Transport Layer Protocols

Roman Dunaytsev

The Bonch-Bruevich Saint-Petersburg
State University of Telecommunications

roman.dunaytsev@spbgut.ru

Lecture № 7

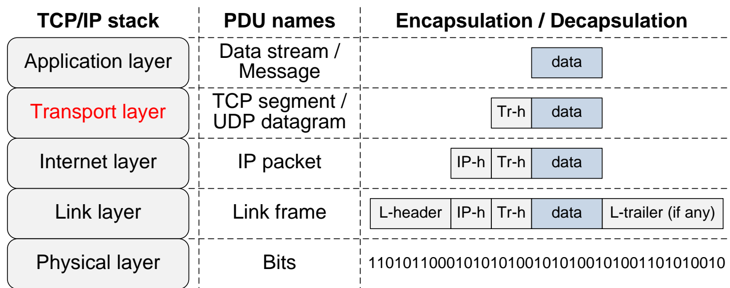
Outline

1. Transport layer
   - Multiplexing/demultiplexing
   - Port numbers
2. UDP
   - Overview
   - Datagram structure
3. TCP
   - Overview
   - Segment structure
   - Sequence numbers and ACKs
   - Connection setup
   - Data transfer
   - Connection teardown
   - Summary

Outline

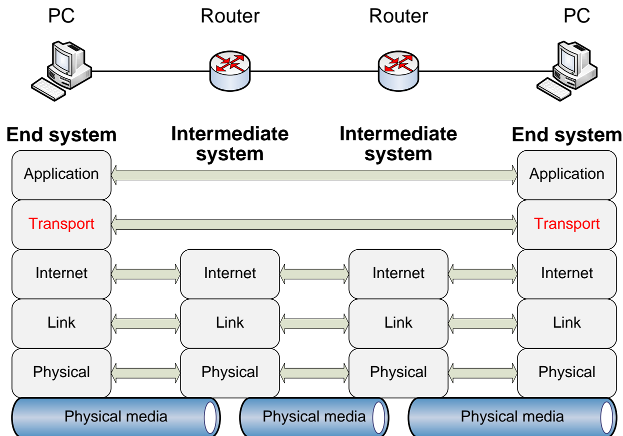
1. Transport layer
   - Multiplexing/demultiplexing
   - Port numbers
2. UDP
   - Overview
   - Datagram structure
3. TCP
   - Overview
   - Segment structure
   - Sequence numbers and ACKs
   - Connection setup
   - Data transfer
   - Connection teardown
   - Summary

- **The transport layer**
  - Resides between the application and internet layers
  - Provides an end-to-end data transfer service for application processes
  - Uses the services offered by the underlying internet layer
  - Hides the details of underlying networking from the application layer



Transport Layer (cont'd)

- Transport layer protocols are **end-to-end** protocols
  - They are only implemented at end systems (aka **hosts**)
  - Therefore, also referred to as **host-to-host** protocols

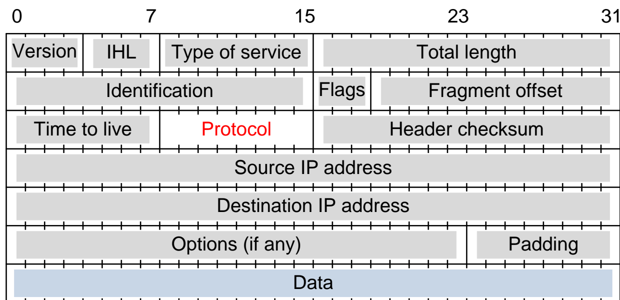


Multiplexing/Demultiplexing

- Multiple application processes are running on a host
- How to deliver data to a given application process?
  - IP provides host-to-host packets delivery but does not know how to deliver packets to a specific application process on the host
- Each IP packet header has:
  - Source and destination IP addresses
  - Protocol field which specifies the higher-layer protocol (e.g., UDP = 17 = 0x11, TCP = 6 = 0x06)
- IP demultiplexes data from incoming packets between the transport layer protocols (UDP and TCP) based on the Protocol field value

Multiplexing/Demultiplexing (cont'd)

- The values of the Protocol field are maintained by **the Internet Assigned Numbers Authority (IANA)**
- Online database is available at www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

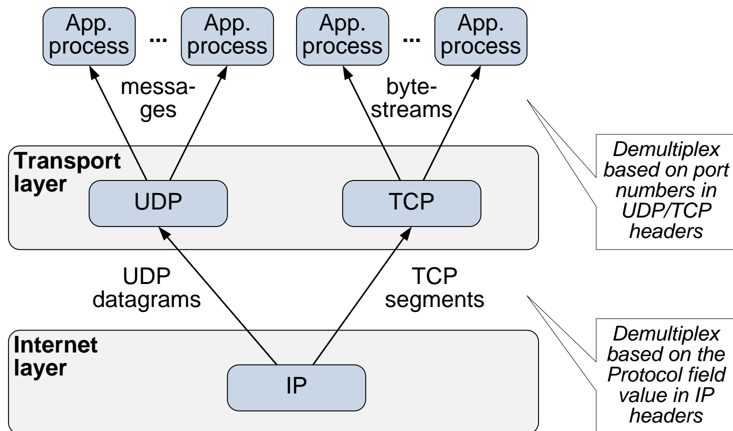


Multiplexing/Demultiplexing (cont'd)

- The transport layer protocols add a mechanism for the application process identification – **port numbers**
- The combination of the following values uniquely identifies a flow in the Internet:
  - Source and destination IP addresses
  - Source and destination port numbers
  - Protocol field value
- E.g., a TCP connection:
  130.230.52.139:1080, 130.230.137.61:80, TCP
- **Demultiplexing** – delivering incoming data to certain higher-layer entities
- **Multiplexing** – gathering data from multiple higher-layer entities, enveloping data with headers (later used for demultiplexing), and passing them to the lower layer

Multiplexing/Demultiplexing (cont'd)

- The transport layer protocols (UDP and TCP) use port numbers to identify application processes



Port Numbers

- There are $2^{16} = 65,536$ port numbers per transport layer protocol in each host
  - I.e., 65,536 UDP ports and 65,536 TCP ports
- The values of port numbers are also maintained by the IANA
  - Online database is available at www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
- **3 ranges of port numbers**:
  - Well-known ports (0 - 1023)
  - Registered ports (1024 - 49,151)
  - Dynamic and/or private ports (49,152 - 65,535)
- The IANA does not enforce adherence in use of port numbers to these assignments; it is simply a set of recommended uses

Port Numbers (cont'd)

- For the purpose of providing services to unknown callers, a service **contact port** should be preassigned
  - Similar to contact information of various authorities, companies, etc.
- Contact ports used by server processes are called **well-known ports**
  - E.g., FTP = 20 (data) and 21 (control), Telnet = 23, DNS = 53, HTTP = 80 (default) and 8080 (alternative)
- Well-known port numbers are used by servers; other port numbers are used by clients
- Client processes on a host use different port numbers, while a server process uses the same port number for all communication sessions

Port Numbers (cont'd)

- Unless a client program explicitly requests a specific port number, the port numbers used by a client process are **ephemeral ports** that are assigned by the local system and are freed up when they are no longer needed
  - Ephemeral ports are often assigned consecutively
  - Microsoft Windows operating systems use the range 1024 - 4999 for their ephemeral port range
- When the client process terminates, the ephemeral port is available for reuse, although most operating systems will not reuse that port number until the entire pool of ephemeral ports have been used

Outline

1 Transport layer
- Multiplexing/demultiplexing
- Port numbers

2 UDP
- Overview
- Datagram structure

3 TCP
- Overview
- Segment structure
- Sequence numbers and ACKs
- Connection setup
- Data transfer
- Connection teardown
- Summary

UDP

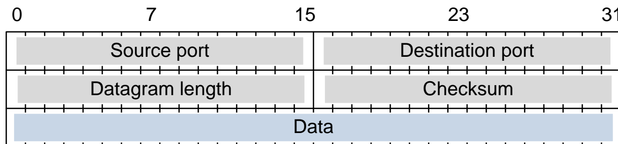
- **User Datagram Protocol (UDP)** – defined in RFC 768
- **Message-oriented**
  - UDP can preserve message boundaries
- **Connectionless**
  - Establishing a connection before sending data is not required
  - Each datagram is handled independently of others in a flow
- **Stateless**
  - Neither the sender nor the receiver has an obligation to keep track of the state of the communication session
- **Unreliable**
  - Data may be lost or delivered out-of-order to an application process
  - UDP does not use ACKs and does not retransmit lost datagrams

UDP (cont'd)

- **UDP functions**:
  - Multiplexing/demultiplexing
  - Error control (optional)
- Thus, UDP adds very little to IP
- **Datagram integrity verification**
  - IP computes checksum only for the IP header
  - UDP checksum applies to the entire UDP datagram plus a pseudo header prefixed at the time of checksum computation
- **No flow or congestion control**
  - UDP can send data as fast as desired
  - But it cannot guarantee their successful delivery
- **No feedback messages**
  - UDP can be used for both unicast or multicast communications

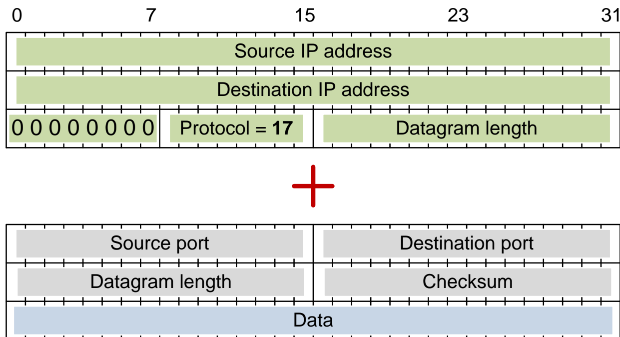
UDP Datagram Structure

- **Source port** and **destination port**, 16 bits each
  - Identify the sending and receiving application processes, respectively
- **Datagram length**, 16 bits
  - Indicates the number of bytes in the UDP datagram (including the header and data)
- **Checksum**, 16 bits – similar to computing IP checksum except:
  - If the length of the datagram is not a multiple of 16 bits, the datagram will be padded out with '0's to make it a multiple of 16 bits (actual datagram is not modified and the pad is not transmitted)
  - UDP adds a pseudo header to the datagram when performing checksum computation (only during checksum computation, the pseudo header is not transmitted)



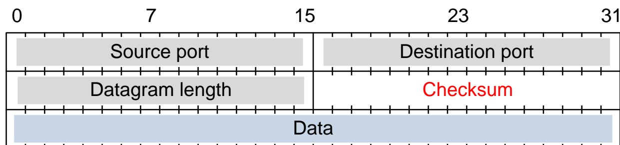
UDP Datagram Structure (cont'd)

- **The pseudo header** is needed to ensure that the datagram has indeed reached the correct destination host, the protocol type is correct (UDP = 17), and the datagram length is also correct
  - It gives protection against misrouted UDP datagrams



UDP Datagram Structure (cont'd)

- If a datagram is found to be corrupted, it is simply discarded and the source UDP entity is not notified
- If a source host does not want to compute the checksum, the checksum field should contain all '0's so that the destination host knows that the checksum has not been computed
  - What if the source host does compute the checksum and finds that the result is 0?
  - Then the checksum field should contain all '1's (another representation of 0)



UDP Datagram Structure (cont'd)

- A UDP datagram captured by **Wireshark**

```
1 0.000000 130.230.52.139 130.230.52.5 DNS Standard query A www.tut.fi
⊞ Frame 1 (70 bytes on wire, 70 bytes captured)
⊞ Ethernet II, Src: Notebook_d3:25:19 (00:06:1b:d3:25:19), Dst: Ibm_28:2e:6a (00:14:5e:28:2e:6a)
⊟ Internet Protocol, Src: 130.230.52.139 (130.230.52.139), Dst: 130.230.52.5 (130.230.52.5)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 56
    Identification: 0x0a3b (2619)
  ⊟ Flags: 0x00
      0... = Reserved bit: Not set
      .0.. = Don't fragment: Not set
      ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 128
    Protocol: UDP (0x11)
  ⊞ Header checksum: 0xc21d [correct]
    Source: 130.230.52.139 (130.230.52.139)
    Destination: 130.230.52.5 (130.230.52.5)
⊟ User Datagram Protocol, Src Port: mxxrlogin (1035), Dst Port: domain (53)
    Source port: mxxrlogin (1035)
    Destination port: domain (53)
    Length: 36
  ⊞ Checksum: 0x6bbd [correct]
⊞ Domain Name System (query)

0000  00 14 5e 28 2e 6a 00 06  1b d3 25 19 08 00 45 00   ..^(.j.. ..%...E.
0010  00 38 0a 3b 00 00 80 11  c2 1d 82 e6 34 8b 82 e6   .8.;.... ....4...
0020  34 05 04 0b 00 35 00 24  6b bd c1 0b 01 00 00 01   4....5.$ k.......
0030  00 00 00 00 00 00 03 77  77 77 03 74 75 74 02 66   .......w ww.tut.f
0040  69 00 00 01 00 01                                  i.....
```

Outline

1. Transport layer
   - Multiplexing/demultiplexing
   - Port numbers
2. UDP
   - Overview
   - Datagram structure
3. TCP
   - Overview
   - Segment structure
   - Sequence numbers and ACKs
   - Connection setup
   - Data transfer
   - Connection teardown
   - Summary

TCP

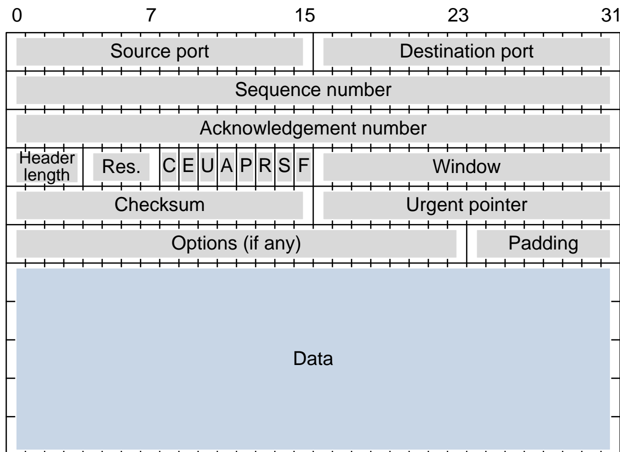
- **Transmission Control Protocol (TCP)** – defined in a number of RFCs (mainly in RFC 793, RFC 1122, and RFC 2581)
  - See RFC 4614 'A Roadmap for Transmission Control Protocol (TCP) Specification Documents'
- **Byte-stream-oriented**
  - TCP considers data as an unstructured, but ordered, stream of bytes
  - Data are delivered in-order to an application process
- **Connection-oriented**
  - A logical connection must be established before exchanging data
- **Stateful**
  - Both sender and receiver keep track of the state of the session
- **Reliable**
  - Each transmitted byte must be acknowledged by the receiving host
  - The sender retransmits, if necessary
- **Full-duplex**
  - Bi-directional data flow over the same connection

TCP (cont'd)

- **TCP functions**:
  - Multiplexing/demultiplexing
  - Ordered data transfer and data segmentation
  - Error control (mandatory)
  - Flow control
  - Congestion control
- Thus, TCP adds a lot to IP
- **Error control**
  - TCP checksum applies to the entire TCP segment plus a pseudo header prefixed at the time of checksum computation
  - TCP triggers retransmission until the data are correctly and completely received
- **Flow and congestion control**
  - TCP regulates the rate at which the sending host transmits data
- **Feedback-based**
  - Can be used for unicast communications only

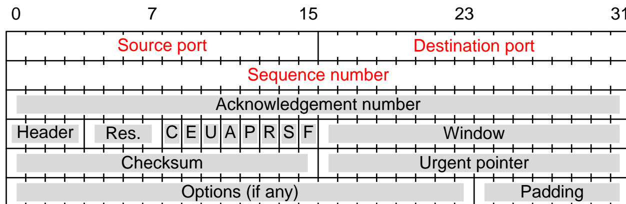
TCP Segment Structure

- A TCP segment consists of a TCP header followed by application data (if any)



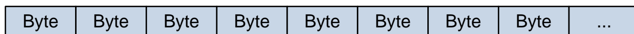
TCP Segment Structure (cont'd)

- **Source port** and **destination port**, 16 bits each
  - Identify the sending and receiving application processes, respectively
- **Sequence number**, 32 bits
  - Identifies the position of the first data byte of this segment in the sender's byte stream (when the SYN bit is not set)
  - TCP assignees a sequence number to each transmitted byte
  - If the SYN bit is set to 1 (during the connection establishment phase), this field indicates **the initial sequence number (ISN)** to be used in the sender's byte stream; the first data byte will be ISN+1
  - Both sides of a TCP connection randomly select their ISNs

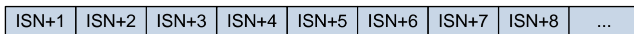


TCP Segment Structure (cont'd)

- Data to send:
  - Unstructured stream of bytes

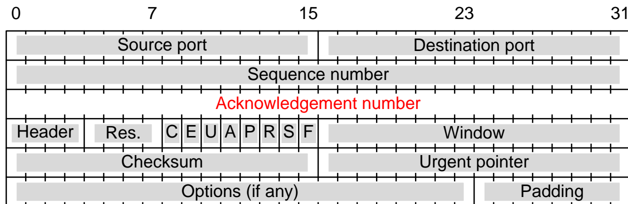


- Connection establishment phase (SYN = 1):
  - Sequence number = ISN, ISN $\in [0, 2^{32} - 1]$
- Data transfer phase (SYN = 0):
  - Ordered stream of bytes



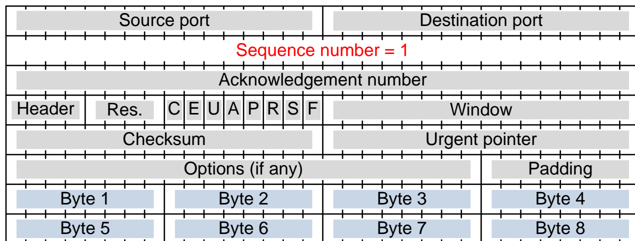
TCP Segment Structure (cont'd)

- **Acknowledgement number**, 32 bits
  - If the ACK bit is set to 1, identifies the sequence number of the next data byte that the sender expects to receive
  - Also indicates that the sender has successfully received all data up to (but not including) this value
  - Once a connection is established the ACK bit is always on
  - If the ACK bit is not set (only during the connection setup phase), the Acknowledgement number field is meaningless

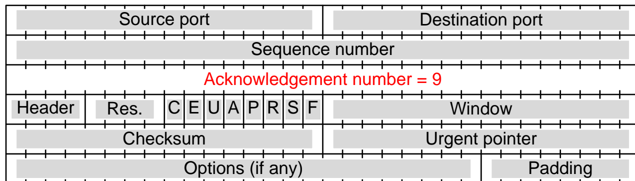


TCP Segment Structure (cont'd)

- A TCP data segment containing 8 bytes of data (ISN = 0)

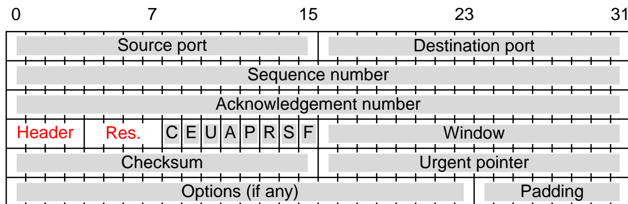


- A TCP acknowledgement (without data) for this data segment



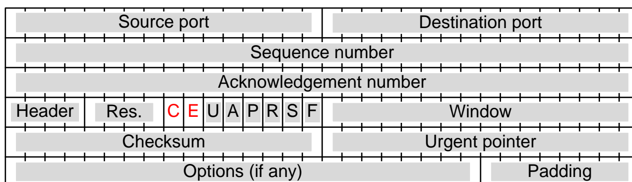
TCP Segment Structure (cont'd)

- **Header length**, 4 bits
  - Specifies the length of the TCP header in 32-bit words
  - It is needed since the Options field is of variable length
  - It indicates the beginning of the data area within the segment
  - The minimum value is 5 (20 bytes) and the maximum value is 15 (60 bytes)
- **Reserved**, 4 bits
  - Not used, must be '0's



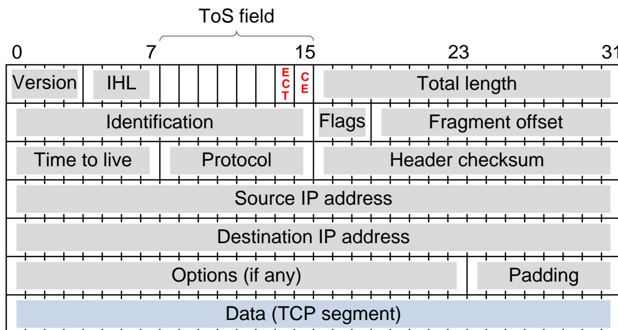
TCP Segment Structure (cont'd)

- **TCP flags**, 8 bits
- **Explicit Congestion Notification (ECN)** support in TCP uses 2 bits that were previously defined as the Reserved field (RFC 3168)
- **CWR (Congestion Window Reduced)**
  - Used by the sending host to indicate that it has received a TCP segment with the ECE flag set to 1
- **ECE (ECN-Echo)**
  - Used to indicate that a host is ECN-capable during the connection establishment phase
  - Used to indicate that a TCP segment was received with the ECN field in the IP header set to 11

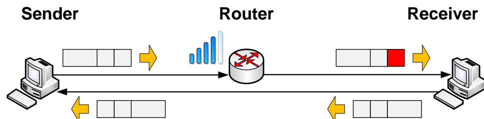


TCP Segment Structure (cont'd)

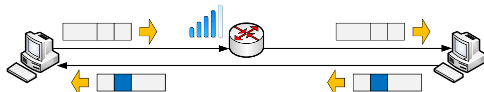
- ECN support in IP uses 2 bits of the Type of Service (ToS) field:
- **ECN-Capable Transport (ECT)**
  - It is set by the source and indicates whether the transport connection supports ECN
- **Congestion Experienced (CE)**
  - It is set by a router and indicates that congestion has been encountered



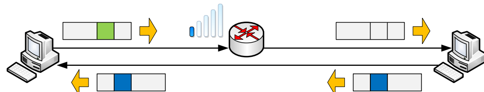
TCP Segment Structure (cont'd)



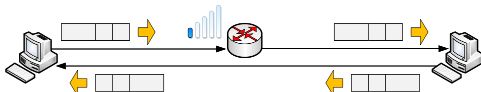
Step 1: To avoid approaching congestion, the router sets CE=1 in the IP header



Step 2: To notify the sender, the receiver sets ECE=1 in the TCP header



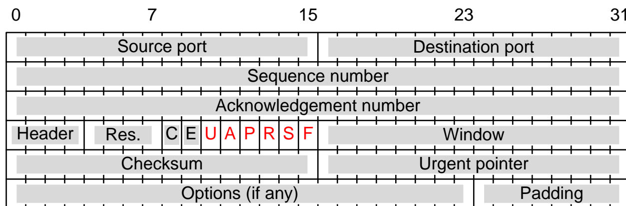
Step 3: The sender reduces its rate and sets CWR=1 in the TCP header



Step 4: Upon receipt of CWR=1, the receiver sends subsequent segments with ECE=0

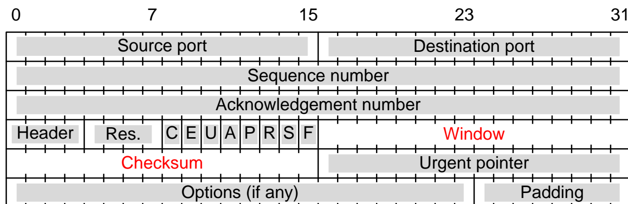
TCP Segment Structure (cont'd)

- TCP flags:
  - **URG (Urgent)** – indicates that this segment contains urgent data
  - **ACK** – indicates that the Acknowledgement number field is valid
  - **PSH (Push)** – specifies that the receiving TCP entity should pass the already received data to the application process immediately (e.g., for delay-sensitive applications)
  - **RST (Reset)** – used to abort the connection
  - **SYN** – used to establish a TCP connection
  - **FIN** – used to terminate a TCP connection



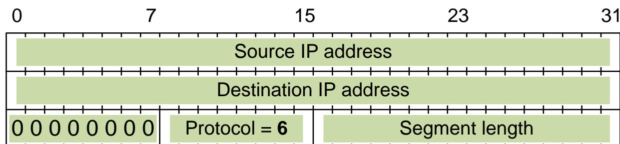
TCP Segment Structure (cont'd)

- **Window**, 16 bits
  - Specifies the number of bytes the sender of this segment is ready to accept
  - Also referred to as the receive window (rwnd)
- **Checksum**, 16 bits
  - Covers the header and data of the TCP segment to allow the receiver to verify the integrity of the incoming TCP segment
  - The TCP checksum computation is similar to that in computing UDP checksum



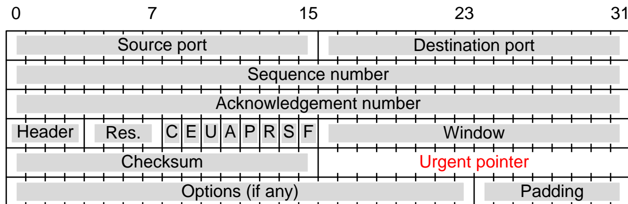
TCP Segment Structure (cont'd)

- If the length of the segment is not a multiple of 16 bits, the segment will be padded out with '0's to make it a multiple of 16 bits
  - The actual segment is not modified and the pad is not transmitted
- TCP adds a pseudo header to the segment when performing checksum computation (only during checksum computation, the pseudo header is not transmitted)
  - It gives protection against misrouted TCP segments
- If a segment is found to be corrupted, it is discarded
- But in contrast to UDP, the source TCP entity will be notified (by the lack of the ACK for this segment)



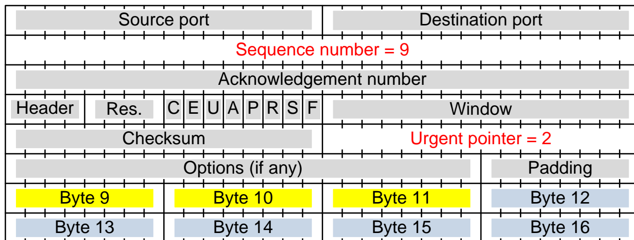
TCP Segment Structure (cont'd)

- **Urgent pointer**, 16 bits
  - If the URG bit is set to 1, specifies a positive offset that must be added to the Sequence number field value of the segment to yield the sequence number of the last byte of urgent data (RFC 1122)
  - This allows the receiver to know how much urgent data are coming
  - TCP's urgent mode is a way for the sender to transmit urgent (aka **out-of-band**) data to the other end
  - E.g., when the Telnet or Rlogin user types the 'Interrupt' key



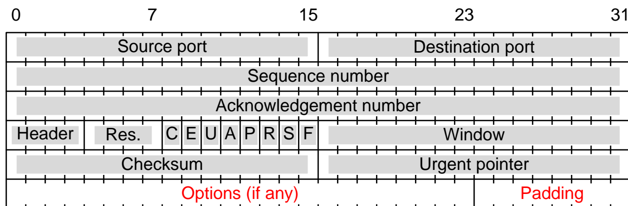
TCP Segment Structure (cont'd)

- A TCP segment with 8 bytes of data
  - Bytes 9, 10, and 11 contain urgent data
  - The last byte of urgent data = Sequence number + Urgent pointer = $= 9 + 2 = 11$



TCP Segment Structure (cont'd)

- **Options**, variable length
  - TCP options extend TCP functionality
  - TCP options can be comprised of a single byte or multiple bytes
  - A host is not required to support all TCP options
- **Padding**, variable length
  - If the length of the header is not a multiple of 32 bits (due to the included options), the header will be padded out with '0's to make it a multiple of 32 bits

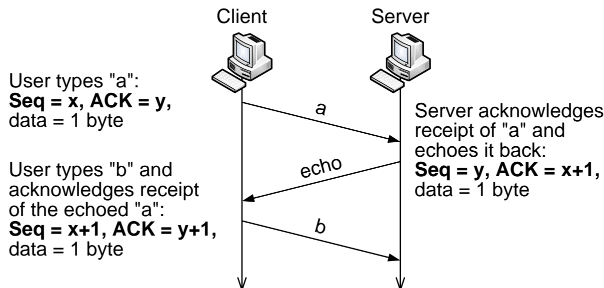


Sequence Numbers and ACKs

- 2 important fields in the TCP header in providing a reliable data transfer service:
  - Sequence number
  - Acknowledgement number
- To illustrate the use of sequence numbers and acknowledgements (ACKs), let us consider the Telnet application
- **Telnet** is defined in RFC 854
- The purpose of Telnet is to provide a general, full-duplex, 8-bit-oriented communications facility
- The term 'telnet' also refers to software which implements the client part of the protocol

Sequence Numbers and ACKs (cont'd)

- Each character typed by the user is sent to the server
  - Character length is 1 byte
- The server sends back a copy of each character (echo)
- Thus, each character traverses the network twice between the time the user hits the key and the time the character is displayed on the user's monitor



Sequence Numbers and ACKs (cont'd)

- ACKs can be **piggybacked**
  - A data segment from host A to host B can also contain an ACK for data sent in the direction from B to A
  - This feature can help to reduce the number of packets transmitted, since the ACK of received data does not have to travel in a packet separate from those that hold data

Sequence Numbers and ACKs (cont'd)

- ACKs are **cumulative**
  - E.g., the sender sends 2 segments with 1 - 1460 and 1461 - 2920 bytes, but the receiver only gets the second segment
  - In this case, the receiver can not acknowledge the second segment, it can only send ACK = 1
  - However, once the first segment arrives, the receiver will send a single ACK for both segments (ACK = 2921)
  - In RFC 2018, a Selective Acknowledgment (SACK) mechanism has been introduced
- Pure ACKs (i.e., segments without data) are not retransmitted
  - Since an ACK for the next segment will acknowledge all successfully received segments up to this moment

Sequence Numbers and ACKs (cont'd)

- Most TCP implementations use the **delayed** ACK algorithm
  - This allows to send an ACK and application data to the sender in a single TCP segment
- Send an ACK back if one of the following conditions is met:
  - No ACK was sent for the previous segment received
  - A segment is received, but no other segment arrives within 500 ms (typically, 200 ms)
  - An incoming segment fills in all or part of a gap in the sequence space
- Moreover, generate an immediate ACK when an out-of-order segment is received (aka a **duplicate** ACK)
  - The purpose of this duplicate ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected

Connection Setup

- In order to reliably transfer data between hosts, TCP first establishes a logical connection
- To establish a TCP connection, 3 segments are sent between a client and a server (aka **the 3-way handshake**)
- The client-side TCP (host A) first sends a special TCP segment to the server-side TCP (host B)
  - This segment does not contain any application-layer data
  - The SYN bit of this segment is set to 1
  - For this reason, it is referred to as **a SYN segment**
  - In addition, the client chooses an initial sequence number, ISN(A), and puts this number in the Sequence number field of the segment
  - This segment is encapsulated into an IP packet and sent to the server
- Thus, the client performs **an active open**

Connection Setup (cont'd)

- E.g., a TCP connection:
- Step 1: the SYN segment

```
1 0.000000 130.230.52.139 130.230.1.66 TCP socks > http [SYN] Seq=3180262531 Win=32768 Len=0 MSS=1460
⊞ Frame 1 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: Notebook_d3:25:19 (00:06:1b:d3:25:19), Dst: All-HSRP-routers_34 (00:00:0c:07:ac:34)
⊞ Internet Protocol, Src: 130.230.52.139 (130.230.52.139), Dst: 130.230.1.66 (130.230.1.66)
⊟ Transmission Control Protocol, Src Port: socks (1080), Dst Port: http (80), Seq: 3180262531, Len: 0
    Source port: socks (1080)
    Destination port: http (80)
    Sequence number: 3180262531
    Header length: 28 bytes
  ⊟ Flags: 0x02 (SYN)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...0 .... = Acknowledgment: Not set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
      .... ..1. = Syn: Set
      .... ...0 = Fin: Not set
    Window size: 32768
  ⊞ Checksum: 0x10eb [correct]
  ⊟ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted

0000  00 00 0c 07 ac 34 00 06  1b d3 25 19 08 00 45 00   .....4.. ..%...E.
0010  00 30 06 52 40 00 80 06  b8 dc 82 e6 34 8b 82 e6   .0.R@... ....4...
0020  01 42 04 38 00 50 bd 8e  f4 83 00 00 00 00 70 02   .B.8.P.. ......p.
0030  80 00 10 eb 00 00 02 04  05 b4 01 01 04 02         ........ ......
```

Connection Setup (cont'd)

- Once the SYN segment arrives at the server-side TCP, the server (host B) allocates a buffer and variables for this connection
- Then it sends a connection-granted segment to the client, which also does not carry any application-layer data, but it contains 3 important pieces of information in its header:
  - The SYN bit is set to 1
  - The Acknowledgment number field contains ISN(A)+1
  - The server chooses its own initial sequence number, ISN(B), and puts this value in the Sequence number field of the segment header
- The connection-granted segment is referred to as **a SYN/ACK segment**
- Thus, the server performs **a passive open**

Connection Setup (cont'd)

- Step 2: the SYN/ACK segment

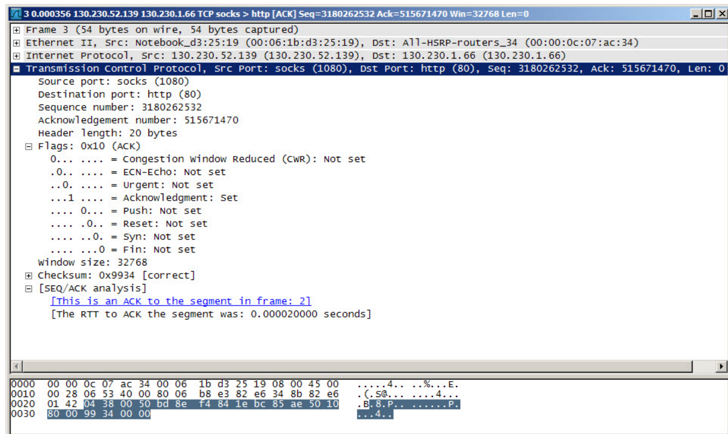
```
2 0.000336 130.230.1.66 130.230.52.139 TCP http > socks [SYN, ACK] Seq=515671469 Ack=3180262532 Win=5840 Len=0 MSS=1460
⊞ Frame 2 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: Cisco_18:4e:00 (00:0f:34:18:4e:00), Dst: Notebook_d3:25:19 (00:06:1b:d3:25:19)
⊞ Internet Protocol, Src: 130.230.1.66 (130.230.1.66), Dst: 130.230.52.139 (130.230.52.139)
⊟ Transmission Control Protocol, Src Port: http (80), Dst Port: socks (1080), Seq: 515671469, Ack: 3180262532, Len: 0
    Source port: http (80)
    Destination port: socks (1080)
    Sequence number: 515671469
    Acknowledgement number: 3180262532
    Header length: 28 bytes
  ⊟ Flags: 0x12 (SYN, ACK)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...1 .... = Acknowledgment: Set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
      .... ..1. = Syn: Set
      .... ...0 = Fin: Not set
    Window size: 5840
  ⊞ Checksum: 0xd5a0 [correct]
  ⊟ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted
  ⊞ [SEQ/ACK analysis]

0000  00 06 1b d3 25 19 00 0f  34 18 4e 00 08 00 45 00   ....%... 4.N...E.
0010  00 30 00 00 40 00 3e 06  01 2f 82 e6 01 42 82 e6   .0..@.>. ./...B..
0020  34 8b 00 50 04 38 1e bc  85 ad bd 8e f4 84 70 12   4..P.8.. ......p.
0030  16 d0 d5 a0 00 00 02 04  05 b4 01 01 04 02         ........ ......
```

Connection Setup (cont'd)

- Upon receiving this connection-granted segment, the client also allocates a buffer and variables for this connection
- Then the client sends to the server one more segment
- This segment acknowledges the server's connection-granted segment
  - The client does so by putting the value of ISN(B)+1 in the Acknowledgment number field of the segment header
- Since the connection is established, the SYN bit is set to 0
  - Thus, it is just an ACK segment

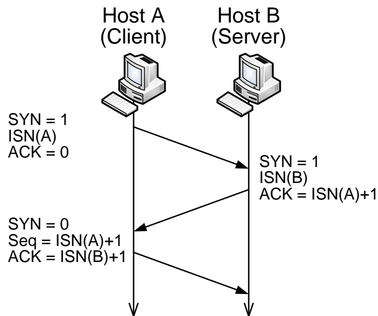
Connection Setup (cont'd)

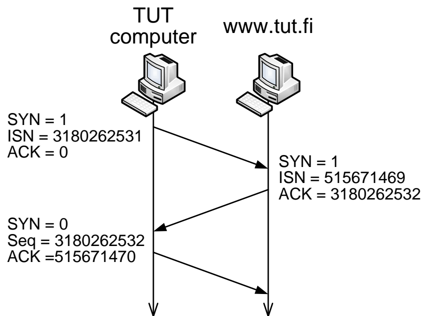
- Step 3: the ACK segment



Connection Setup (cont'd)

- 3-way handshake



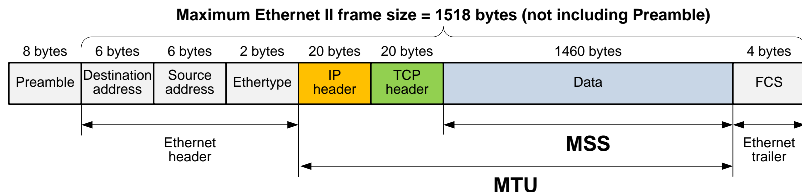


Data Transfer

- Once a TCP connection is established, the client and server hosts can send segments containing data to each other
- In each of these subsequent segments, the SYN bit will be set to 0
- **How it works (sender side)**:
  1. An application process passes a stream of data to the TCP send buffer
  2. From time to time TCP grabs chunks of data from the send buffer
  3. The maximum amount of data that can be placed into a segment is limited by the Maximum Segment Size
  4. TCP encapsulates each chunk of data into a TCP segment (i.e., adds a TCP header), thereby forming TCP segments
  5. These segments are passed down to the internet layer, where they are separately encapsulated within IP packets
  6. Then IP packets are sent to the network

Data Transfer (cont'd)

- **How it works (receiver side)**:
  1. When TCP receives a segment, it places the segment's data to the TCP receive buffer
  2. The corresponding application process reads data from this buffer
- **The Maximum Segment Size (MSS)** is equal to the Maximum Transmission Unit (MTU) minus the size of IP and TCP headers without options
  - For IPv4: MSS = MTU – 40 bytes
  - For IPv6: MSS = MTU – 60 bytes
- E.g., for IPv4 over Ethernet II: MSS = 1500 – 40 = 1460 bytes

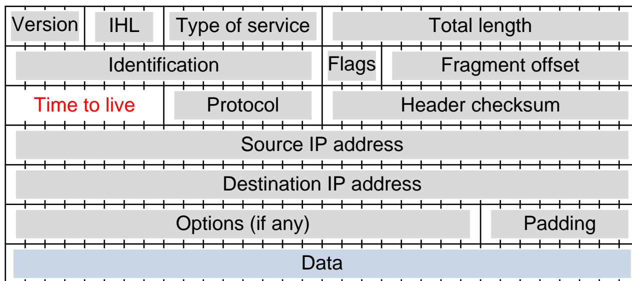


Connection Teardown

- Either of the 2 application processes participating in a TCP connection can terminate the connection
- When a TCP connection ends, the resources (buffers and variables) in the hosts are deallocated
- Suppose the server (host B) decides to close the connection:
    1. The server's application process issues a 'Close' command
    2. So the server sends a TCP segment with the FIN bit set to 1
    3. Thus, the server performs **an active close**
    4. When the client receives this FIN segment, it sends an ACK back
    5. Then the client's application process issues a 'Close' command
    6. The client sends its own TCP segment with the FIN bit set to 1
    7. Thus, the client performs **a passive close**
    8. Finally, the server acknowledges the client's FIN segment

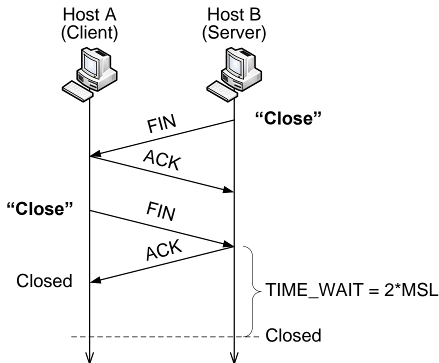
Connection Teardown (cont'd)

- When TCP does an active close and sends the final ACK, the connection must stay in the TIME_WAIT state for twice **the Maximum Segment Lifetime (MSL)**
  - MSL is the time a TCP segment can exist in the network
  - MSL is enforced by **the Time to Live (TTL)** field in an IP header
  - Typically, MSL is set to 2 minutes, 1 minute, or 30 seconds
  - If a retransmission of the final FIN arrives while TCP in the TIME_WAIT state, this ACK is retransmitted and the timer is restarted



Connection Teardown (cont'd)

- The TIME_WAIT state is required for 2 main reasons:
  - To provide enough time to ensure that the ACK is received by the other host and to retransmit it if needed
  - To provide a 'buffering period' between the end of this connection and any subsequent ones



Connection Teardown (cont'd)

- E.g., a TCP connection:
- Step 1: the active close FIN segment

```
173 23.455973 130.230.1.66 130.230.52.139 TCP http > ansoft-lm-1 [FIN, ACK] Seq=513913797 Ack=340003931 Win=8576 Len=0
⊞ Frame 173 (60 bytes on wire, 60 bytes captured)
⊞ Ethernet II, Src: Cisco_18:4e:00 (00:0f:34:18:4e:00), Dst: Notebook_d3:25:19 (00:06:1b:d3:25:19)
⊞ Internet Protocol, Src: 130.230.1.66 (130.230.1.66), Dst: 130.230.52.139 (130.230.52.139)
⊟ Transmission Control Protocol, Src Port: http (80), Dst Port: ansoft-lm-1 (1083), Seq: 513913797, Ack: 340003931, Le
    Source port: http (80)
    Destination port: ansoft-lm-1 (1083)
    Sequence number: 513913797
    Acknowledgement number: 340003931
    Header length: 20 bytes
  ⊟ Flags: 0x11 (FIN, ACK)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...1 .... = Acknowledgment: Set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
      .... ..0. = Syn: Not set
      .... ...1 = Fin: Set
    Window size: 8576
  ⊞ Checksum: 0x5b29 [correct]
  ⊟ [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 160]
      [The RTT to ACK the segment was: 15.963839000 seconds]

0000  00 06 1b d3 25 19 00 0f  34 18 4e 00 08 00 45 00   ....%... 4.N...E.
0010  00 28 9a 0d 40 00 3e 06  67 29 82 e6 01 42 82 e6   .(..@.>. g)...B..
0020  34 8b 00 50 04 3b 1e a1  b3 c5 14 44 0c 5b 50 11   4..P.;.. ...D.[P.
0030  21 80 5b 29 00 00 00 00  00 00 00 00               !.[)..  ....
```

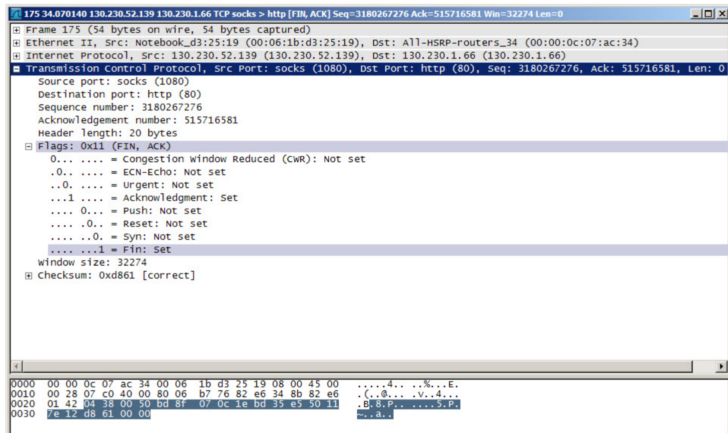
Connection Teardown (cont'd)

- Step 2: the ACK for the active close FIN segment

```
174 23.455999 130.230.52.139 130.230.1.66 TCP ansoft-lm-1 > http [ACK] Seq=340003931 Ack=513913798 Win=32155 Len=0
⊞ Frame 174 (54 bytes on wire, 54 bytes captured)
⊞ Ethernet II, Src: Notebook_d3:25:19 (00:06:1b:d3:25:19), Dst: All-HSRP-routers_34 (00:00:0c:07:ac:34)
⊞ Internet Protocol, Src: 130.230.52.139 (130.230.52.139), Dst: 130.230.1.66 (130.230.1.66)
⊟ Transmission Control Protocol, Src Port: ansoft-lm-1 (1083), Dst Port: http (80), Seq: 340003931, Ack: 513913798, Le
    Source port: ansoft-lm-1 (1083)
    Destination port: http (80)
    Sequence number: 340003931
    Acknowledgement number: 513913798
    Header length: 20 bytes
  ⊟ Flags: 0x10 (ACK)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...1 .... = Acknowledgment: Set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
      .... ..0. = Syn: Not set
      .... ...0 = Fin: Not set
    Window size: 32155
  ⊞ Checksum: 0xff0d [correct]
  ⊟ [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 173]
      [The RTT to ACK the segment was: 0.000026000 seconds]

0000  00 00 0c 07 ac 34 00 06  1b d3 25 19 08 00 45 00   .....4.. ..%...E.
0010  00 28 07 b9 40 00 80 06  b7 7d 82 e6 34 8b 82 e6   .(..@... .}..4...
0020  01 42 04 3b 00 50 14 44  0c 5b 1e a1 b3 c6 50 10   .B.;.P.D .[....P.
0030  7d 9b ff 0d 00 00                                  }.....
```

Connection Teardown (cont'd)

- Step 3: the passive close FIN segment



Connection Teardown (cont'd)

- Step 4: the ACK for the passive close FIN segment

```
176 34.070537 130.230.1.66 130.230.52.139 TCP http > socks [ACK] Seq=515716581 Ack=3180267277 Win=8576 Len=0
⊞ Frame 176 (60 bytes on wire, 60 bytes captured)
⊞ Ethernet II, Src: Cisco_18:4e:00 (00:0f:34:18:4e:00), Dst: Notebook_d3:25:19 (00:06:1b:d3:25:19)
⊞ Internet Protocol, Src: 130.230.1.66 (130.230.1.66), Dst: 130.230.52.139 (130.230.52.139)
⊟ Transmission Control Protocol, Src Port: http (80), Dst Port: socks (1080), Seq: 515716581, Ack: 3180267277, Len: 0
    Source port: http (80)
    Destination port: socks (1080)
    Sequence number: 515716581
    Acknowledgement number: 3180267277
    Header length: 20 bytes
  ⊟ Flags: 0x10 (ACK)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...1 .... = Acknowledgment: Set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
      .... ..0. = Syn: Not set
      .... ...0 = Fin: Not set
    Window size: 8576
  ⊞ Checksum: 0x34f4 [correct]
  ⊟ [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 175]
      [The RTT to ACK the segment was: 0.000397000 seconds]

0000  00 06 1b d3 25 19 00 0f  34 18 4e 00 08 00 45 00   ....%... 4.N...E.
0010  00 28 00 00 40 00 3e 06  01 37 82 e6 01 42 82 e6   .(..@.>. .7...B..
0020  34 8b 00 50 04 38 1e bd  35 e5 bd 8f 07 0d 50 10   4..P.8.. 5.....P.
0030  21 80 34 f4 00 00 00 00  00 00 00 00               !.4..... ....
```

TCP Functions

TCP function	Implementation
Multiplexing/demultiplexing	Port numbers
Ordered data transfer and data segmentation	Connection establishment and termination MSS option Path MTU discovery
Error control	Checksum computation Sequence numbers Protection against wrapped sequences Cumulative and selective ACKs Retransmission timer and retransmissions
Flow control	Receive window Silly window syndrome avoidance Nagle algorithm Window scale option
Congestion control	Karn's algorithm Initial window Slow start Congestion avoidance Fast retransmit and fast recovery ECN-support

UDP vs. TCP

- Why is there UDP?
  - No connection establishment $\Rightarrow$ no signaling overhead
  - No connection state at the end hosts $\Rightarrow$ few resources are required
  - Small header $\Rightarrow$ small control overhead
  - Error control is optional $\Rightarrow$ suitable for loss-tolerant applications
  - No flow or congestion control $\Rightarrow$ unbounded sending rate
  - Simple implementation $\Rightarrow$ flexibility and scalability

UDP	**TCP**
Message-oriented	Byte-stream-oriented
Connectionless	Connection-oriented
Stateless	Stateful
Unreliable	Reliable
Unicast and multicast	Unicast only
Used by a few user applications (VoIP, multimedia streaming, etc.)	Used by many user applications (WWW, email, FTP, Telnet, etc.)
Used by many network services (RIP, SNMP, DNS, etc.)	Used by a few network services (e.g., DNS zone transfers)

Data Integrity Verification

Protocol	Checksum field, bits	Algorithm	Checksum covers	
			Header	Data
SLIP	No			
PPP	16 or 32	CRC-16/-32	Yes	Yes
Ethernet Token Bus / Ring Wi-Fi FDDI	32	CRC-32	Yes	Yes
IPv4	16	Internet checksum	Yes	No
IPv6	No			
ICMPv4 ICMPv6	16	Internet checksum	Yes	Yes
UDP	16	Internet checksum	Yes + pseudoheader	Yes
TCP	16	Internet checksum	Yes + pseudoheader	Yes