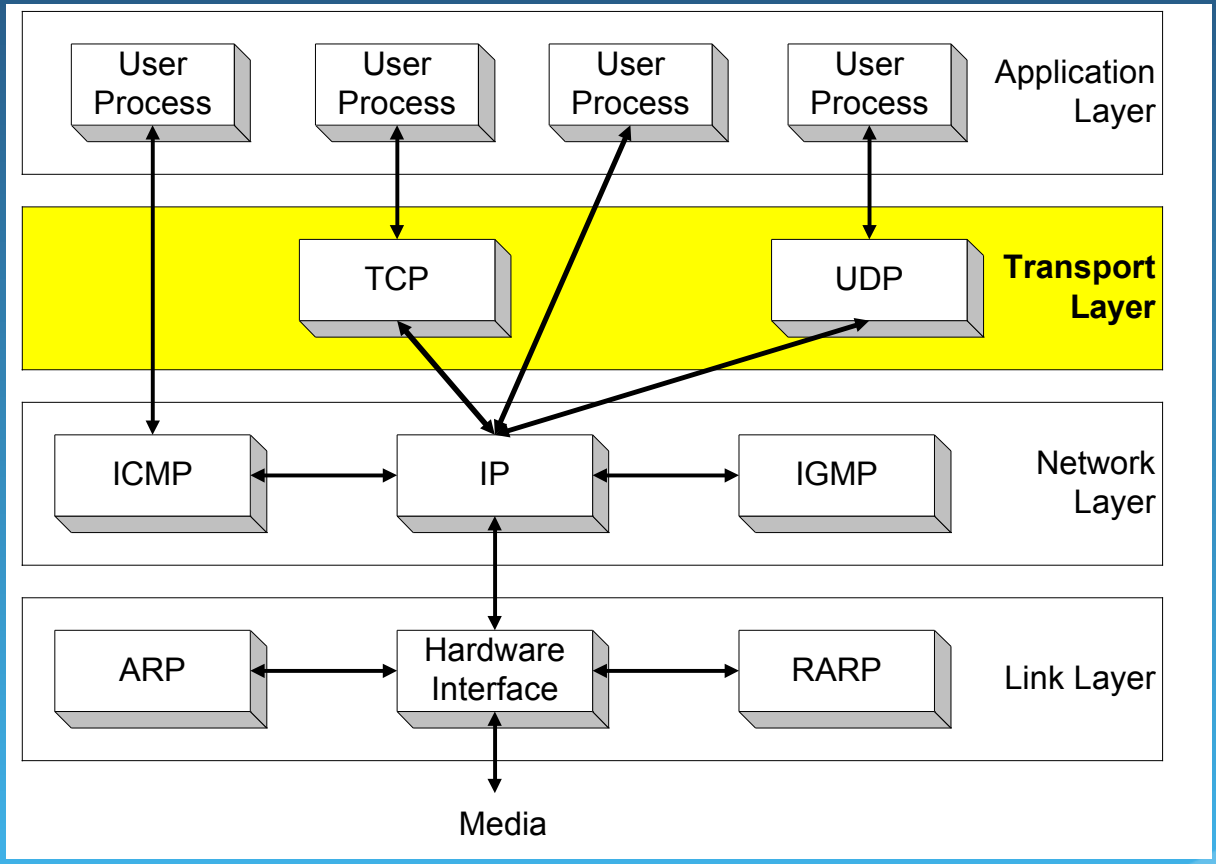
Chapter 5: Transport Protocols

Magda El Zarki
Professor of CS
Univ. of CA, Irvine
Email: elzarki@uci.edu
http://www.ics.uciedu/~magda

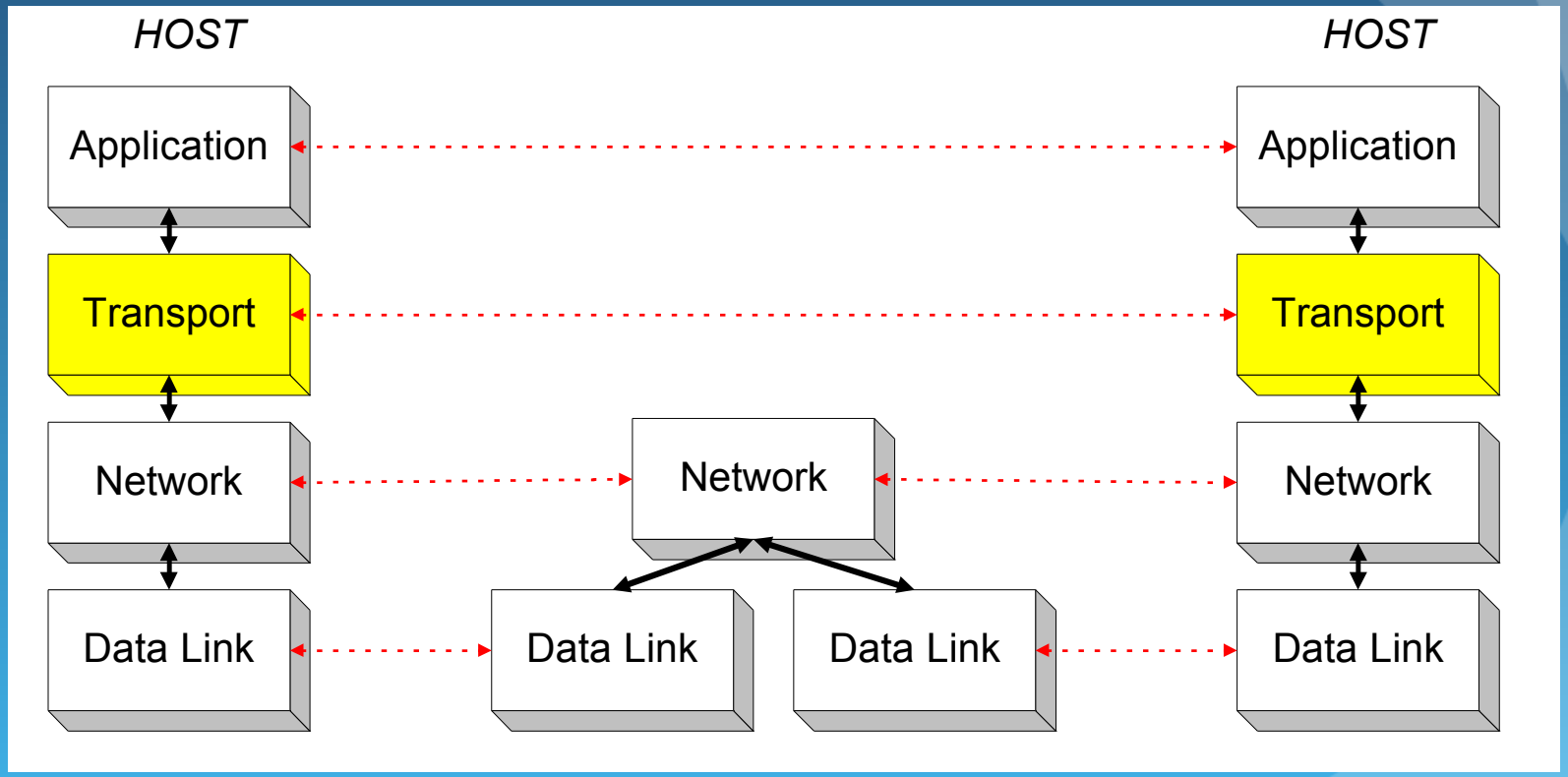
Orientation

- We move one layer up and look at the transport layer.



Overview

- Transport layer protocols are end-to-end protocols
- They are only implemented at the hosts



Transport Protocols in the Internet

- The Internet supports 2 transport protocols

UDP - User Datagram Protocol

- datagram oriented
- unreliable, connectionless
- simple
- unicast and multicast
- useful only for few applications, e.g., multimedia applications
- used a lot for services
  - network management (SNMP), routing (RIP), naming (DNS), etc.

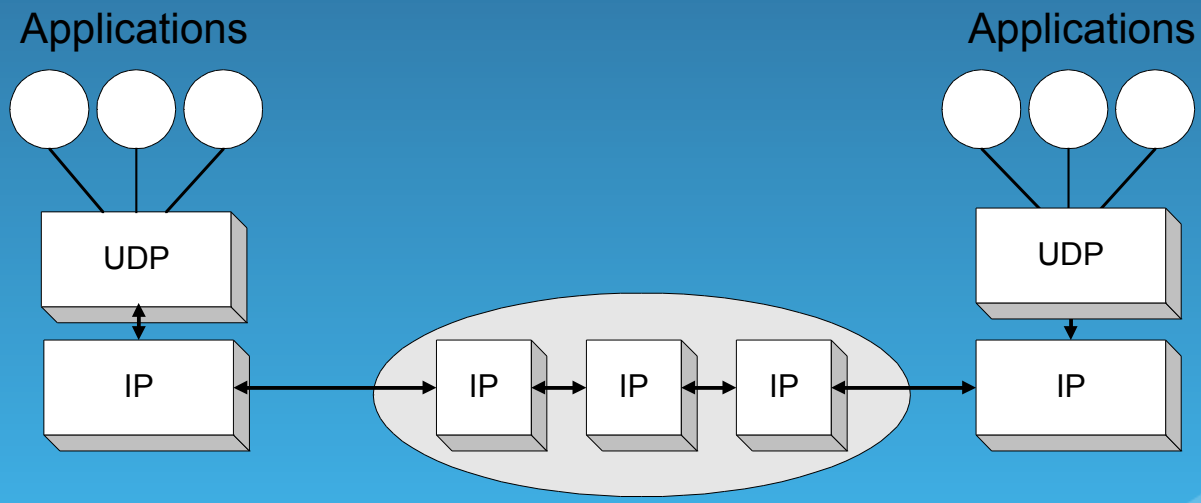
TCP - Transmission Control Protocol

- stream oriented
- reliable, connection-oriented
- complex
- only unicast
- used for most Internet applications:
  - web (http), email (smtp), file transfer (ftp), terminal (telnet), etc.

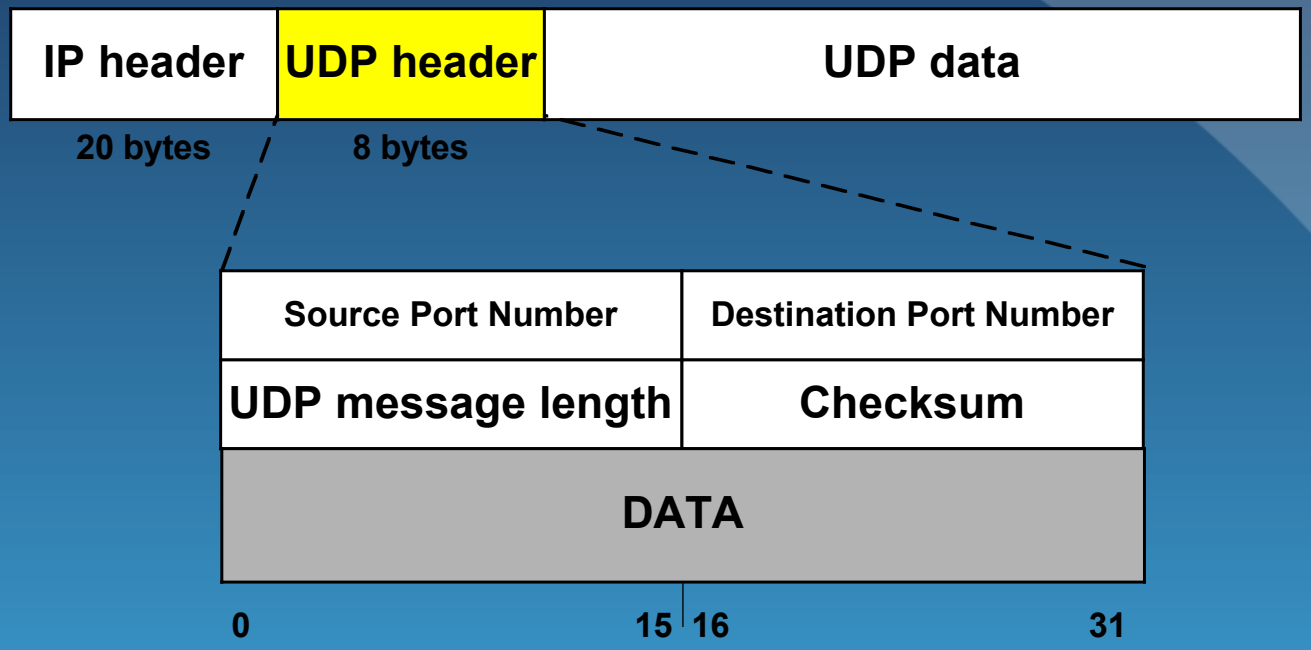
UDP – User Datagram Protocol

Overview

- UDP supports unreliable transmissions of datagrams
- UDP merely extends the host-to-to-host delivery service of IP datagram to an application-to-application service
- The only thing that UDP adds is multiplexing and demultiplexing



UDP Format



- **Port numbers** identify sending and receiving applications (processes). Maximum port number is 2^{16}-1= 65,535

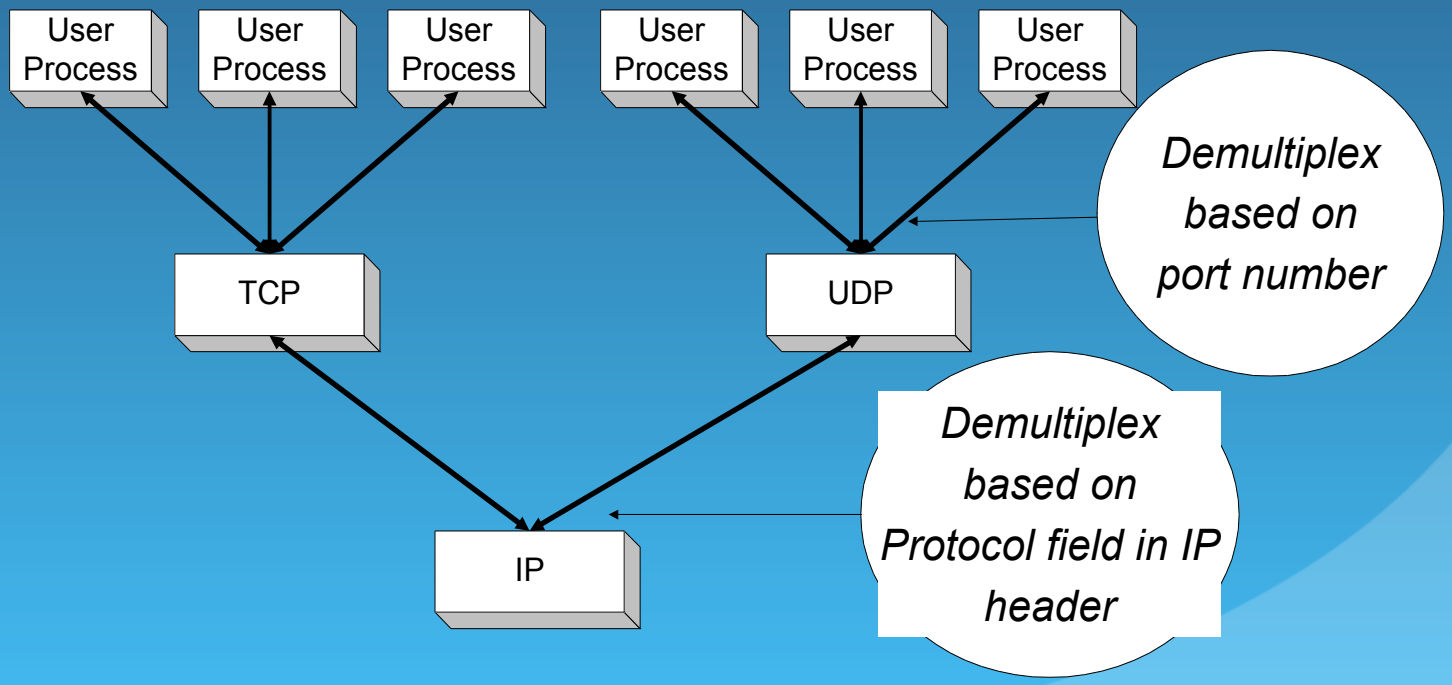
- **Message Length** is at least 8 bytes (I.e., Data field can be empty) and at most 65,535

UDP Checksum

- UDP checksum computation is optional for IPv4. If a checksum is not used it should be set to the value zero.
- If used, the checksum is computed using a PSEUDO-HEADER that contains some of the same information from the real IPv4 header. The PSEUDO-HEADER is not the real IPv4 header used to send an IP packet. The following table defines the PSEUDO-HEADER used only for the checksum calculation.
  - IP Source address
  - IP Destination address
  - Protocol Type: UDP in this case
  - UDP length – header and data
  - Source Port & Destination Port
  - UDP Length
  - Data
- The source and destination addresses are those in the IPv4 header. The protocol is that for UDP (see List of IP protocol numbers): 17 (0x11). The UDP length field is the length of the UDP header and data.

Port Numbers

- UDP (and TCP) use port numbers to identify applications
- A globally unique address at the transport layer (for both UDP and TCP) is a tuple **<IP address, port number>**
- There are 65,535 UDP ports per host.

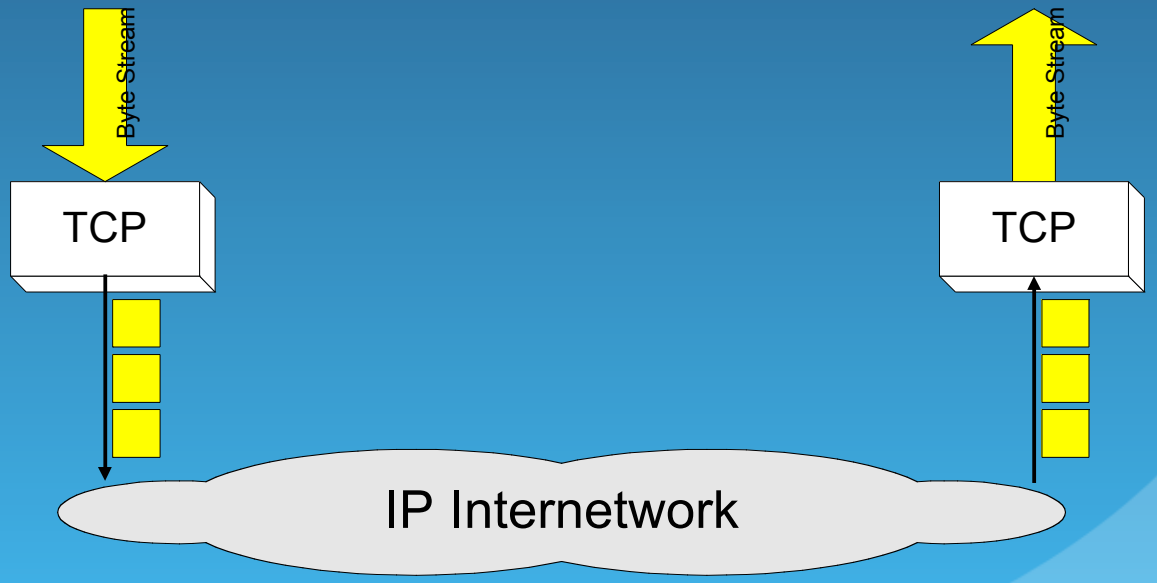


Transport Protocol - TCP

Overview

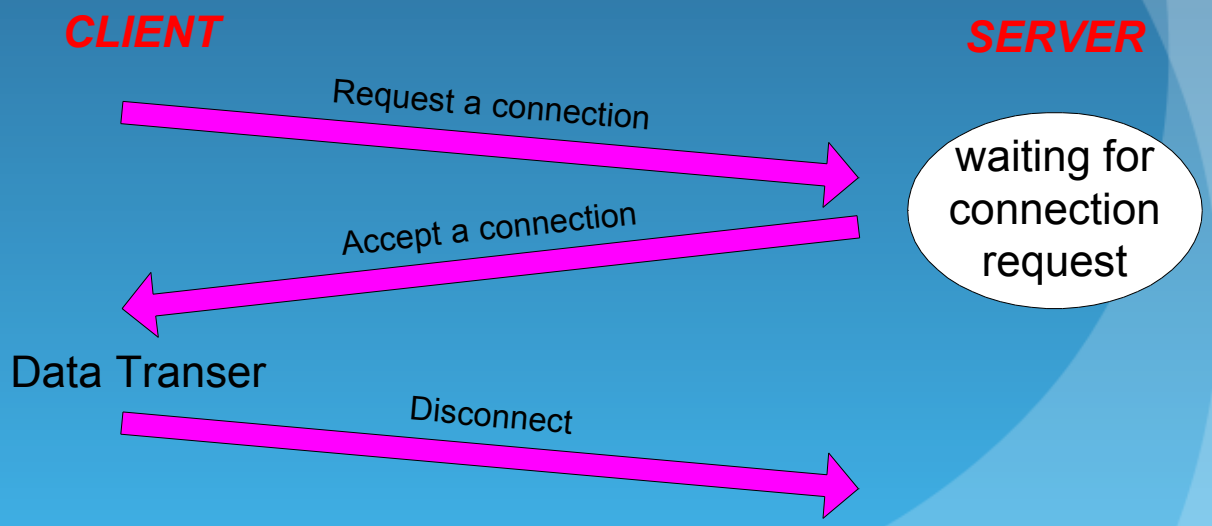
TCP = Transmission Control Protocol

- Connection-oriented protocol
- Provides a reliable unicast end-to-end byte stream over an unreliable internetwork.



Connection-Oriented

- Before any data transfer, TCP establishes a **connection**:
  - One TCP entity is waiting for a connection (“**server**”)
  - The other TCP entity (“**client**”) contacts the server
- The actual procedure for setting up connections is more complex.
- Each connection is full duplex

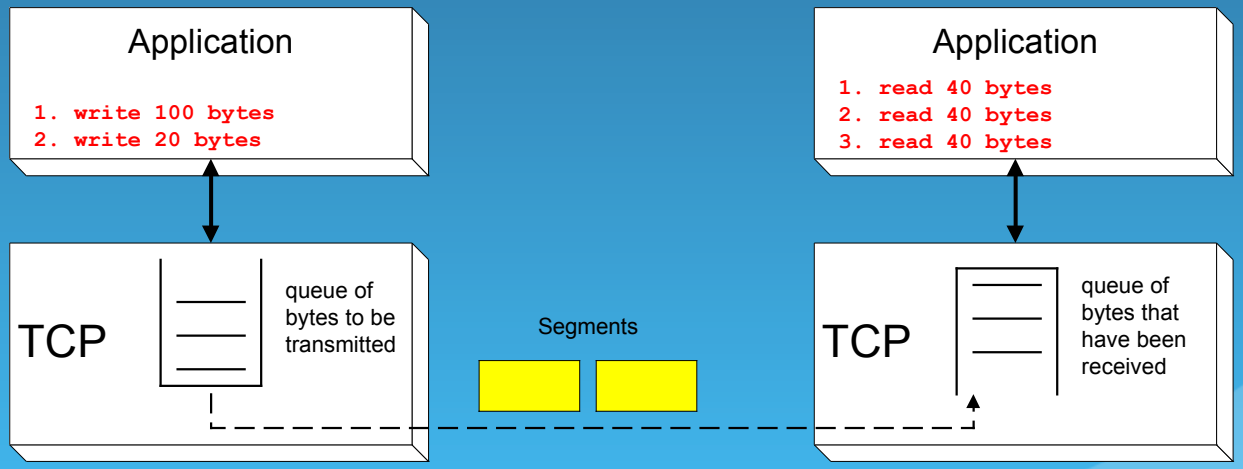


Reliable

- Byte stream is broken up into chunks which are called **segments**
  - Receiver sends acknowledgements (ACKs) for segments
  - TCP maintains a timer. If an ACK is not received in time, the segment is retransmitted
- **Detecting errors:**
  - TCP has checksums for header and data. Segments with invalid checksums are discarded
  - Each byte that is transmitted has a sequence number

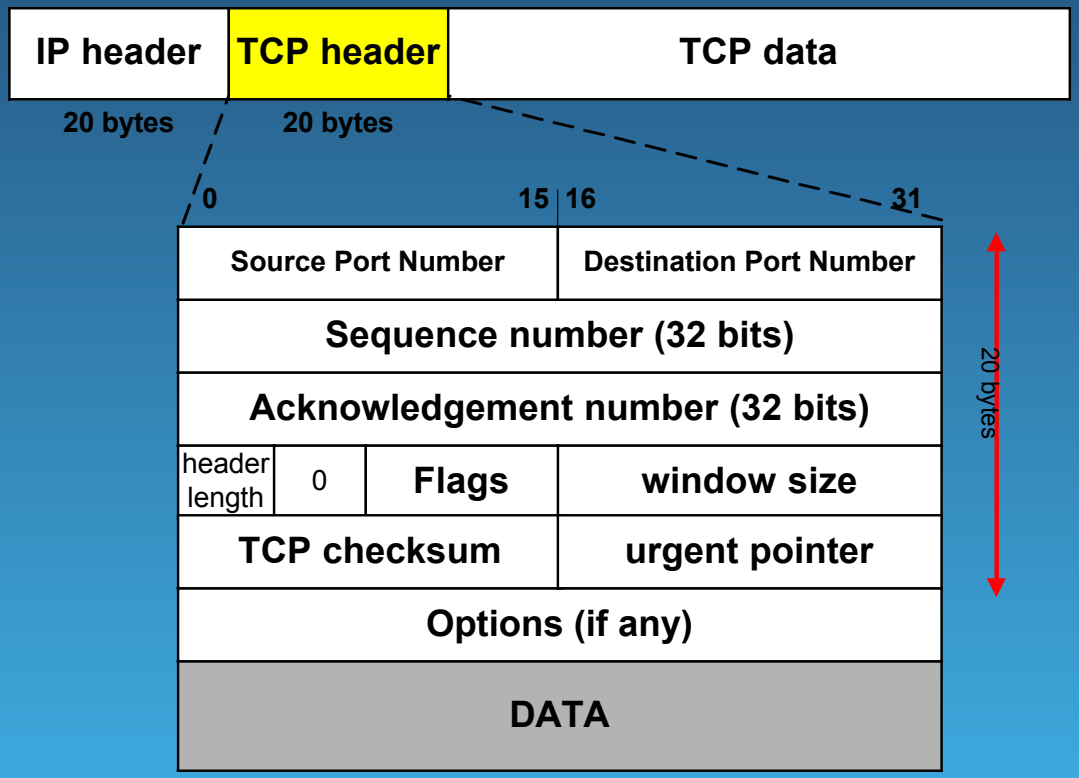
Byte Stream Service

- To the lower layers, TCP handles data in blocks - segments.
- To the higher layers TCP handles data as a sequence of bytes and does not identify boundaries between bytes
- So: Higher layers do not know about the beginning and end of segments.



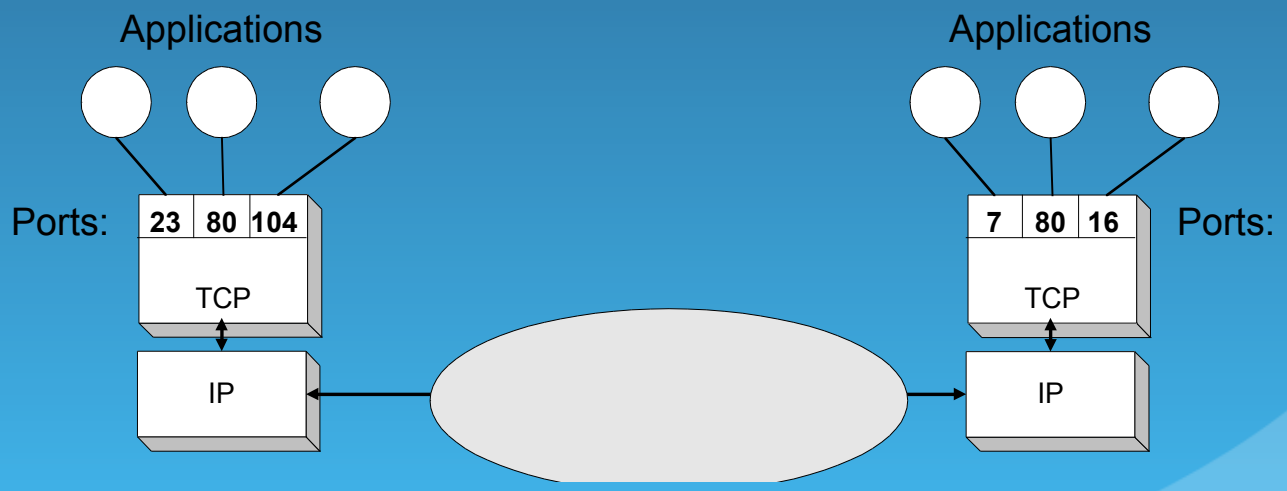
TCP Format

- TCP segments have a 20 byte header with >= 0 bytes of data.



TCP header fields

- **Port Number:**
  - A port number identifies the endpoint of a connection.
  - A pair **<IP address, port number>** identifies one endpoint of a connection.
  - Two pairs **<client IP address, client port number>** and **<server IP address, server port number>** identify a TCP connection.



TCP header fields

- **Sequence Number (SeqNo):**
  - Sequence number is 32 bits long.
  - So the range of SeqNo is

    $0 <= SeqNo <= 2^{32} - 1 \approx 4.3$ Gbyte
  - Each sequence number identifies a byte in the byte stream
  - Initial Sequence Number (ISN) of a connection is set during connection establishment

TCP header fields

- **Acknowledgement Number (AckNo):**
  - Acknowledgements can be piggybacked, i.e

    a segment from A -> B can contain an acknowledgement for a data sent in the B -> A direction
  - A hosts uses the AckNo field to send acknowledgements. (If a host sends an AckNo in a segment it sets the "**ACK flag**")
  - The AckNo contains the ***next*** SeqNo that a hosts is expecting to receive

    Example: The acknowledgement for a segment with sequence numbers 0-1500 is AckNo=1501

TCP header fields

- **Acknowledge Number (cont'd)**
  - TCP uses the sliding window flow protocol to regulate the flow of traffic from sender to receiver
  - TCP uses the following variation of sliding window:
    - no NACKs (**N**egative **ACK**nowledgement)
    - only cumulative ACKs
- Example:

  Assume: Sender sends two segments with "1..1500" and "1501..3000", but receiver only gets the second segment.

  In this case, the receiver cannot acknowledge the second packet. It can only send AckNo=1

TCP header fields

- **Header Length (4bits):**
  - Length of header in 32-bit words
  - Note that TCP header has variable length (with minimum 20 bytes)

TCP header fields

- **Flag bits:**
  - **URG: Urgent pointer is valid**
    - If the bit is set, the following bytes contain an urgent message in the range:

      SeqNo <= urgent message <= SeqNo+urgent pointer
  - **ACK: Acknowledgement Number is valid**
  - **PSH: PUSH Flag**
    - Notification from sender to the receiver that the receiver should pass all data that it has to the application.
    - Normally set by sender when the sender's buffer is empty – indicates flush your buffer, I am done with data for now.

TCP header fields

- **Flag bits:**
  - **RST: Reset the connection**
    - The flag causes the receiver to reset the connection
    - Receiver of a RST terminates the connection and indicates higher layer application about the reset
  - **SYN: Synchronize sequence numbers**
    - Sent in the first packet when initiating a connection
  - **FIN: Sender is finished with sending**
    - Used for closing a connection
    - Both sides of a connection must send a **FIN**

TCP header fields

- **Window Size:**
  - Each side of the connection advertises the window size
  - Window size is the maximum number of bytes that a receiver can accept.
  - Maximum window size is 2^{16}-1= 65535 bytes
- **TCP Checksum:**
  - TCP checksum covers both TCP header **and** TCP data (also covers some parts of the IP header)
    - The TCP checksum is a 16 bit 1's complement sum of all the 16 bit words in the TCP header plus the IP source and destination address values, the protocol value (6) and the length of the TCP segment (header + text).
- **Urgent Pointer:**
  - Only valid if **URG** flag is set

TCP header fields

- **Options**:
  - **NOP** is used to pad TCP header to multiples of 4 bytes
  - **Maximum Segment Size**
  - **Window Scale Options**
    - Increases the TCP window from 16 to 32 bits
    - This option can only be used in the SYN segment (first segment) during connection establishment time
  - **Timestamp Option**
    - Can be used for roundtrip measurements

Connection Management in TCP

- **Opening a TCP Connection**
- **Closing a TCP Connection**
- **Special Scenarios**
- **State Diagram**

TCP Connection Establishment

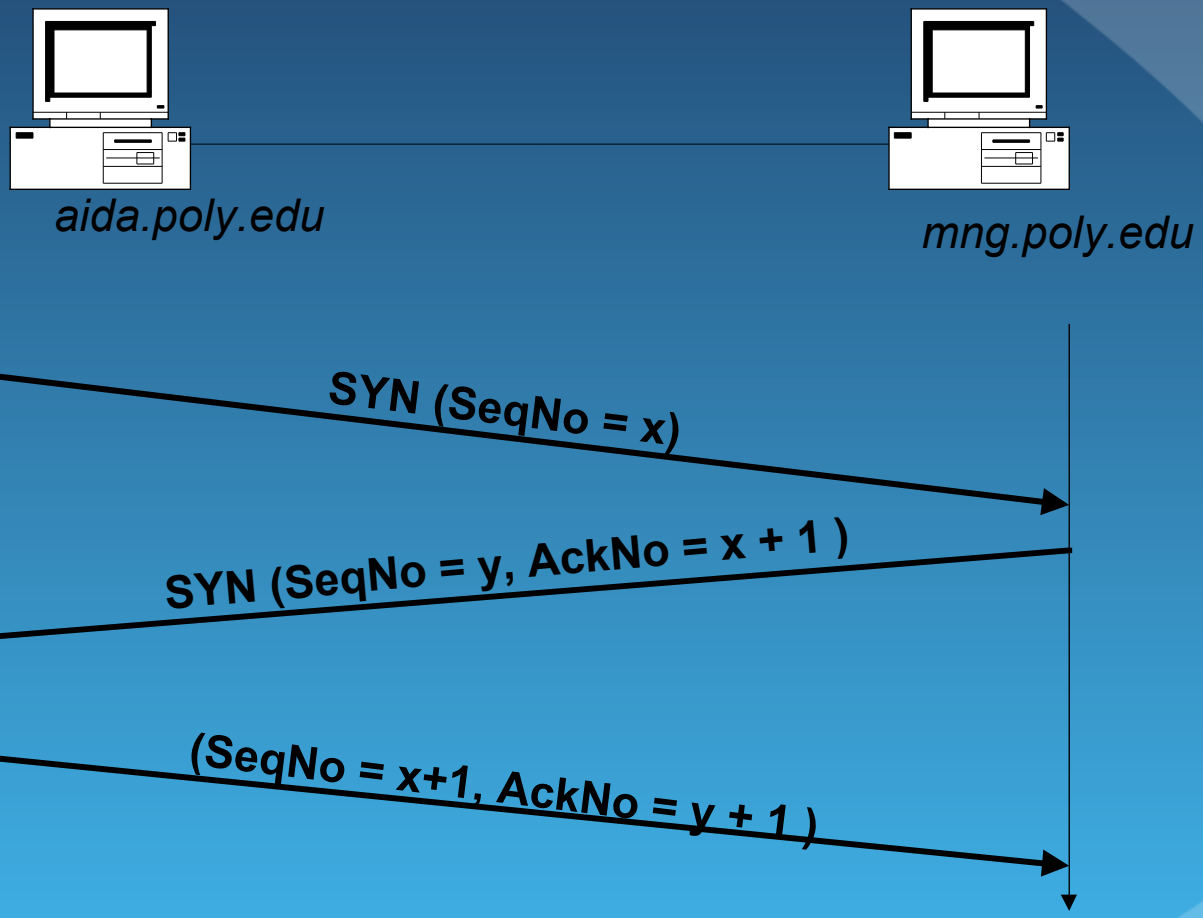
- TCP uses a **three-way handshake** to open a connection:

  (1) ACTIVE OPEN: Client sends a segment with
  - SYN bit set
  - port number of client
  - initial sequence number (ISN) of client

  (2) PASSIVE OPEN: Server responds with a segment with
  - SYN bit set
  - initial sequence number of server
  - ACK for ISN of client

  (3) Client acknowledges by sending a segment with:
  - ACK ISN of server

Three-Way Handshake

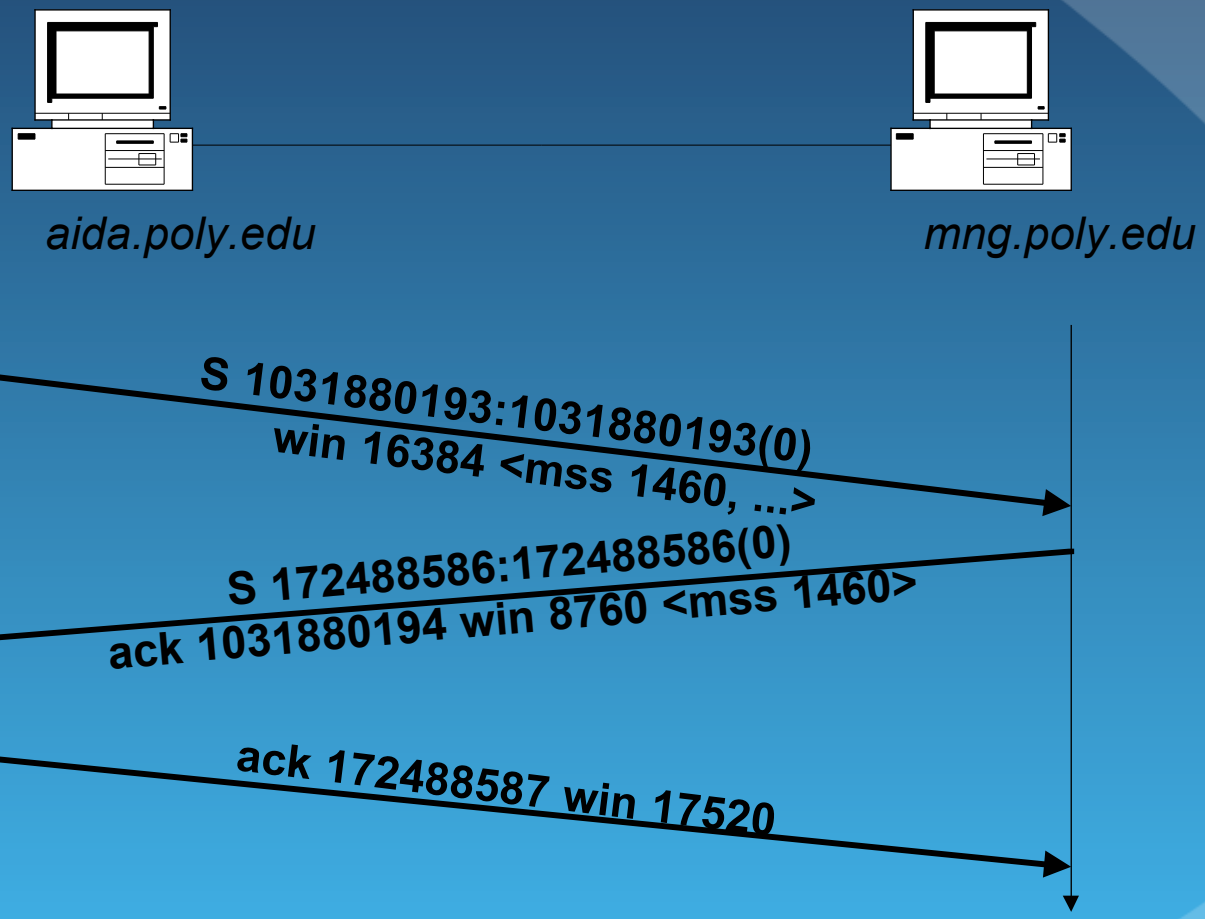


A Closer Look with tcpdump

aida issues
an "telnet mng"


```
1   aida.poly.edu.1121 > mng.poly.edu.telnet: S 1031880193:1031880193(0)
                                    win 16384 <mss 1460,nop,wscale 0,nop,nop,timestamp>

2   mng.poly.edu.telnet > aida.poly.edu.1121: S 172488586:172488586(0)
                                    ack 1031880194 win 8760 <mss 1460>

3   aida.poly.edu.1121 > mng.poly.edu.telnet: . ack 172488587 win 17520

4   aida.poly.edu.1121 > mng.poly.edu.telnet: P 1031880194:1031880218(24)
                                    ack 172488587 win 17520

5   mng.poly.edu.telnet > aida.poly.edu.1121: P 172488587:172488590(3)
                                    ack 1031880218 win 8736

6   aida.poly.edu.1121 > mng.poly.edu.telnet: P 1031880218:1031880221(3)
                                    ack 172488590 win 17520
```

Three-Way Handshake



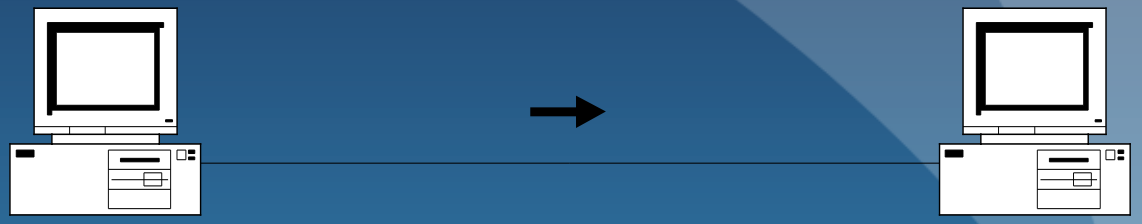
TCP Connection Termination

- Each end of the data flow must be shut down independently **(“half-close”)**
- If one end is done it sends a FIN segment. This means that no more data will be sent

- Four steps involved:
  - (1) X sends a FIN to Y **(active close)**
  - (2) Y ACKs the FIN,
    - (at this time: Y can still send data to X)
  - (3) and Y sends a FIN to X **(passive close)**
  - (4) X ACKs the FIN.

Connection termination with tcpdump

aida issues
an "telnet mng"



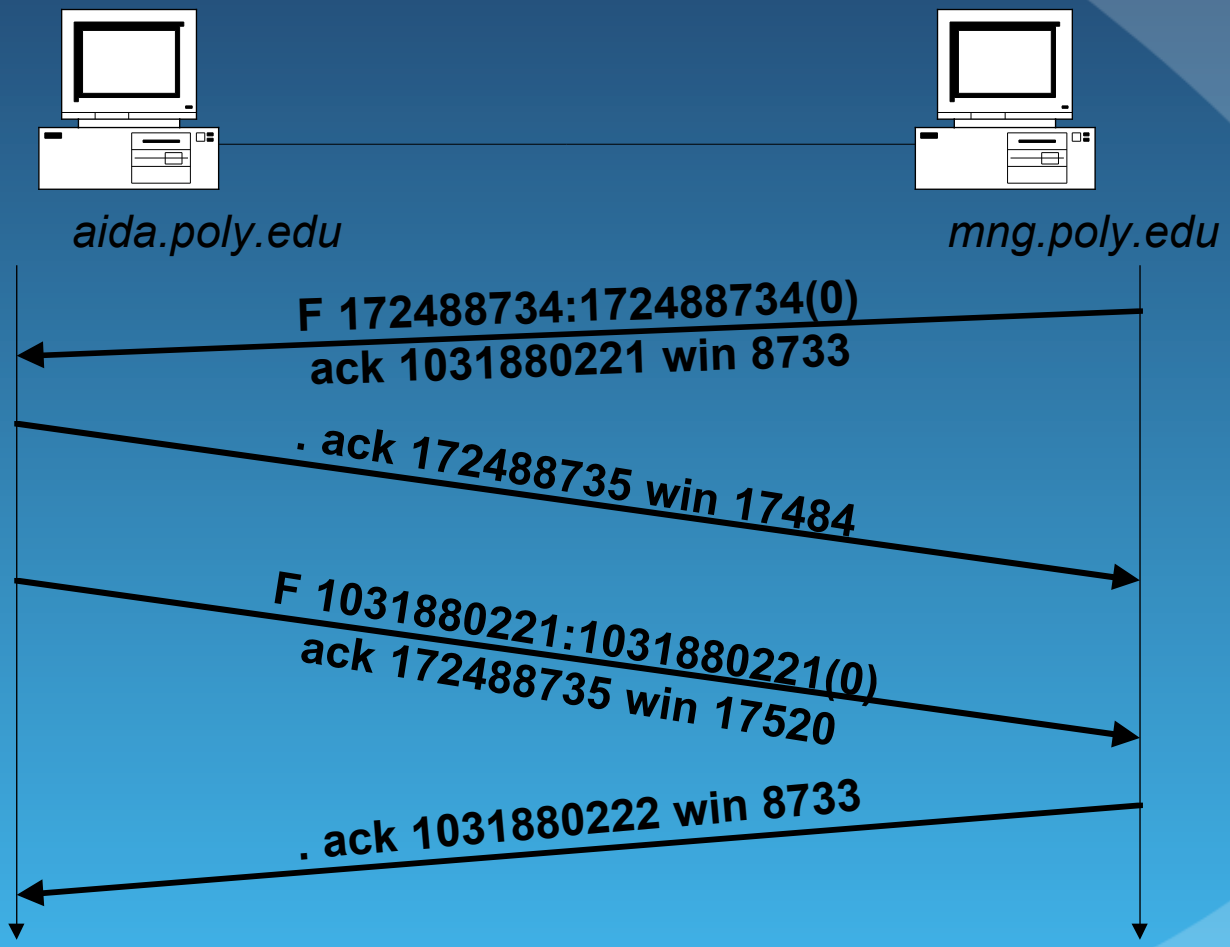
1 mng.poly.edu.telnet > aida.poly.edu.1121: F 172488734:172488734(0)
ack 1031880221 win 8733

2 aida.poly.edu.1121 > mng.poly.edu.telnet: . ack 172488735 win 17484

3 aida.poly.edu.1121 > mng.poly.edu.telnet: F 1031880221:1031880221(0)
ack 172488735 win 17520

4 mng.poly.edu.telnet > aida.poly.edu.1121: . ack 1031880222 win 8733

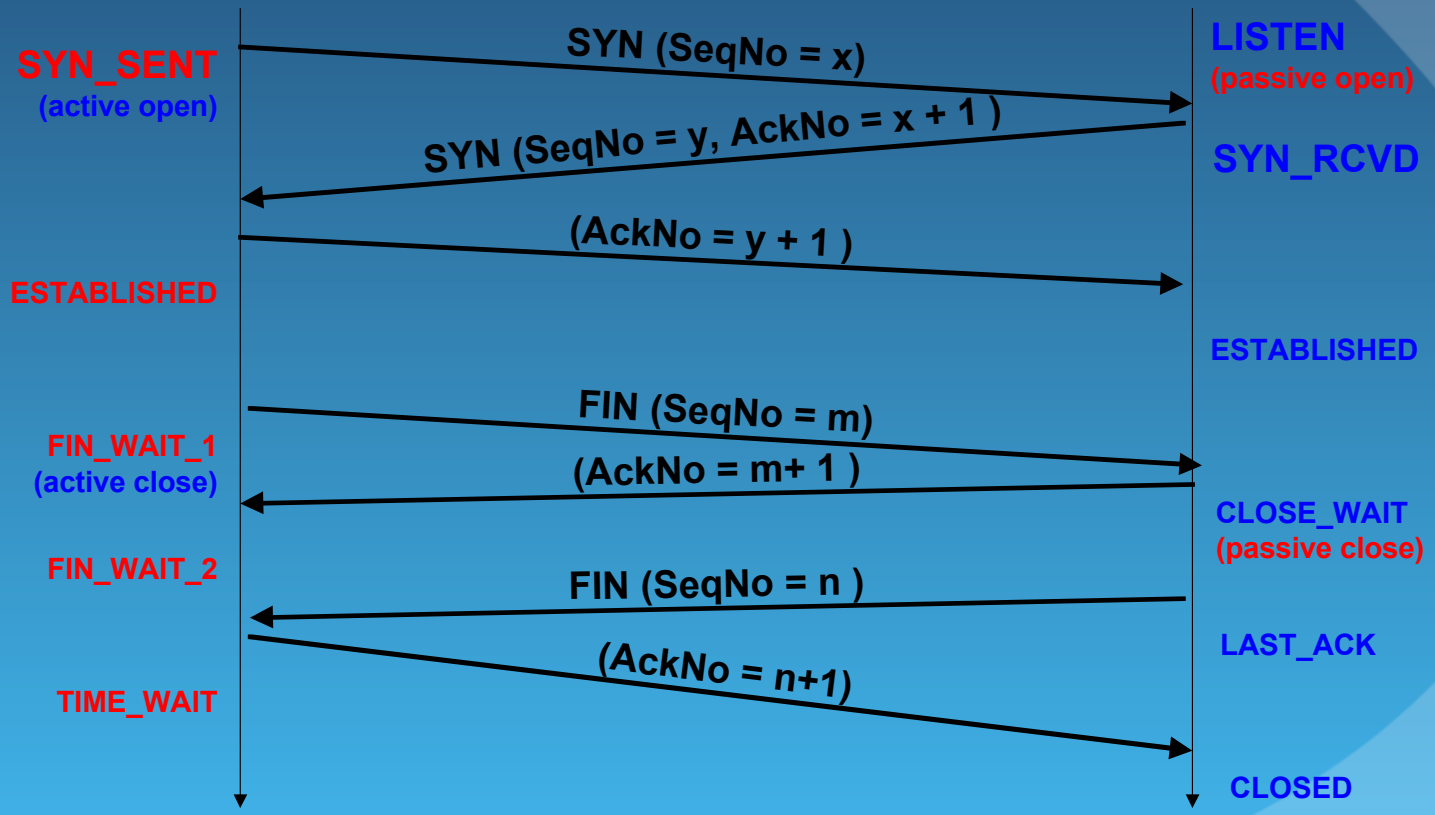
TCP Connection Termination



TCP States

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for Ack
SYN SENT	The client has started to open a connection
ESTABLISHED	Normal data transfer state
FIN WAIT 1	Client has said it is finished
FIN WAIT 2	Server has agreed to release
TIMED WAIT	Wait for pending packets (“2MSL wait state”)
CLOSING	Both Sides have tried to close simultaneously
CLOSE WAIT	Server has initiated a release
LAST ACK	Wait for pending packets

TCP States in "Normal" Connection Lifetime



2MSL Wait State

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must stay in in the TIME_WAIT state for twice the maximum segment lifetime**.

2MSL= 2 * Maximum Segment Lifetime

- Why?
  TCP is given a chance to resend the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.

TCP – Data Flow

Interactive and bulk data

- TCP applications can be put into the following categories

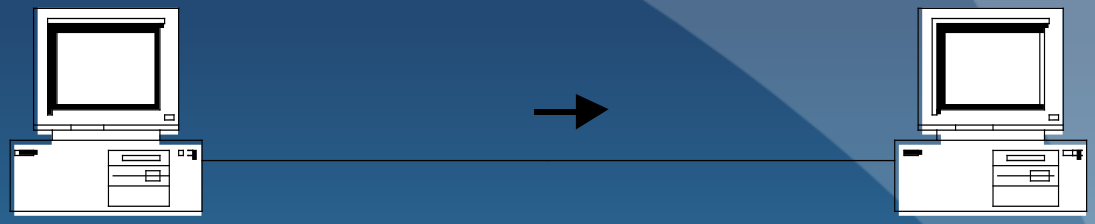
  bulk data transfer - ftp, mail, http

  interactive data transfer - telnet, rlogin

- TCP has algorithms to deal which each type of application efficiently.

tcpdump of an LAN rlogin session

rlogin session
from Argon
to Neon



This is the output of typing 3 (three) characters :

```
44.062449 argon.cs.virginia.edu.1023 > neon.cs.virginia.edu.login: P 0:1(1) ack 1
44.063317 neon.cs.virginia.edu.login > argon.cs.virginia.edu.1023: P 1:2(1) ack 1 win 8760
44.182705 argon.cs.virginia.edu.1023 > neon.cs.virginia.edu.login: . ack 2 win 17520

48.946471 argon.cs.virginia.edu.1023 > neon.cs.virginia.edu.login: P 1:2(1) ack 2 win 17520
48.947326 neon.cs.virginia.edu.login > argon.cs.virginia.edu.1023: P 2:3(1) ack 2 win 8760
48.982786 argon.cs.virginia.edu.1023 > neon.cs.virginia.edu.login: . ack 3 win 17520

55:00.116581 argon.cs.virginia.edu.1023 > neon.cs.virginia.edu.login: P 2:3(1) ack 3 win
17520
55:00.117497 neon.cs.virginia.edu.login > argon.cs.virginia.edu.1023: P 3:4(1) ack 3 win 8760
55:00.183694 argon.cs.virginia.edu.1023 > neon.cs.virginia.edu.login: . ack 4 win 17520
```

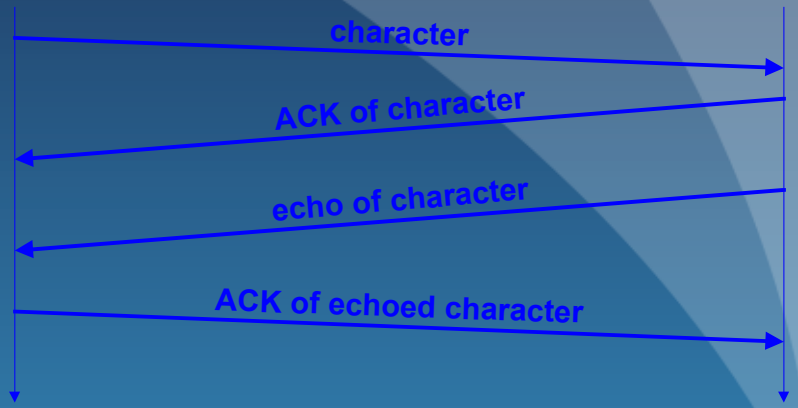
Rlogin

- ``Rlogin'' is a remote terminal application
- Originally built only for Unix systems
- Rlogin sends one segment per character (keystroke)
- Receiver echoes the character back

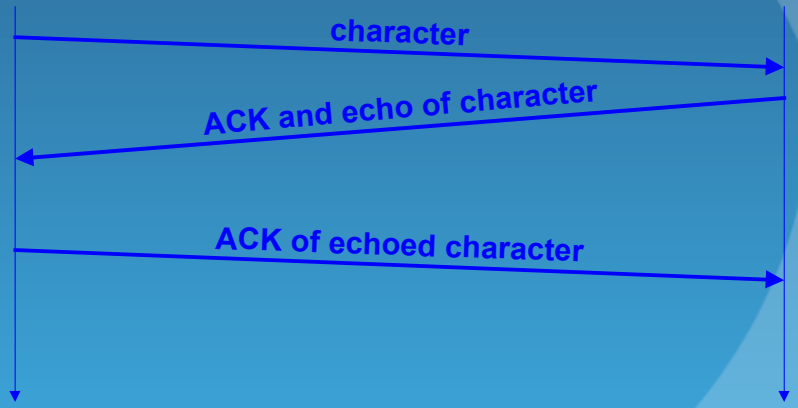
- -> expect to have four segments per keystroke

Rlogin

- We would expect that tcpdump shows this pattern:



- However, tcpdump shows this pattern:



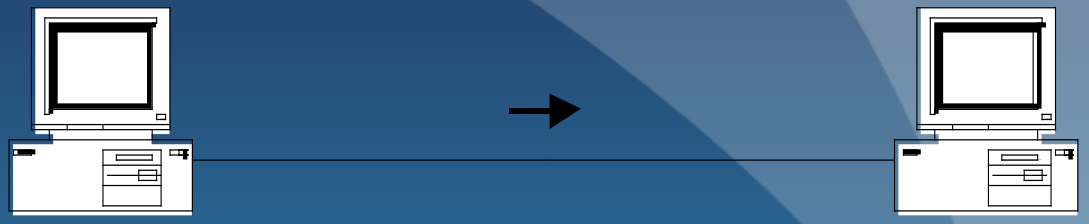
- So, TCP has delayed the transmission of an ACK

Delayed Acknowledgement

- TCP delays transmission of ACKs for up to 200ms(?)
- The hope is to have data ready in that time frame. Then, the ACK can be piggybacked with the data segment.
- Delayed ACKs explain why the ACK and the "echo of character" are sent in the same segment.

tcpdump of a wide-area rlogin session

rlogin session
between argon.cs.virginia.edu
and
tenet.cs.berkeley.edu



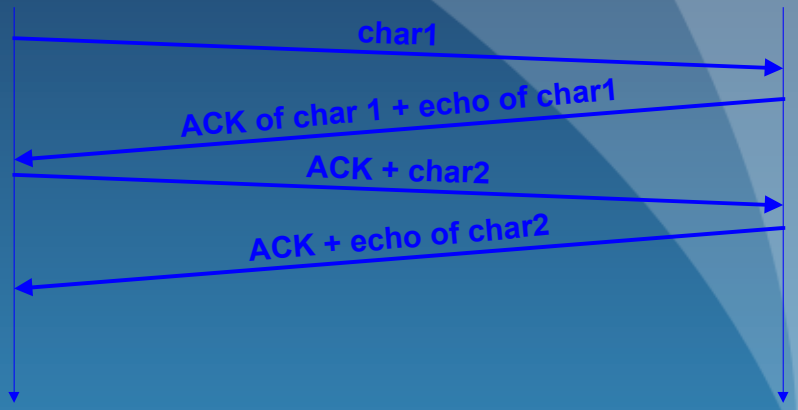
argon.cs.virginia.edu *tenet.cs.berkeley.edu*

This is the output of typing 9 characters :

54:16.401963 argon.cs.virginia.edu.1023 > tenet.CS.Berkeley.EDU.login: P 1:2(1) ack 2 win 16384
54:16.481929 tenet.CS.Berkeley.EDU.login > argon.cs.virginia.edu.1023: P 2:3(1) ack 2 win 16384
54:16.482154 argon.cs.virginia.edu.1023 > tenet.CS.Berkeley.EDU.login: P 2:3(1) ack 3 win 16383
54:16.559447 tenet.CS.Berkeley.EDU.login > argon.cs.virginia.edu.1023: P 3:4(1) ack 3 win 16384
54:16.559684 argon.cs.virginia.edu.1023 > tenet.CS.Berkeley.EDU.login: P 3:4(1) ack 4 win 16383
54:16.640508 tenet.CS.Berkeley.EDU.login > argon.cs.virginia.edu.1023: P 4:5(1) ack 4 win 16384
54:16.640761 argon.cs.virginia.edu.1023 > tenet.CS.Berkeley.EDU.login: P 4:8(4) ack 5 win 16383
54:16.728402 tenet.CS.Berkeley.EDU.login > argon.cs.virginia.edu.1023: P 5:9(4) ack 8 win 16384

Wide-area Rlogin: Observation 1

- Transmission of segments follows a different pattern.
- The delayed acknowled-gment does not kick in
- Reason is that there is always data in the transmit buffer at aida when the ECHO arrives –> long transmission delays in the network



Wide-area Rlogin: Observation 2

- Aida never has multiple segments outstanding.
- This is due to **Nagle's Algorithm:**

  Each TCP connection can have only one small (1-byte) segment outstanding that has not been acknowledged.

- **Implementation:** Send one byte and buffer all subsequent bytes until acknowledgement is received.Then send all buffered bytes in a single segment. (Only enforced if byte is arriving from application one byte at a time) Note – this is not delayed ACK!!!
- Nagle's rule reduces the amount of small segments.
  The algorithm can be disabled.

TCP: Flow, Congestion and Error Control

What is Flow/Congestion/Error Control ?

- **Flow Control:** Algorithms to prevent that the sender overruns the receiver with information?
- **Congestion Control:** Algorithms to prevent that the sender overloads the network
- **Error Control:** Algorithms to recover or conceal the effects from packet losses

→ The goal of each of control mechanism is different.

→ But the implementation is combined

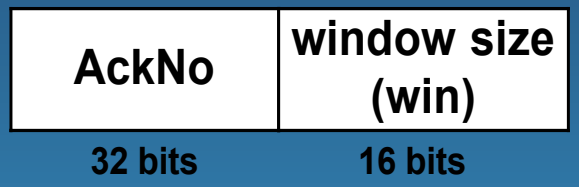
TCP Flow Control

TCP Flow Control

- TCP implements a form of sliding window flow control
  - Sending acknowledgements is separated from setting the window size at sender.
  - Acknowledgements do not automatically increase the window size
  - Acknowledgements are cumulative

Window Management in TCP

- The receiver is returning two parameters to the sender



- The interpretation is:
  - **I am ready to receive new data with SeqNo= AckNo, AckNo+1, …., AckNo+Win-1**

- Receiver can acknowledge data without opening the window
- Receiver can change the window size without acknowledging data

Sliding Window: Example

TCP Congestion Control

TCP Congestion Control

- TCP has a mechanism for congestion control. The mechanism is implemented at the sender
- The window size at the sender is set as follows:

Send Window = MIN (flow control window, congestion window)

where

- flow control window is advertised by the receiver
- congestion window is adjusted based on feedback from the network

TCP Congestion Control

- The sender has two additional parameters:
  - **Congestion Window** (**cwnd**)
    Initial value is 1 MSS (=maximum segment size) counted as bytes
  - **Slow-start threshhold Value** (**ssthresh**)
    Initial value is the advertised window size

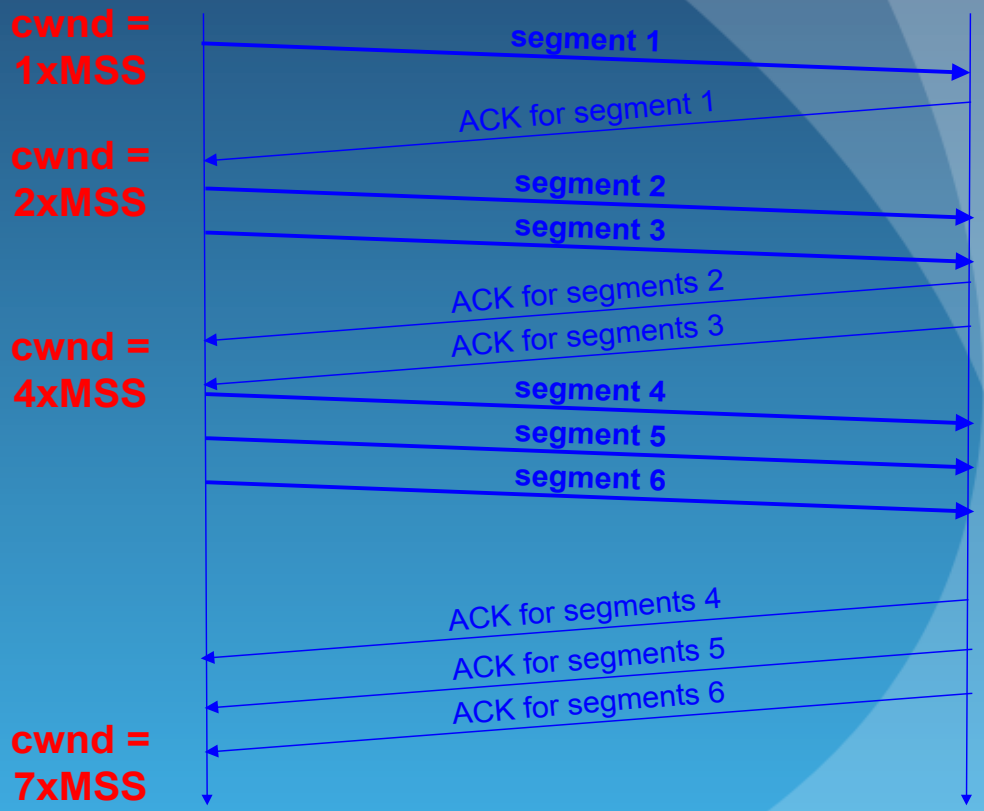
- Congestion control works in two modes:
  - **slow start** (cwnd < ssthresh)
  - **congestion avoidance** (cwnd >= ssthresh)

Slow Start

- Initial value:
  - **cwnd = 1 segment**
- **Note: cwnd is actually measured in bytes:**
  1 segment = MSS bytes
- Each time an ACK is received, the congestion window is increased by MSS bytes.
  - **cwnd = cwnd + 1**
  - If an ACK acknowledges two segments, cwnd is still increased by only 1 segment.
  - Even if ACK acknowledges a segment that is smaller than MSS bytes long, cwnd is increased by 1.
- Does Slow Start increment slowly? Not really.
  In fact, the increase of cwnd can be exponential

Slow Start Example

- The congestion window size grows very rapidly
  - For every ACK, we increase cwnd by 1 irrespective of the number of segments ACK'ed
- TCP slows down the increase of *cwnd* when ***cwnd > ssthresh***



Congestion Avoidance

- Congestion avoidance phase is started if cwnd has reached the slow-start threshold value
  - If cwnd >= ssthresh then each time an ACK is received, increment cwnd as follows:
    - cwnd = cwnd + 1/ [cwnd]

Where [cwnd] is the largest integer smaller than cwnd

- So *cwnd* is increased by one segment (=MSS bytes) only if all segments have been acknowledged.

Slow Start / Congestion Avoidance

If *cwnd* <= *ssthresh* **then**

Each time an Ack is received:
cwnd = cwnd + *1*

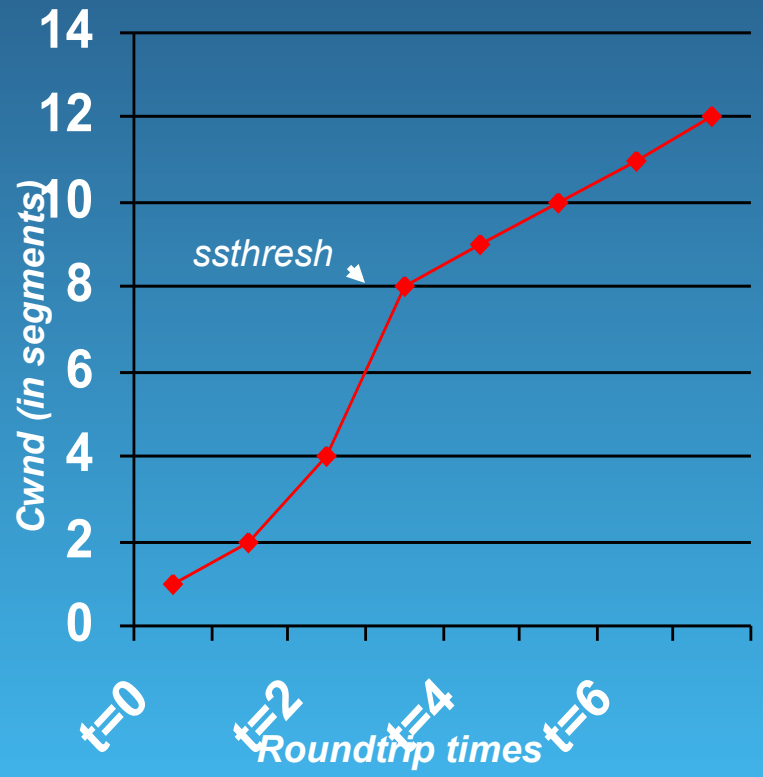
else */* cwnd > ssthresh */*

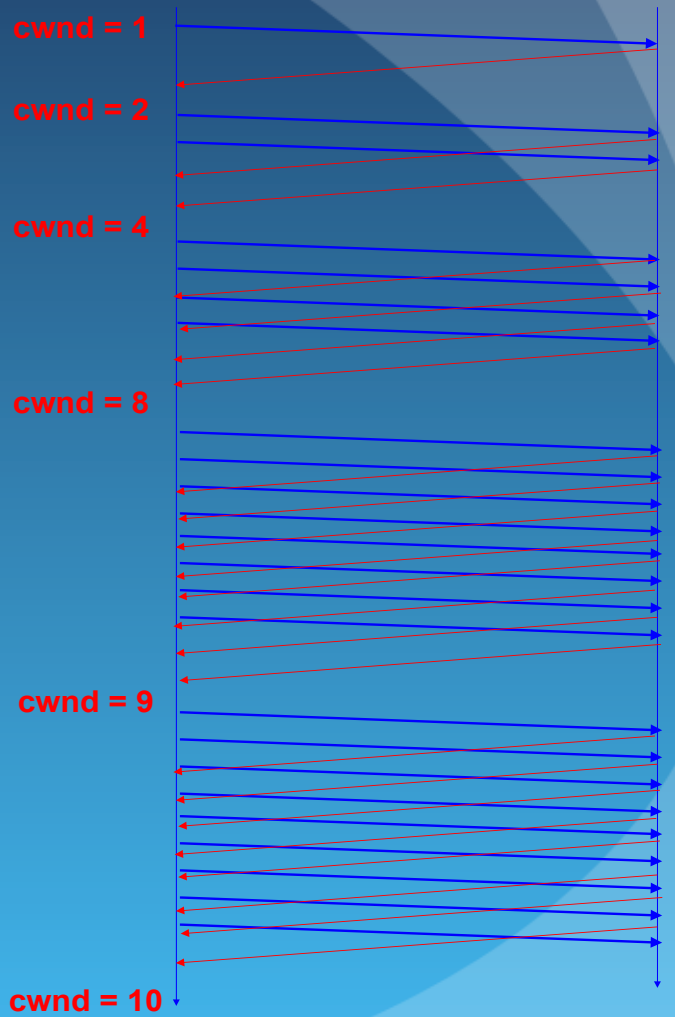
Each time an Ack is received :
cwnd = cwnd + 1 / [cwnd]

endif

Example of Slow Start/Congestion Avoidance

Assume that *ssthresh = 8*





Responses to Congestion

- Most often, a packet loss in a network is due to an overflow at a congested router (rather than due to a transmission error)
- TCP assumes there is congestion in the network if it detects a packet loss
- A TCP sender can detect lost packets via:
  - Timeout of a retransmission timer
  - Receipt of a duplicate ACK
- When TCP detects that there is congestion it reduces the size of the sending window

TCP Tahoe

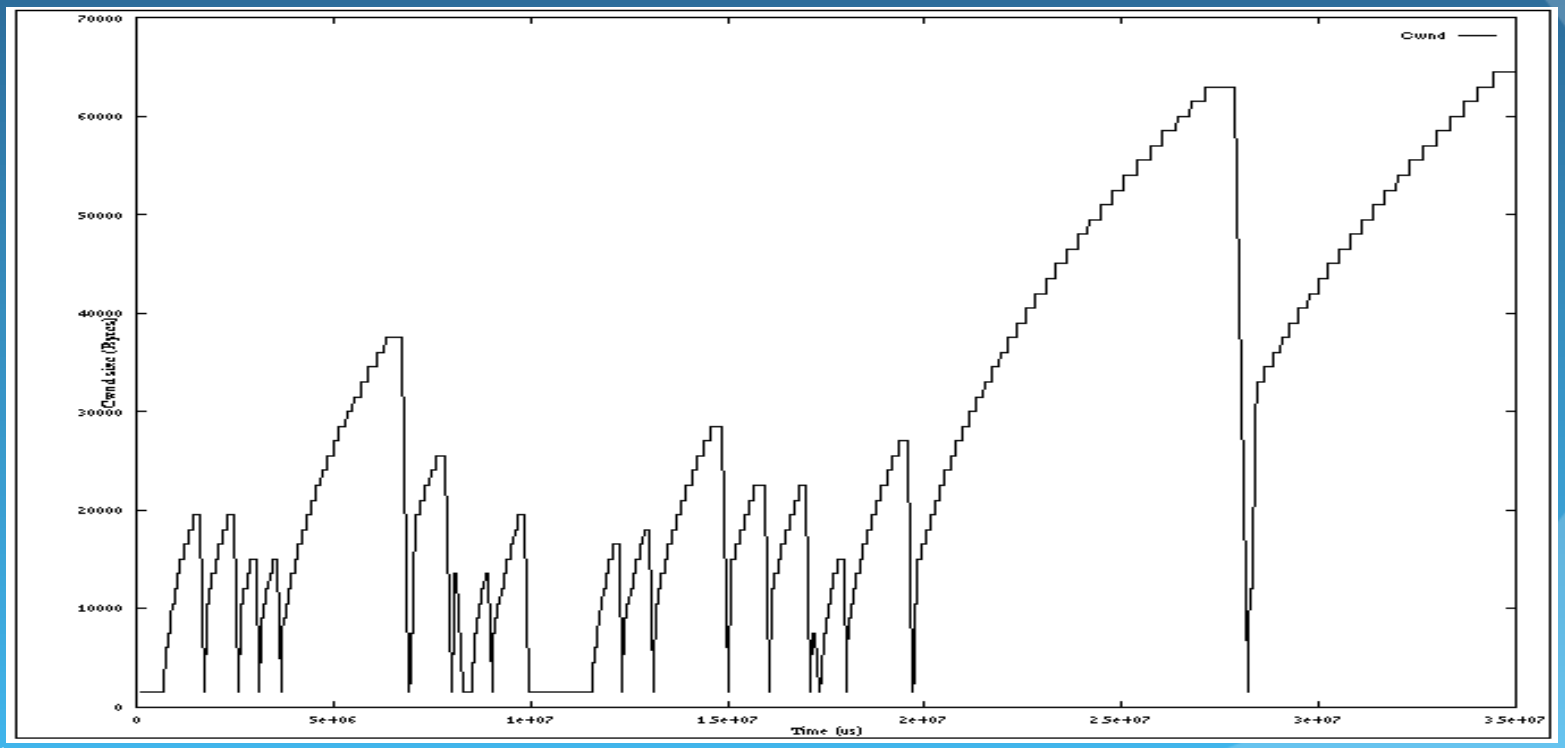
- Congestion is assumed if sender has timed-out or receipt of duplicate ACK
- Each time congestion is detected by TCP,
  - cwnd is reset to one:

    cwnd = 1
  - ssthresh is set to half the current size of the congestion window:

    ssthressh = cwnd / 2
  - and slow-start is entered

Slow Start / Congestion Avoidance

- A typical plot of cwnd for a TCP connection (MSS = 1500 bytes) with TCP Tahoe:



TCP Error Control

Background: ARQ Error Control

- Two types of errors:
  - **Lost packets**
  - **Damaged packets**

- Most Error Control techniques are based on:
  1. Error Detection Scheme (Parity checks, CRC).
  2. Retransmission Scheme.

- Error control schemes that involve error detection and retransmission of lost or corrupted packets are referred to as **Automatic Repeat Request (ARQ) error control**.

Background: ARQ Error Control

- All retransmission schemes use all or a subset of the following procedures:
  - Positive acknowledgments (**ACK**)
  - Negative acknowledgment (**NACK**)
- All retransmission schemes (using ACK, NACK or both) rely on the use of **timers**

- The most common ARQ retransmission schemes are:
  Stop-and-Wait ARQ
  Go-Back-N ARQ
  Selective Repeat ARQ

Error Control in TCP

- TCP implements a variation of the **Go-back-N** retransmission scheme
- TCP maintains a **Retransmission Timer** for each connection:
  - The timer is started for every segment transmission.
  - A timeout causes a retransmission
- TCP couples error control and congestion control (i.e., it assumes that errors are caused by congestion)

TCP Retransmission Timer

- **Retransmission Timer:**
  - The setting of the retransmission timer is crucial for efficiency
  - Timeout value too small → results in unnecessary retransmissions
  - Timeout value too large → long waiting time before a retransmission can be issued
  - A problem is that the delays in the network are not fixed
  - Therefore, the retransmission timers must be adaptive

Round-Trip Time Measurements

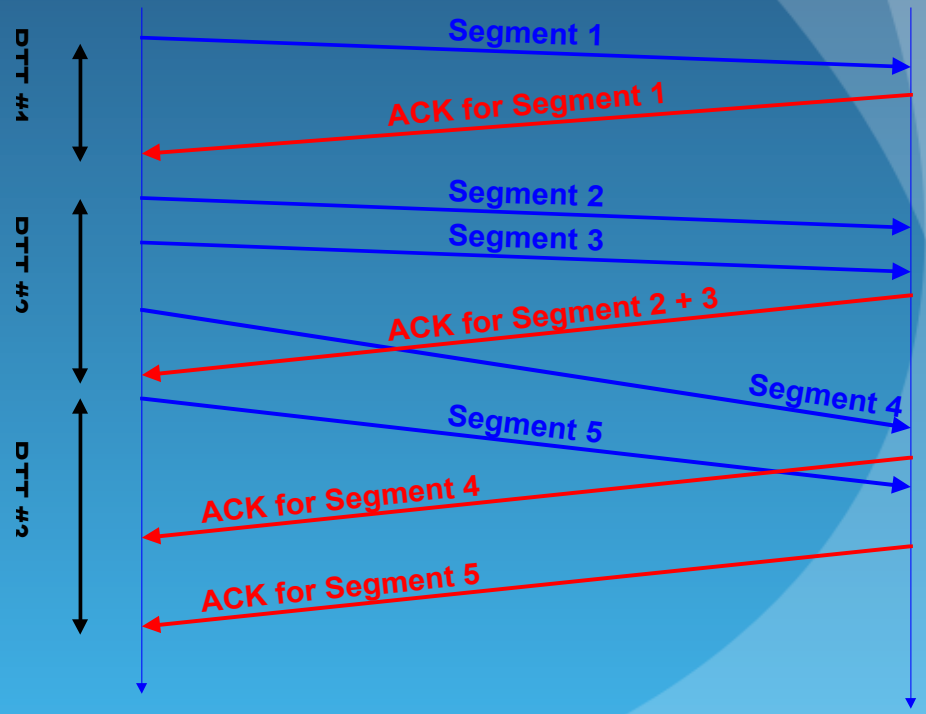
- The retransmission mechanism of TCP is adaptive
- The retransmission timers are set based on round-trip time (RTT) measurements that TCP performs

The RTT is based on time difference between segment transmission and ACK

But:

TCP does not ACK each segment.

Each connection has only one timer



Modifications to TCP

- There have been many improvements suggested and implemented for TCP to get around some of its congestion control problems in new high speed networks and wireless channels.
- Reno, New Reno, etc…… allow for partial acknowledgements, SACKs, retransmission without time out, etc., to overcome the delays associated with recovery from losses and false congestion assumptions.
- Question: Do any of these modifications fit the needs of real-time online gaming??