
COMPUTER NETWORKS

Instructor Notes

Unit – 3 Transport Layer

Introduction to Transport Layer Services: Relationship Between Transport and Network Layer, Overview of the Transport layer in the Internet, Multiplexing and Demultiplexing; Connectionless Transport UDP : UDP Segment Structure, UDP Checksum; Principles of Reliable Data Transfer : Building a Reliable Data Transfer Protocol, Pipelined Reliable Data Transfer Protocol, Go Back N Protocol, Selective Repeat; Connection Oriented Transport TCP: The TCP Connection, TCP Segment Structure, Flow Control, TCP Connection Management, TCP Congestion Control

S.No	Topics	Reference	Page Nos.
1	Introduction to transport layer, Relationship between transport and network layer, Overview of the transport layer in the Internet	T1	Chapter 3 - 3.1 Page 223-229
2	Multiplexing and Demultiplexing	T1	Chapter 3 - 3.2 Page 230-237
3	Connectionless transport: UDP, Segment structure, Checksum	T1	Chapter 3 - 3.3 Page 238-243
4	Principles of reliable data transfer, Building a reliable data transfer protocol	T1	Chapter 3 - 3.4 Page 244- 260
5	Pipelined reliable data transfer protocol	T1	Chapter 3 - 3.4.3 & 3.4.4 Page 260-271
6	Go-Back-N protocol	T1	Chapter 3 - 3.4.3 Page 260- 265
7	Selective repeat	T1	Chapter 3 - 3.4.4 Page 265- 271

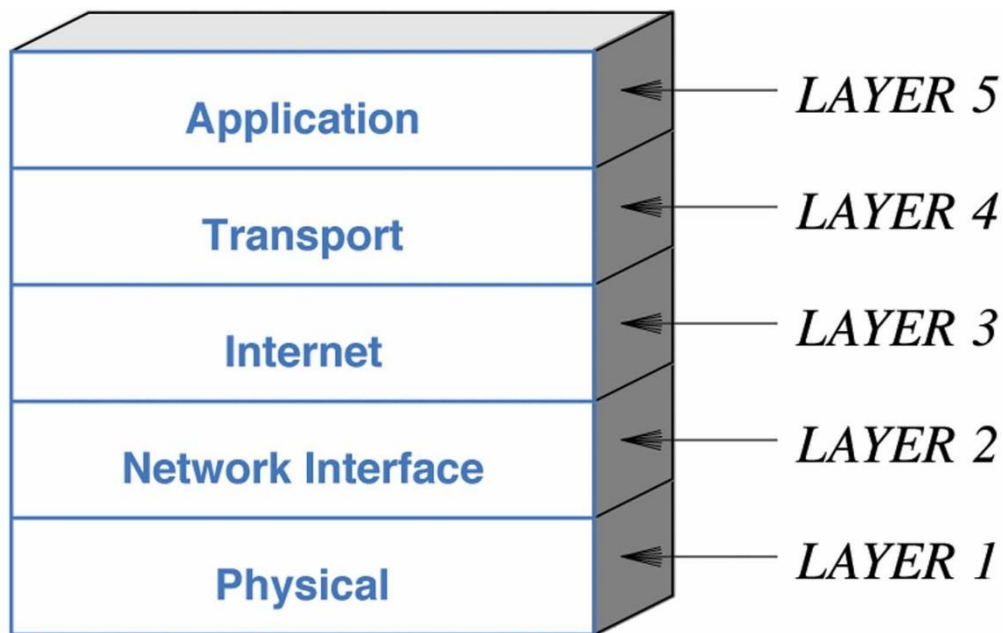
8	Connection Oriented Transport: TCP, The TCP connection, TCP segment structure	T1	Chapter 3 - 3.5 Page 272 - 283
9	Flow control	T1	Chapter 3 - 3.5.4 Page 283- 291
10	TCP connection management	T1	Chapter 3 - 3.5.5 & 3.5.6 Page 291-301
11	TCP congestion control	T1	Chapter 3 - 3.6 Page 302-308
12	TCP congestion control	T1	Chapter 3 - 3.7 Page 311-326

Transport Layer

1. TCP/IP layers
2. TCP/IP layers with some protocols
3. Transport Protocols
4. Multiplexing and Demultiplexing
5. Endpoint Identification
6. Well-known port numbers
7. The User Datagram Protocol
8. The Connectionless Paradigm
9. Message-Oriented Interface
10. Connectionless Multiplexing and Demultiplexing
11. UDP Segment Structure
12. UDP Header Example (DNS Request)
13. UDP Header Example (DNS Response)
14. Internet Checksum
15. Checksum Example
16. Example of Checksum Failure
17. UDP Encapsulation
18. Protocols Using UDP
19. Transmission Control Protocol (TCP)
20. End-To-End Service
21. Connection-Oriented Multiplexing and Demultiplexing
22. Multiplexing and Demultiplexing Example
23. Reliable Data Transfer

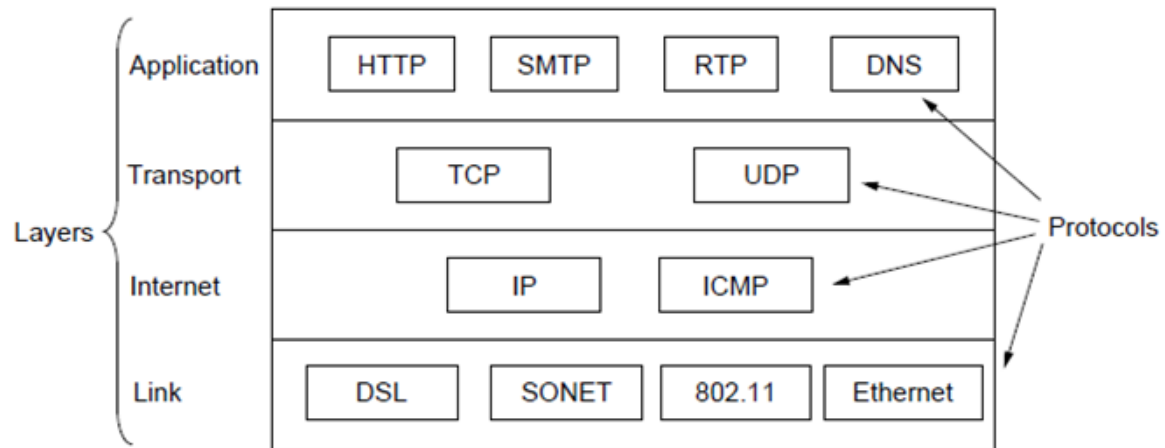
- 24.Simple Reliable Data Transfer
 - 25.Pipelined Reliable Data Transfer
 - 26.Packet Loss and Retransmission
 - 27.Packet Loss and Retransmission - Example
 - 28.Adaptive Retransmission
 - 29.Adaptive Retransmission - Example
 - 30.TCP Segment Structure
 - 31.TCP Flags
 - 32.TCP Example (HTTP Request)
 - 33.TCP Example (HTTP Response)
 - 34.Sequence and Acknowledgement Numbers
 - 35.Example: Lost Acknowledgement
 - 36.Example: Single Retransmission
 - 37.Example: No Retransmission Necessary
 - 38.Flow Control
 - 39.Flow Control Example
 - 40.TCP Connection Establishment
 - 41.Example: Connection Establishment
 - 42.TCP Example SYN
 - 43.TCP Example ACK+SYN
 - 44.SYN Flood Attack
 - 45.TCP Connection Release
 - 46.Congestion Control
-

1. TCP/IP layers



- recall the 5-layer model above
- the *network interface* layer is often called the *link* layer
- we use the generic term *packet* for each block of data transmitted
- recall that each layer adds its own header, so nature of "packet" varies
- so in fact the following terms are usually used for "packets" at each layer
 - *frames* at the link layer
 - *datagrams* at the internet layer
 - *segments* at the transport layer
- we focus on the transport layer in this section

2. TCP/IP layers with some protocols



- we focus on the *UDP* and *TCP* in this section

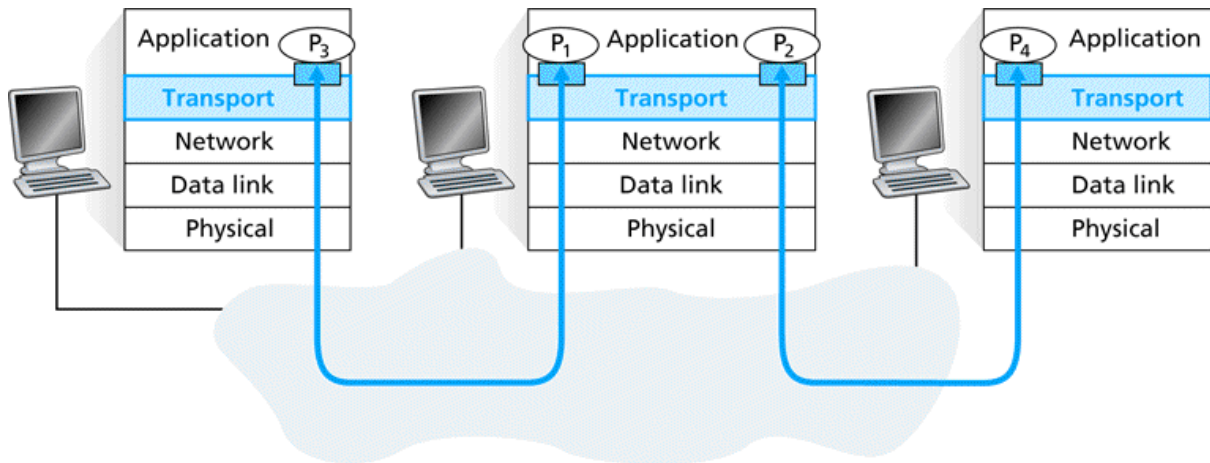
3. Transport Protocols

- Internet Protocol (IP) provides a packet delivery service across an internet
- however, IP cannot distinguish between multiple *processes* (applications) running on the same computer
- fields in the IP datagram header identify only *computers*
- a protocol that allows an application to serve as an *end-point* of communication is known as a *transport protocol* or an *end-to-end protocol*
- the TCP/IP protocol suite provides two transport protocols:
 - the *User Datagram Protocol* (UDP)
 - the *Transmission Control Protocol* (TCP)

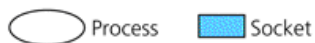
4. Multiplexing and Demultiplexing

- a *socket* is the interface through which a process (application) communicates with the transport layer
- each process can potentially use many sockets
- the transport layer in a receiving machine receives a sequence of segments from its network layer
- delivering segments to the correct socket is called *demultiplexing*

- assembling segments with the necessary information and passing them to the network layer is called *multiplexing*
- multiplexing and demultiplexing are need whenever a communications channel is *shared*

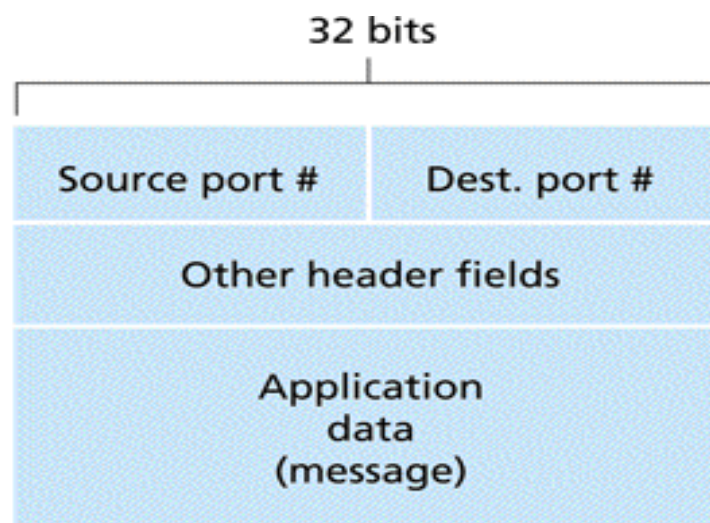


Key:



5. Endpoint Identification

- sockets must have unique identifiers
- each segment must include header fields identifying the socket
- these header fields are the *source port number field* and the *destination port number field*
- each port number is a 16-bit number: 0 to 65535



6. Well-known port numbers

- port numbers below 1024 are called *well-known ports* and are reserved for standard services, e.g.:

Port number	Application protocol	Description	Transport protocol
21	FTP	File transfer	TCP
23	Telnet	Remote login	TCP
25	SMTP	E-mail	TCP
53	DNS	Domain Name System	UDP
79	Finger	Lookup information about a user	TCP
80	HTTP	World wide web	TCP
110	POP-3	Remote e-mail access	TCP
119	NNTP	Usenet news	TCP
161	SNMP	Simple Network Management Protocol	UDP

- these pre-defined port numbers are registered with the [Internet Assigned Numbers Authority](#) (IANA)

7. The User Datagram Protocol

- UDP is less complex and easier to understand than TCP
- the characteristics of UDP are given below:
 - end-to-end*: UDP can identify a specific process running on a computer
 - connectionless*: UDP follows the connectionless paradigm (see below)
 - message-oriented*: processes using UDP send and receive individual messages called *segments* or *user datagrams*
 - best-effort*: UDP offers the same best-effort delivery as IP
 - arbitrary interaction*: UDP allows processes to send to and receive from as many other processes as it chooses
 - operating system independent*: UDP identifies processes independently of the local operating system

8. The Connectionless Paradigm

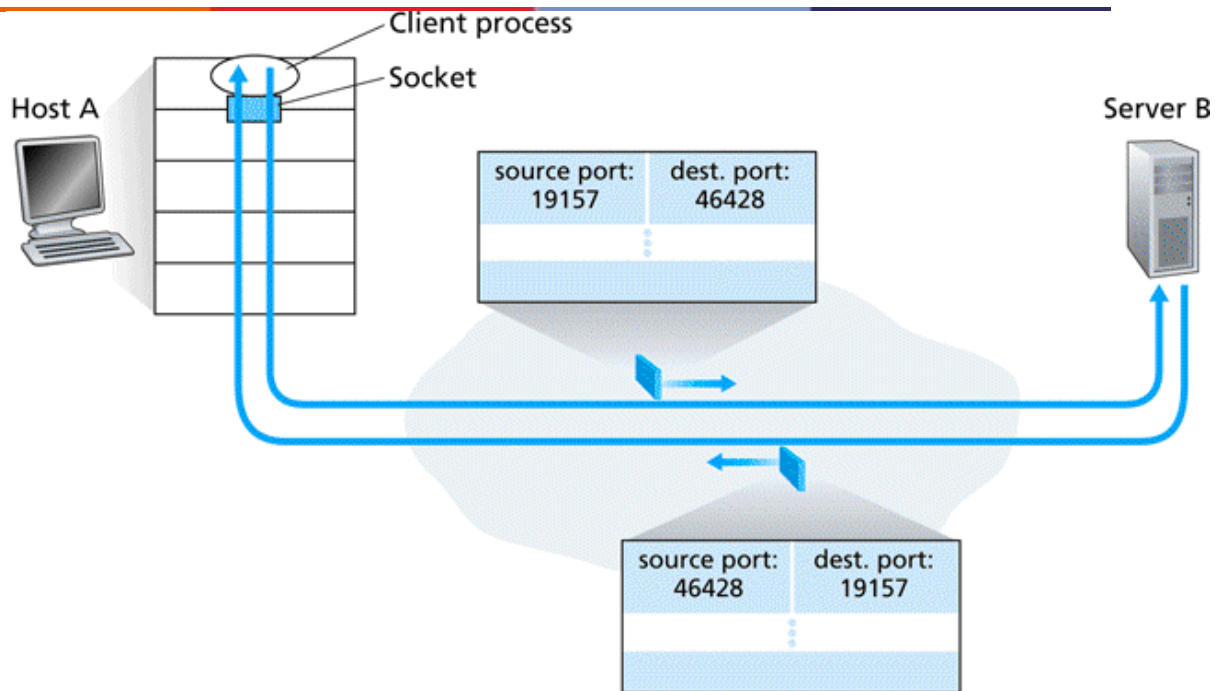
- UDP uses a *connectionless* communication setup
- a process using UDP does not need to establish a connection before sending data (unlike TCP)
- when two processes stop communicating there are no, additional, control messages (unlike TCP)
- communication consists only of the data segments themselves

9. Message-Oriented Interface

- UDP provides a *message-oriented* interface
- each message is sent as a single UDP segment
- however, this also means that the maximum size of a UDP message depends on the maximum size of an IP datagram
- allowing large UDP segments can cause problems
- sending large segments can result in IP fragmentation (see later)
- UDP offers the same best-effort delivery as IP
- this means that segments can be lost, duplicated, or corrupted in transit
- this is why UDP is suitable for applications such as voice or video that can tolerate delivery errors

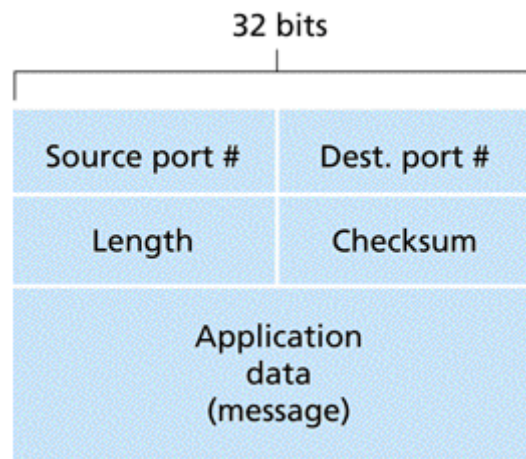
10. Connectionless Multiplexing and Demultiplexing

- say a process on Host A, with port number 19157, wants to send data to a process with UDP port 46428 on Host B
- transport layer in Host A creates a segment containing source port, destination port, and data
- passes it to the network layer in Host A
- transport layer in Host B examines destination port number and delivers segment to socket identified by port 46428
- note: a UDP socket is fully identified by a two-tuple consisting of
 - a destination IP address
 - a destination port number
- source port number from Host A is used at Host B as "return address":



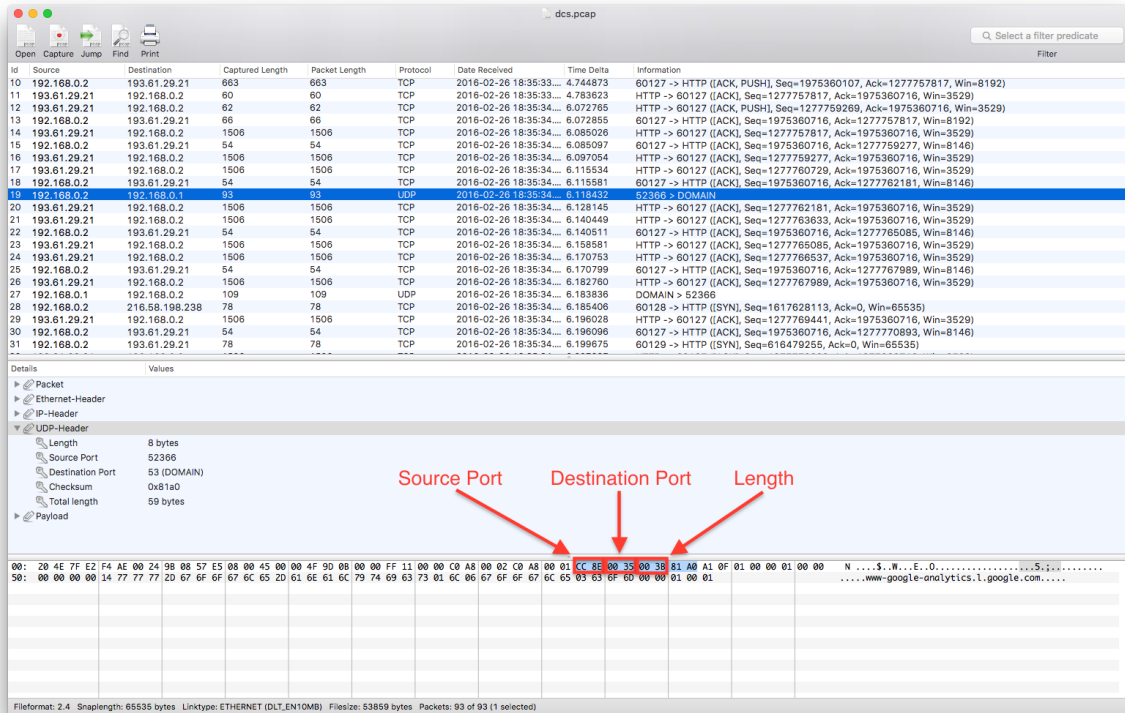
11. UDP Segment Structure

- UDP segment is sometimes called a *user datagram*
- it consists of an 8-byte header followed by the application data (sometimes called *payload*), as shown below



- *Source port #* identifies the *UDP process* which sent the segment
- *Dest port #* identifies the *UDP process* which will handle the application data
- *Length* specifies the length of the segment, including the header, in bytes
- *Checksum* is optional (see below)

12. UDP Header Example (DNS Request)



The image shows a Wireshark packet capture of a network traffic. The packet list pane at the top shows a list of packets. Packet 20 is selected, which is a DNS request. The packet details pane shows the structure of the packet, including the Ethernet II, IP, and UDP headers. The UDP header is expanded, showing the source port (52386), destination port (53), and length (59 bytes). The packet bytes pane shows the raw data of the packet, with hex and ASCII views. Red arrows point to the source port, destination port, and length fields in the UDP header.

No.	Time	Source	Destination	Protocol	Length	Info
10	0.000000	192.168.0.2	193.61.29.21	TCP	60	60127 -> HTTP [ACK, PUSH], Seq=1975360107, Ack=1277757817, Win=8192
11	0.000000	193.61.29.21	192.168.0.2	TCP	60	HTTP -> 60127 [ACK], Seq=1277757817, Ack=1975360716, Win=3529
12	0.000000	193.61.29.21	192.168.0.2	TCP	62	HTTP -> 60127 [ACK, PUSH], Seq=1277759269, Ack=1975360716, Win=3529
13	0.000000	192.168.0.2	193.61.29.21	TCP	66	60127 -> HTTP [ACK], Seq=1975360716, Ack=1277757817, Win=8192
14	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277757817, Ack=1975360716, Win=3529
15	0.000000	192.168.0.2	193.61.29.21	TCP	54	60127 -> HTTP [ACK], Seq=1975360716, Ack=1277759277, Win=8146
16	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277759277, Ack=1975360716, Win=3529
17	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277760729, Ack=1975360716, Win=3529
18	0.000000	192.168.0.2	193.61.29.21	TCP	84	60127 -> HTTP [ACK], Seq=1975360716, Ack=1277762181, Win=8146
19	0.000000	193.61.29.21	192.168.0.2	TCP	23	60127 -> DOMAIN
20	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277762181, Ack=1975360716, Win=3529
21	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277763633, Ack=1975360716, Win=3529
22	0.000000	192.168.0.2	193.61.29.21	TCP	54	60127 -> HTTP [ACK], Seq=1975360716, Ack=1277765085, Win=8146
23	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277765085, Ack=1975360716, Win=3529
24	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277766537, Ack=1975360716, Win=3529
25	0.000000	192.168.0.2	193.61.29.21	TCP	54	60127 -> HTTP [ACK], Seq=1975360716, Ack=1277767989, Win=8146
26	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277767989, Ack=1975360716, Win=3529
27	0.000000	192.168.0.1	192.168.0.2	UDP	109	DOMAIN -> 52386
28	0.000000	192.168.0.2	216.58.198.238	TCP	78	60128 -> HTTP [SYN], Seq=1617628113, Ack=0, Win=65535
29	0.000000	193.61.29.21	192.168.0.2	TCP	1506	HTTP -> 60127 [ACK], Seq=1277769441, Ack=1975360716, Win=3529
30	0.000000	192.168.0.2	193.61.29.21	TCP	54	60127 -> HTTP [ACK], Seq=1975360716, Ack=1277770893, Win=8146
31	0.000000	192.168.0.2	193.61.29.21	TCP	78	60129 -> HTTP [SYN], Seq=616479255, Ack=0, Win=65535

Details

Packet 20: Ethernet II, IP, UDP, Payload

UDP Header:

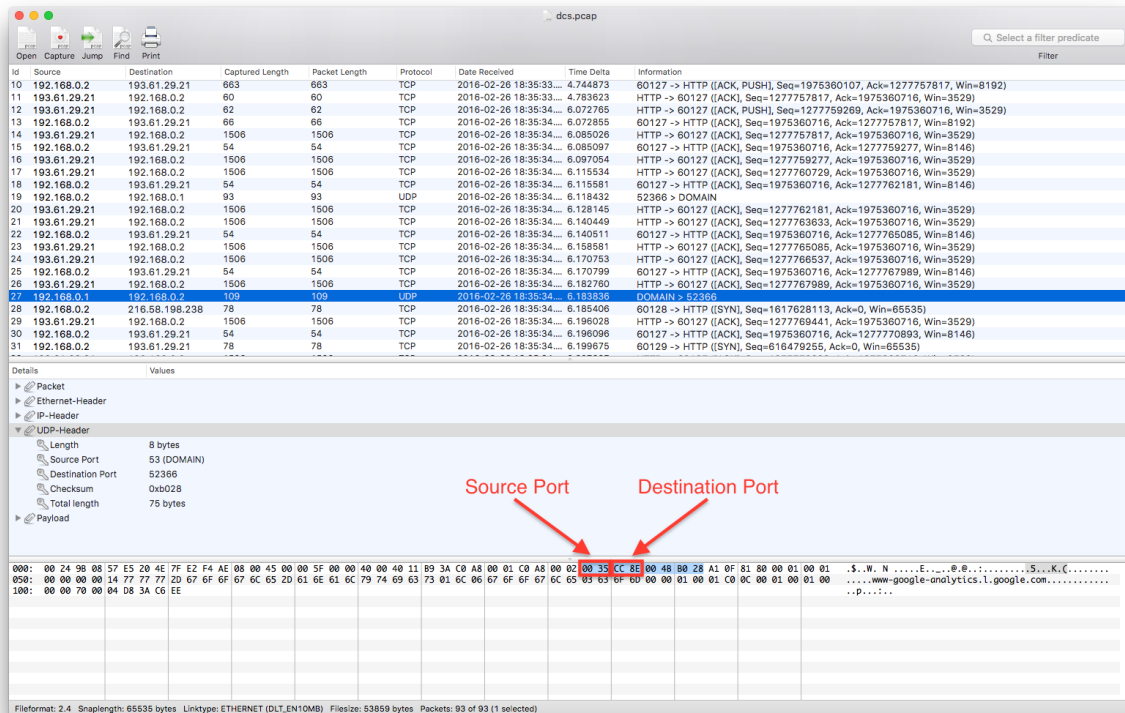
- Length: 8 bytes
- Source Port: 52386
- Destination Port: 53 (DOMAIN)
- Checksum: 0x81a0
- Total length: 59 bytes

Packet bytes:

```

00: 20 4e 7f e2 f4 ae 00 24 98 05 57 e5 08 00 45 00 00 4f 90 08 00 ff 11 00 00 c0 a8 00 02 c0 a5 00 01 cc 8e 00 35 00 3b 81 a8 a1 0f 01 00 00 01 00 00 N...$.W...E..0.....5.....
58: 00 00 00 00 14 77 77 77 2d 67 6f 6f 67 6c 65 2d 61 6e 61 6c 79 74 69 63 73 01 6c 86 67 6f 6f 6c 65 03 63 6f c0 00 00 01 00 01 .....wm-google-analytics.l.google.com....
  
```

13. UDP Header Example (DNS Response)



Source Port: 53 (DOMAIN)
Destination Port: 52366

14. Internet Checksum

- both UDP and TCP use a 16-bit *Checksum* field
- the sender can choose to compute a checksum or set the field to zero
- the receiver only verifies the checksum if the value is non-zero
- note that the checksum is computed using ones-complement arithmetic, so a computed zero value is stored as all-ones

15. Checksum Example

H	e	l	l	o	w	o	r	l	d	.	
48	65	6C	6C	6F	20	77	6F	72	6C	64	2E

$$4865 + 6C6C + 6F20 + 776F + 726C + 642E + \text{carry} = 71FC$$

- to compute the checksum, the sender treats the data as a sequence of binary integers and computes their sum, as illustrated above
- each pair of characters is treated as a 16-bit integer
- if the sum overflows 16 bits, the carry bits are added to the total
- the advantage of such checksums is their size and ease of computation
- addition requires very little computation and the cost of sending an additional 16-bits is negligible

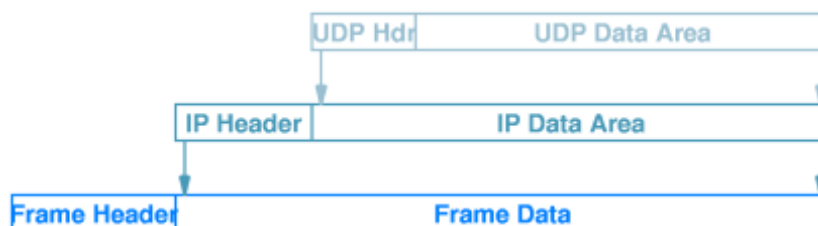
16. Example of Checksum Failure

Data Item In Binary	Checksum Value	Data Item In Binary	Checksum Value
0001	1	0011	3
0010	2	0000	0
0011	3	0001	1
0001	1	0011	3
totals	7		7

- checksums do not detect all common errors, as illustrated above
- a transmission error has inverted the second bit in each of the four data items, yet the checksums are identical

17. UDP Encapsulation

- recall that each layer in the protocol stack adds its own header
- each UDP segment is encapsulated in a network-layer (IP) datagram
- each IP datagram is encapsulated in a link-layer frame



18. Protocols Using UDP

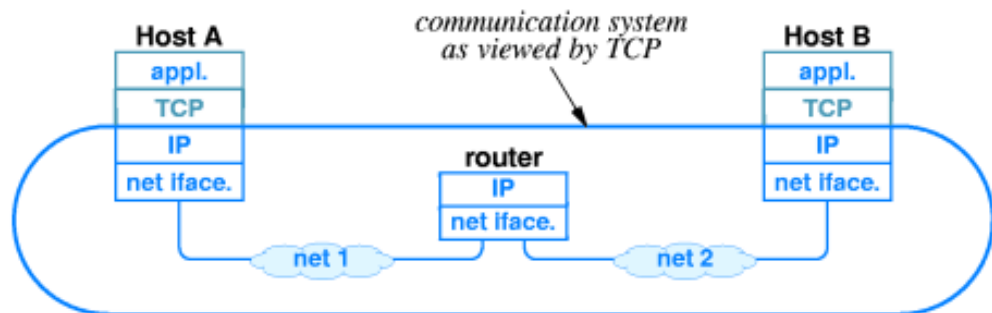
- UDP is especially useful in client-server situations, when a client sends a short request to the server and expects a short response
- if either the request or response is lost, the client times out and tries again
- if all is well, only two packets are required
- an example of an application that uses UDP in this way is the *Domain Name System* (DNS)

19. Transmission Control Protocol (TCP)

- the *Transmission Control Protocol* (TCP) is the transport level protocol that provides *reliability* in the TCP/IP protocol suite
- from an application program's perspective, TCP offers:
 - *connection-oriented*: an application requests a connection, and then uses it for data transfer
 - *point-to-point communication*: each TCP connection has exactly two end points
 - *reliability*: TCP guarantees that the data sent across the connection will be delivered exactly as sent, without missing or duplicate data
 - *full-duplex connection*: a TCP connection allows data to flow in both directions at any time
 - *stream interface*: TCP allows an application to send a continuous stream of bytes across the connection
 - *reliable startup*: TCP requires that two applications must agree to the new connection before it is established
 - *graceful shutdown*: TCP guarantees to deliver all the data reliably before closing the connection

20. End-To-End Service

- TCP uses IP to carry messages, known as *segments*
- each TCP segment is encapsulated in an IP datagram and sent across the Internet
- TCP treats IP as a packet communication system:



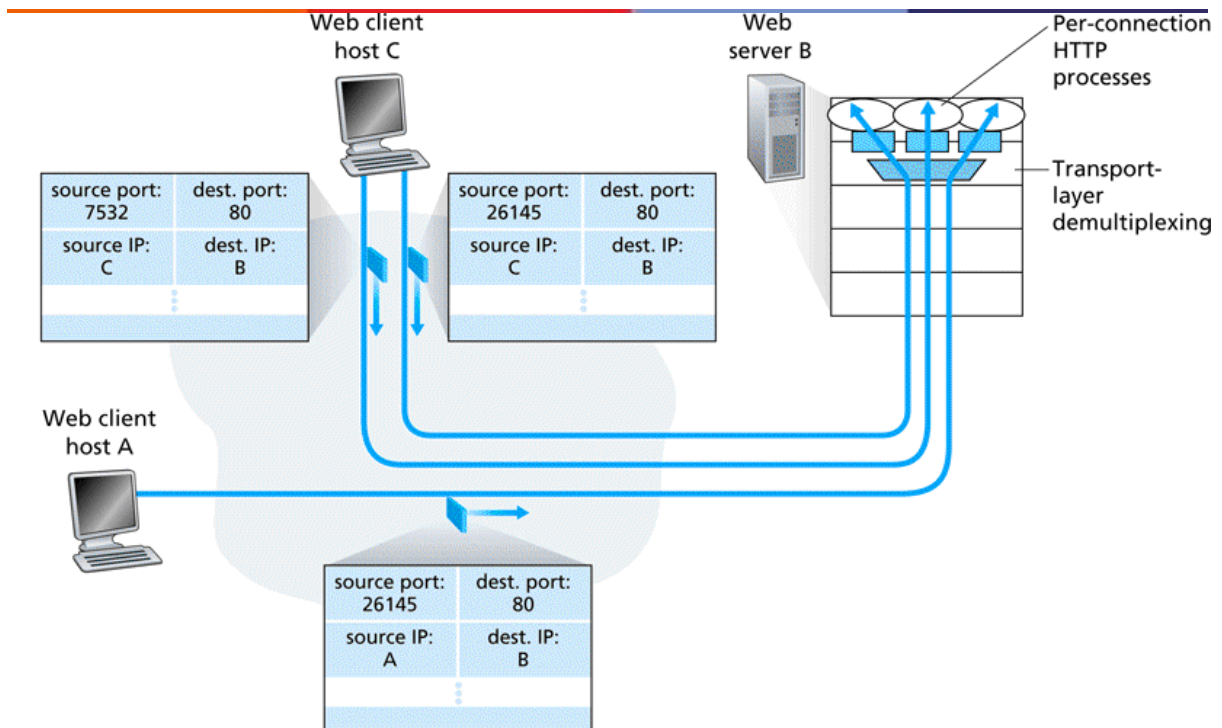
- as illustrated, TCP software is required at both ends of the virtual connection, but not on intermediate routers
- from TCP's point of view, the entire Internet is a communication system capable of accepting and delivering messages without changing their contents

21. Connection-Oriented Multiplexing and Demultiplexing

- each TCP connection has exactly two end-points
- this means that two arriving TCP segments with different source IP addresses or source port numbers will be directed to *two different sockets, even if they have the same destination port number*
- so a TCP socket is identified by a four-tuple:
(source IP address, source port #, destination IP address, destination port #)
- recall UDP uses only (destination IP address, destination port #)

22. Multiplexing and Demultiplexing Example

- an example where clients A and C both communicate with B on port 80:



23. Reliable Data Transfer

- TCP is a *reliable* data transfer protocol
- implemented on top of an *unreliable* network layer (IP)
- some problems:
 - bits in a packet may be *corrupted*
 - packets can be *lost* by the underlying network
- some solutions:
 - *acknowledgements* (ACKs) can be used to indicate packet received correctly
 - a *countdown timer* can be used to detect packet loss
 - packet *retransmission* can be used for lost packets

24. Simple Reliable Data Transfer

- a simple reliable data transfer protocol might
 - send a packet
 - wait until it is sure the receiver has received it correctly
- such a protocol is known as a *stop-and-wait* protocol
- performance of such a protocol on the Internet would be poor

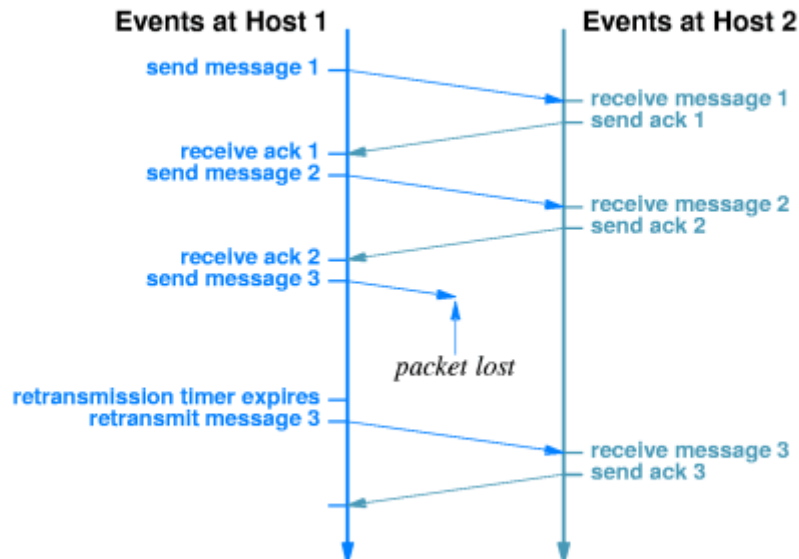
25. Pipelined Reliable Data Transfer

- a *pipelined* protocol allows for multiple data packets to be sent while waiting for acknowledgements
- this results in better network utilisation
- sender and receiver now need buffers to hold multiple packets
- packets need *sequence numbers* in order to identify them
- an acknowledgement needs to refer to corresponding sequence number
- retransmission can give rise to duplicate packets
- sequence numbers in packets allow receiver to detect duplicates

26. Packet Loss and Retransmission

- TCP copes with the loss of packets using *retransmission*
- when TCP data arrives, an *acknowledgement* is sent back to the sender
- when TCP data is sent, a timer is started
- if the timer expires before an acknowledgement arrives, TCP retransmits the data

27. Packet Loss and Retransmission - Example

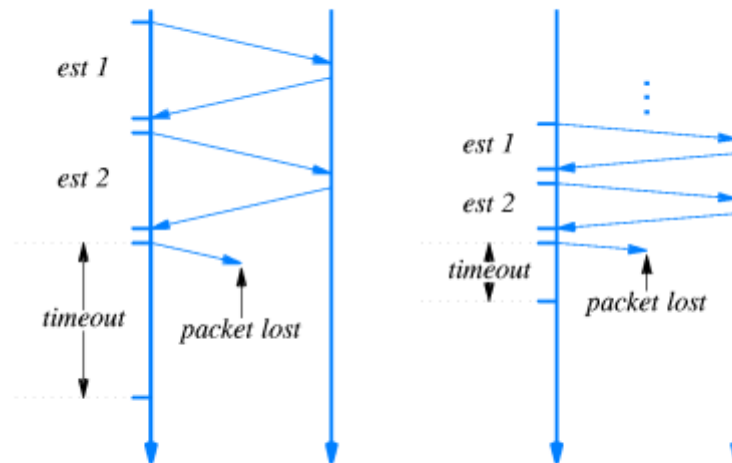


- host on the left is sending data; host on the right is receiving it
- TCP must be ready to retransmit any packet that is lost
- how long should TCP wait?
- the TCP software does not know whether it is using
 - a local area network (acknowledgements within a few milliseconds) or
 - a long-distance satellite connection (acknowledgements within a few seconds)

28. Adaptive Retransmission

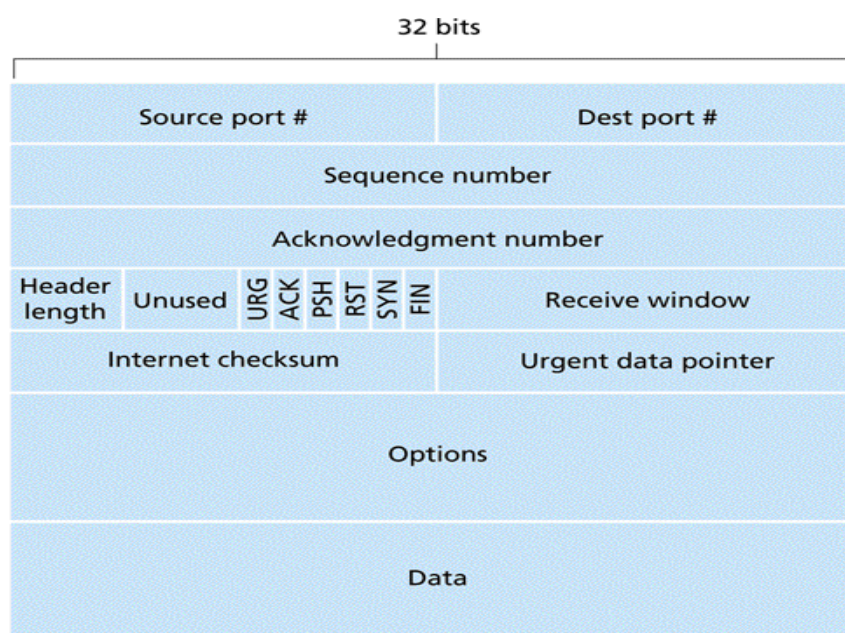
- TCP estimates the *round-trip delay* for each active connection
- for each connection, TCP generates a sequence of round-trip estimates and produces a weighted average (mean)
- it also maintains an estimate of the variance
- it then uses a linear combination of the estimated mean and variance as the value of the timeout

29. Adaptive Retransmission - Example

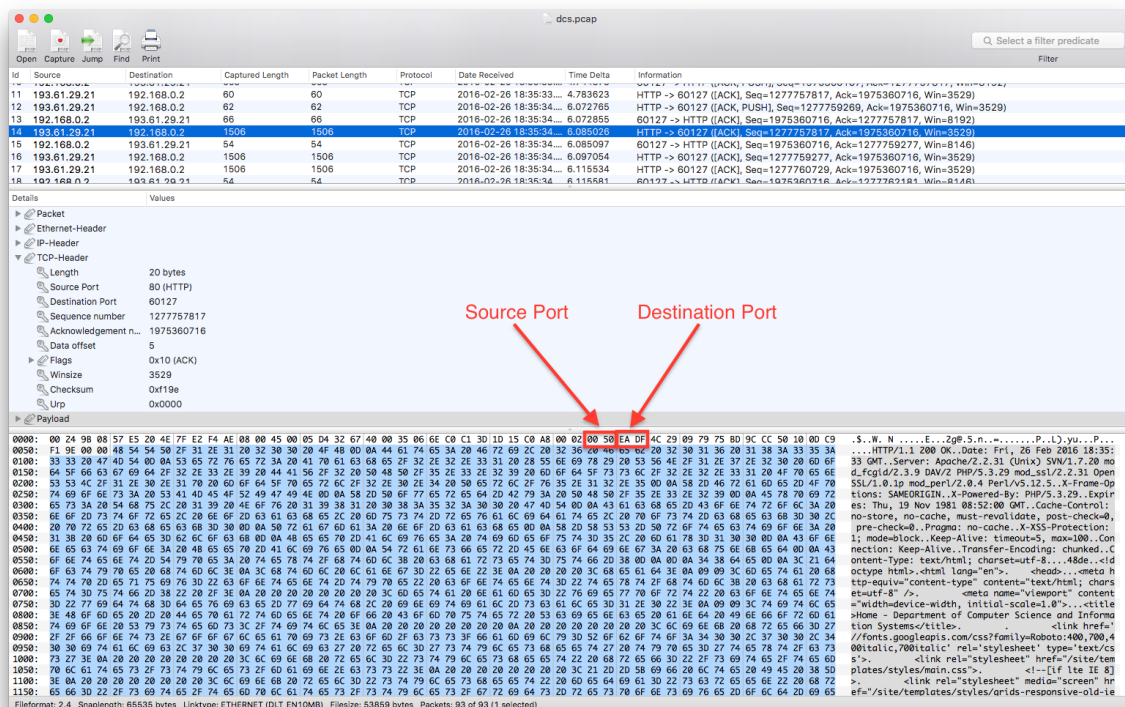


- the connection on the left above has a relatively long round-trip delay
- the connection on the right above has a shorter round-trip delay
- the goal is to wait long enough to decide that a packet was lost, without waiting longer than necessary
- when delays start to vary, TCP adjusts the timeout accordingly

30. TCP Segment Structure



33. TCP Example (HTTP Response)

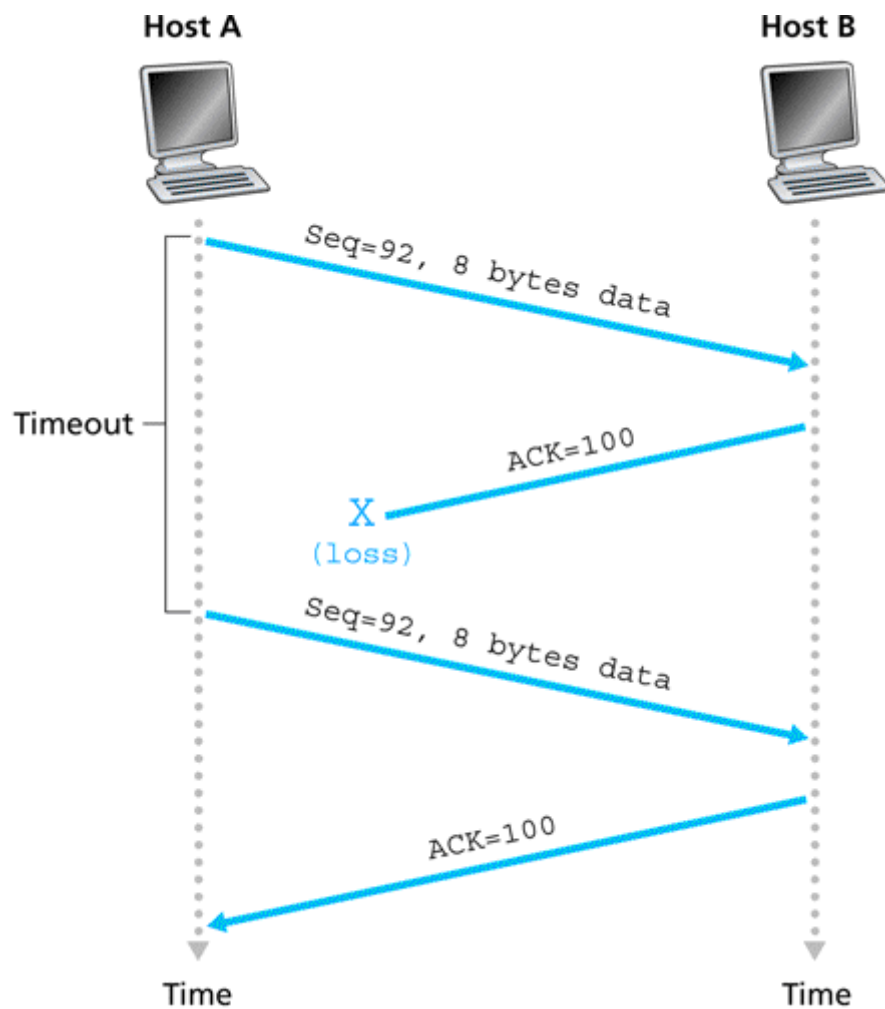


The image shows a Wireshark packet capture of an HTTP response. The packet list at the top shows a TCP segment from 193.61.29.21 to 192.168.0.2, sequence 127775817, and acknowledgment 1975360716. The packet details pane shows the TCP header with source port 80 and destination port 60127. The packet payload shows the start of an HTML document.

34. Sequence and Acknowledgement Numbers

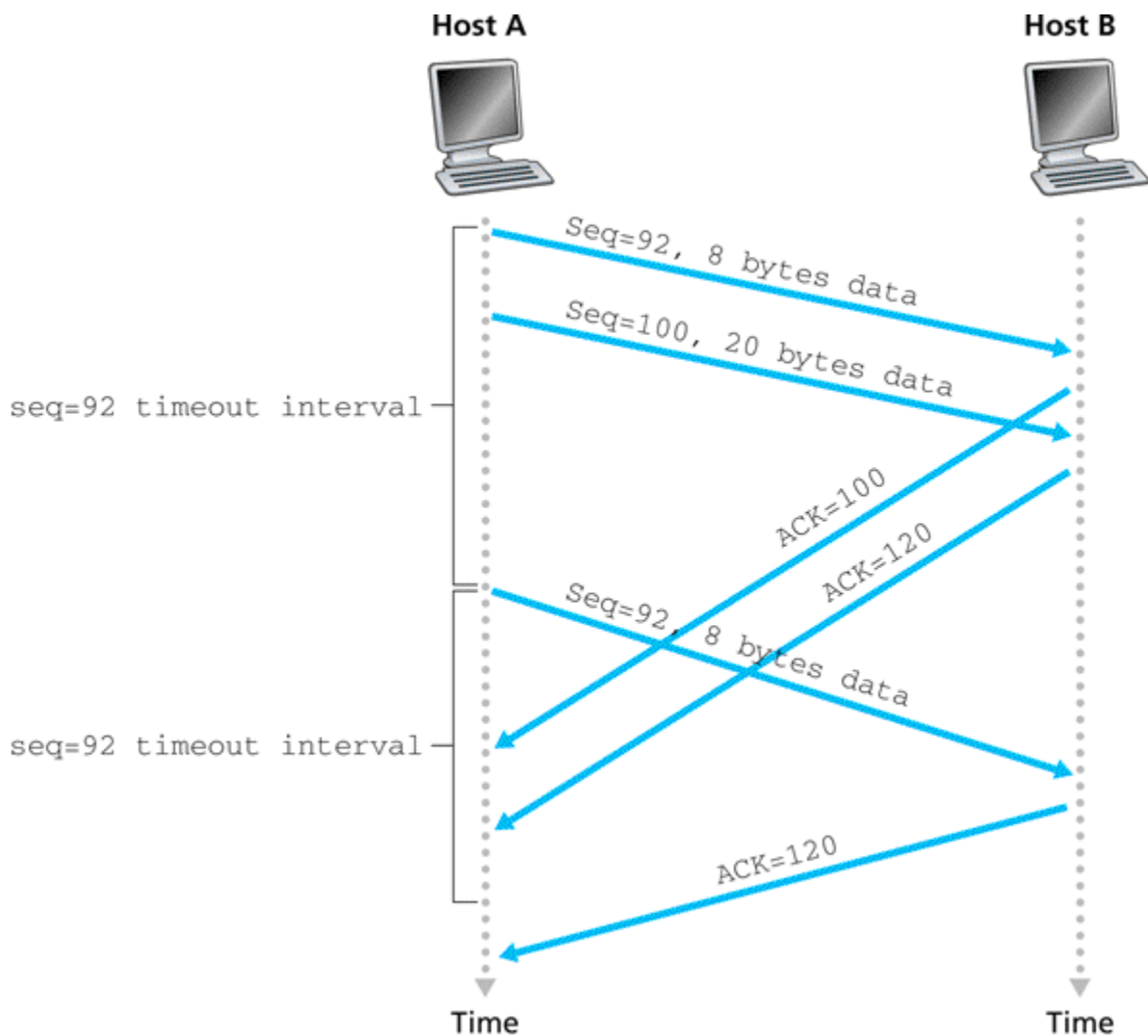
- TCP views data as an ordered stream of bytes
- sequence numbers are with respect to the stream of transmitted bytes
- the *sequence number* for a segment is therefore the byte-stream number of the first data byte in the segment
- the receiver uses the sequence number to re-order segments arriving out of order and to compute an acknowledgement number
- an *acknowledgement number* identifies the sequence number of the incoming data that the receiver expects next
- suppose Host A has received bytes 0 through 535 and 900 through 1000 from Host B, but not bytes 536 through 899
- A's next segment to B will contain 536 in the acknowledgement number field
- TCP only acknowledges bytes up to the first missing byte in the stream
- TCP is said to provide *cumulative acknowledgements*

35. Example: Lost Acknowledgement



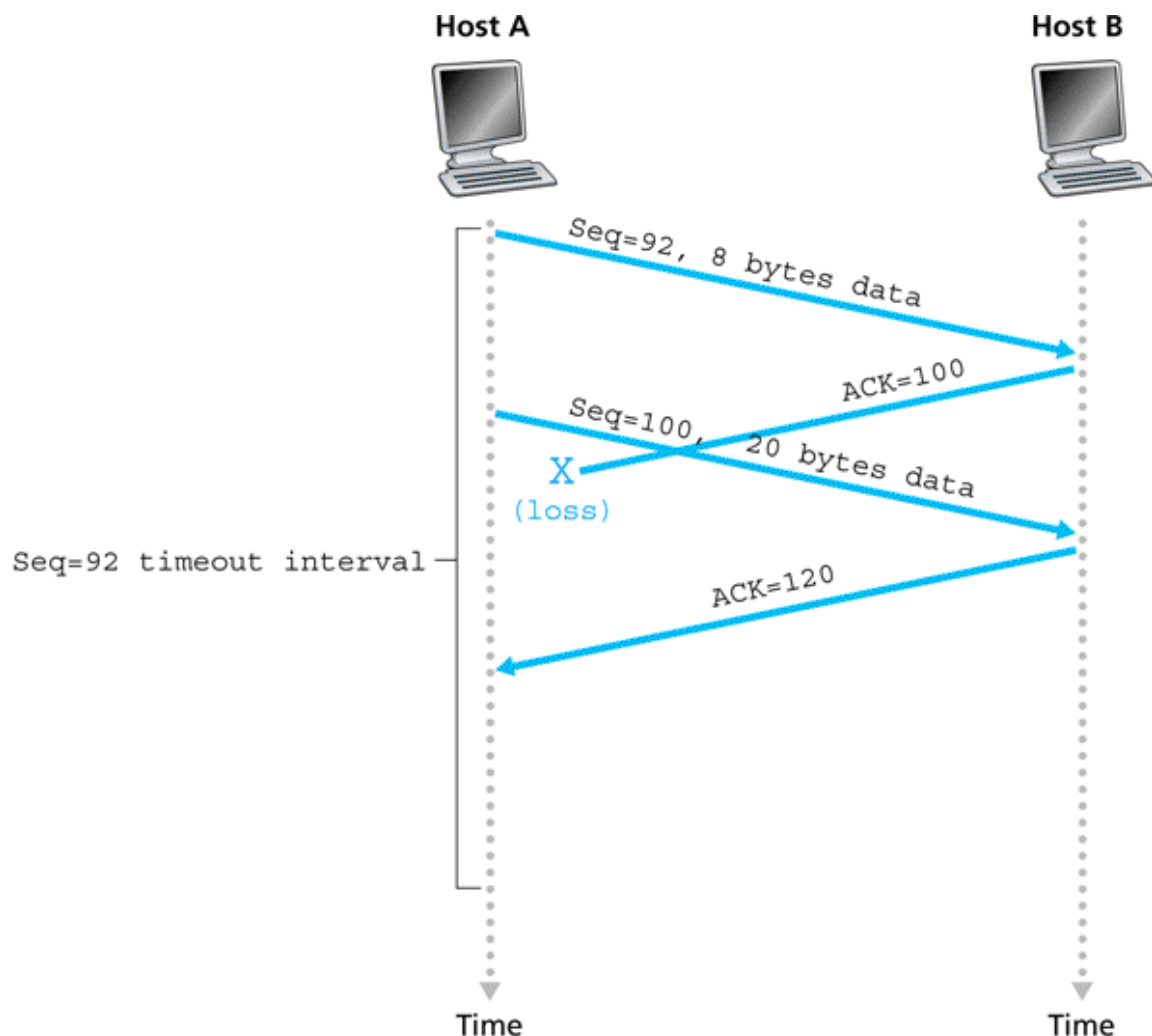
- Host A sends one segment to Host B
- this segment has sequence number 92 and contains 8 bytes of data
- the acknowledgement from B is lost
- A retransmits after its timer expires

36. Example: Single Retransmission



- Host A sends two segments back to back to Host B
- acknowledgements from B arrive only after timeout
- if acknowledgement for second segment arrives before the new timeout, the second segment will not be retransmitted

37. Example: No Retransmission Necessary



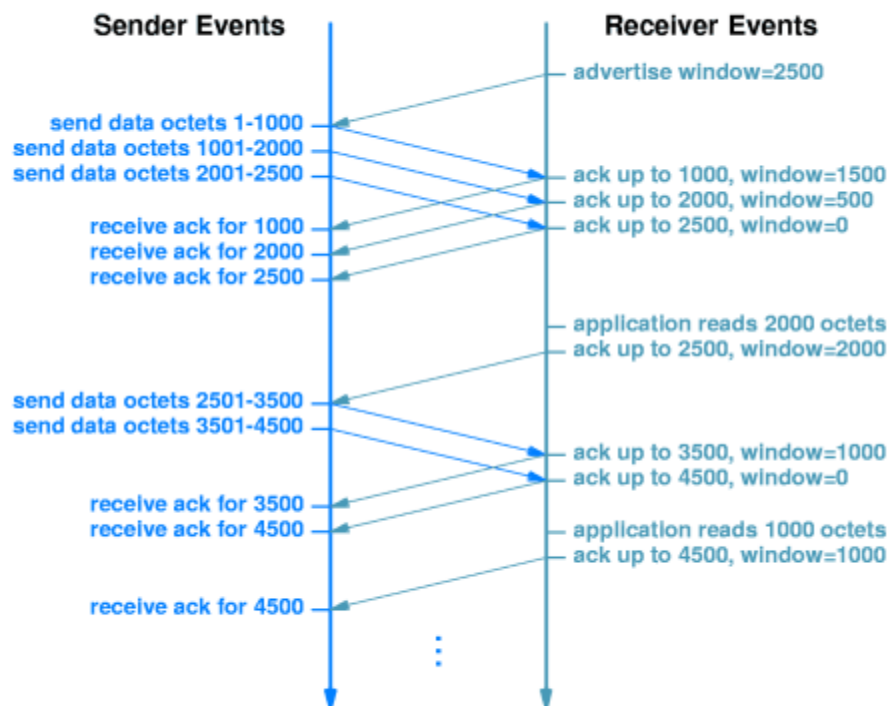
- Host A sends two segments back to back to Host B (as in previous example)
- suppose the acknowledgement for the first segment is lost
- if second acknowledgement arrives before timeout, A does not retransmit either segment

38. Flow Control

- TCP uses a *window* mechanism to control the flow of data
- when a connection is established, each end of the connection allocates a buffer to hold incoming data, and sends the size of the buffer to the other end

- as data arrives, the receiver sends acknowledgements together with the amount of buffer space available called a *window advertisement*
- if the receiving application can read data as quickly as it arrives, the receiver will send a positive window advertisement with each acknowledgement
- however, if the sender is faster than the receiver, incoming data will eventually fill the receiver's buffer, causing the receiver to advertise a *zero window*
- a sender that receives a zero window advertisement must stop sending until it receives a positive window advertisement

39. Flow Control Example



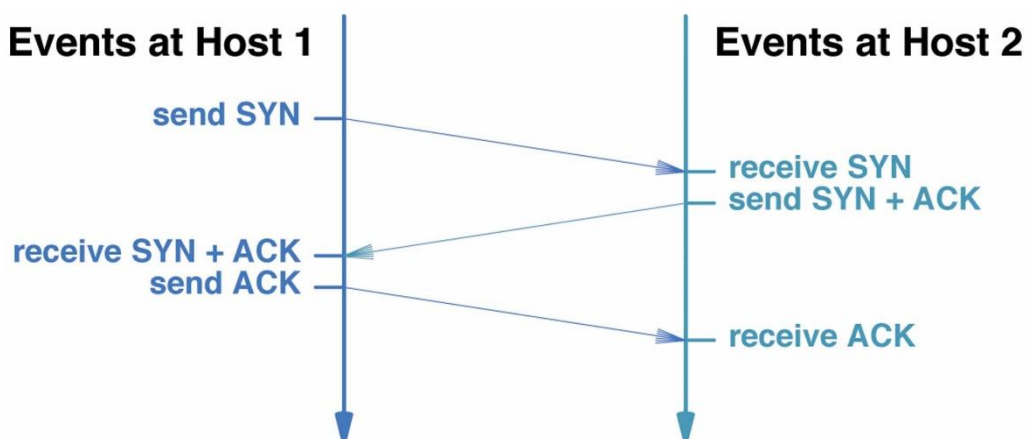
- sender is using a maximum segment size of 1000 bytes
- receiver advertises an initial window size of 2500 bytes
- sender transmits three segments (two containing 1000 bytes and one containing 500 bytes); then waits for an acknowledgement

- the first three segments fill the receiver's buffer faster than the receiving application can consume the data, so the advertised window reaches zero
- after the application reads 2000 bytes, the receiving TCP sends an additional acknowledgement advertising a window of 2000 bytes
- sender responds by sending two 1000-byte segments resulting in another zero window
- application reads 1000 bytes, so the receiving TCP sends an acknowledgement with a positive window size

40. TCP Connection Establishment

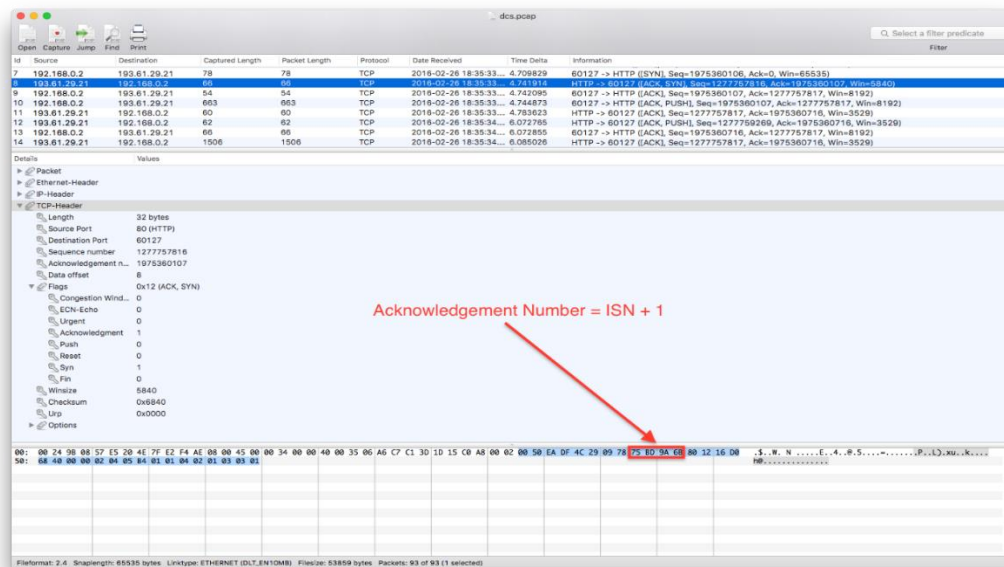
- connections are established by means of a *three-way handshake*
- each side sends a control message, specifying window size and *Initial Sequence Number (ISN)* which is randomly chosen
- a random ISN reduces the chance of a "lost" segment from an already-terminated connection being considered part of this connection
- the three steps are:
 - the sender sends a TCP segment (including window size and ISN) with the SYN flag on
 - the recipient sends a segment (including window size and ISN) with both SYN and ACK flags on
 - the sender replies with ACK

41. Example: Connection Establishment



- host 1 opens the connection with an ISN
- host 2 accepts the connect request by sending a TCP segment which
 - acknowledges host 1's request (ACK flag on)

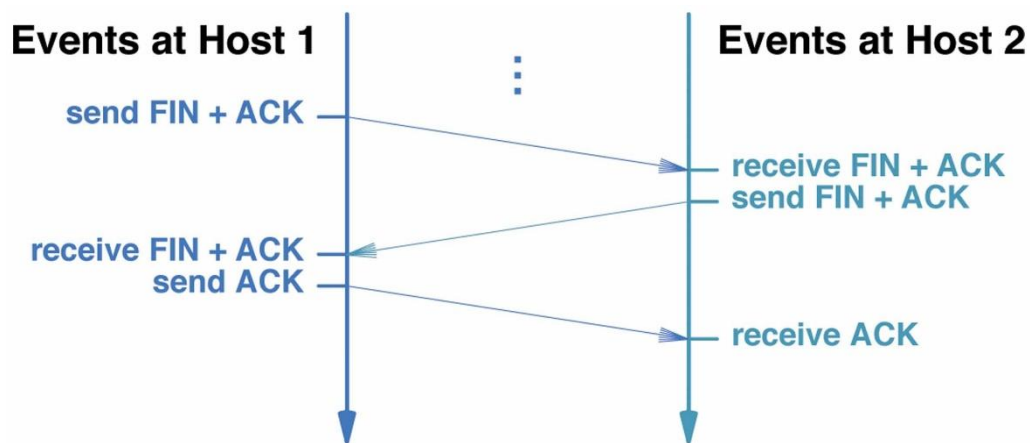
43. TCP Example ACK+SYN



44. SYN Flood Attack

- *SYN Flood Attack* is a type of Denial of Service (DoS) attack
- attacker sends a large number of TCP SYN segments without completing the third handshake step
- server sets up buffer space etc. for all SYN requests and so consumes all its resources
- solution is for server to choose as ISN a hash function of
 - source and destination IP addresses
 - source and destination port numbers
 - secret number known only to the server
- not to allocate resources until third handshake step
- nor to remember ISN
- if an ACK comes back, it can compute the hash value and check it against the ACK value (minus one)
- if no ACK, no resources have been allocated

45. TCP Connection Release



- a three-way handshake is also used to terminate a connection
- in this example, host 1 terminates the connection by transmitting a segment with the FIN flag set containing optional data
- host 2 acknowledges this (the FIN flag also consumes one byte of sequence space) and sets its own FIN flag
- the third and last segment contains host 1's acknowledgement of host 2's FIN flag

46. Congestion Control

- packet loss typically results from buffer overflow in routers as the network becomes *congested*
- congestion results from too many senders trying to send data at too high a rate
- packet retransmission treats a symptom of congestion, but not the cause
- to treat the cause, senders must be "throttled" (reduce their rate)
- TCP implements a congestion control algorithm based on perceived congestion by the sender:
 - if it perceives little congestion, it increases its send rate
 - if it perceives there is congestion, it reduces its send rate