes
Connection oriented	No	Yes	Yes	Yes
Multiple streams	-	-	Yes	No

UDP, TCP