Transport Layer

Dr. Xiqun Lu

College of Computer Science

Zhejiang University

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

The Transport Layer

- Together with the network layer, the transport layer is the heart of the protocol hierarchy.
  - The transport layer provides efficient, reliable, cost-effective data transport from the source machine to the destination machine.
  - <u>The transport layer provides **end-to-end** connectivity across the network</u>.

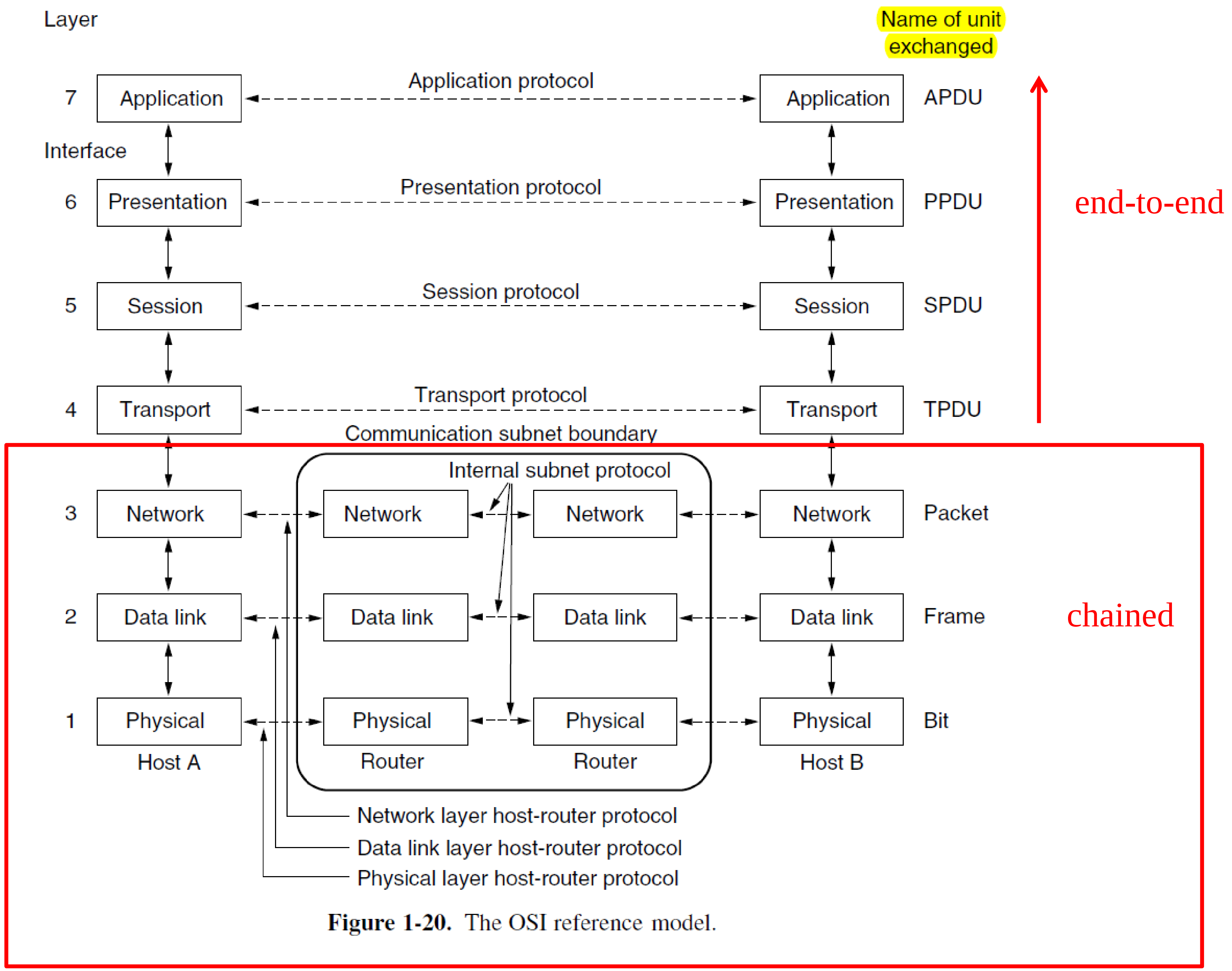


Figure 1-20. The OSI reference model.

The Relationship of the Network, Transport, and Application Layers

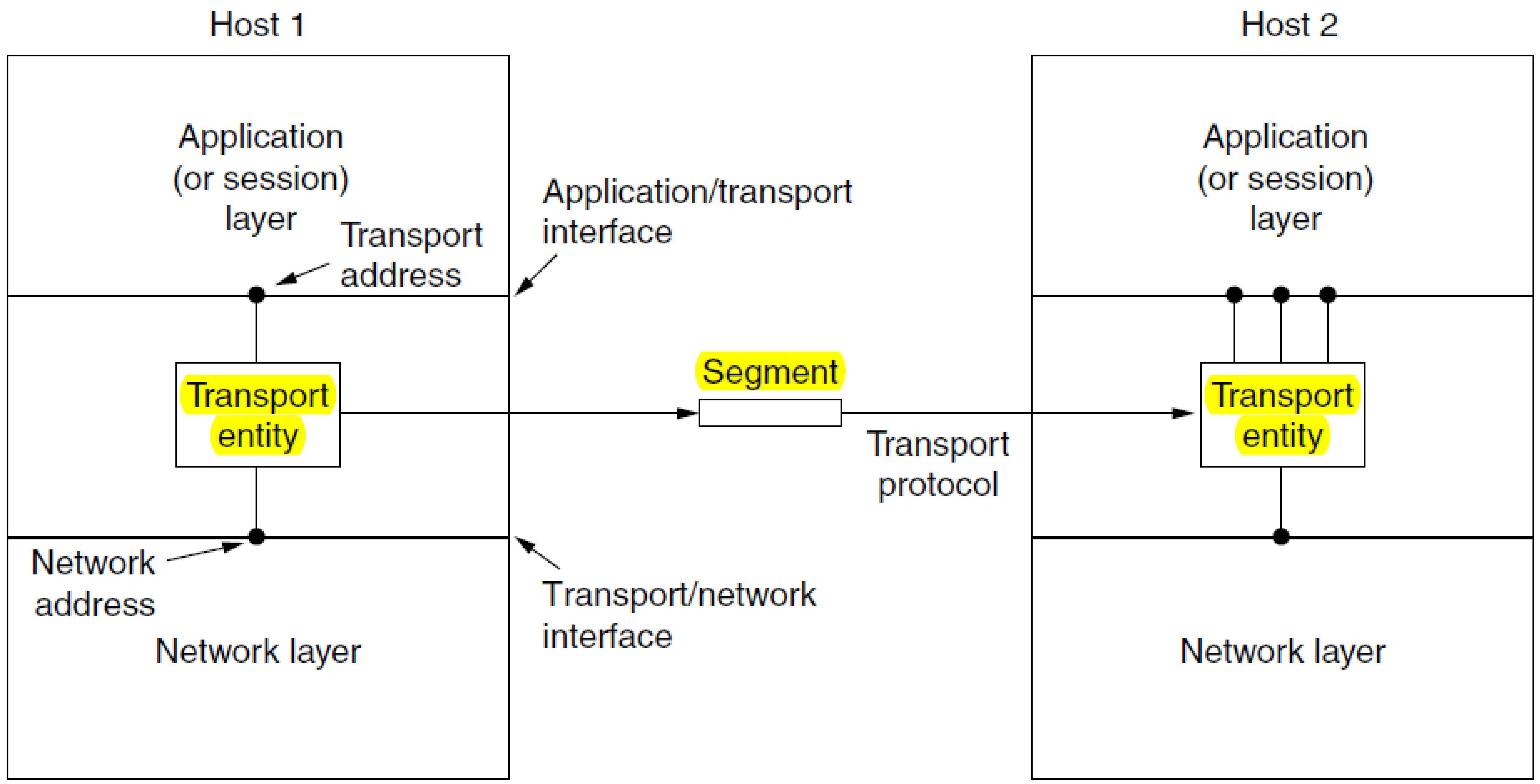


Figure 6-1. The network, transport, and application layers.

The Transport Service

- Two types of transport service
  - Connection-oriented transport service.
    - Connections has three phases: establishment, data transfer, and release
  - Connectionless transport service
- The transport layer service is so similar to the network layer service, why are there two distinct layers?
  - <u>The transport code runs entirely on the users' machines, but the network layer mostly runs on the routers.</u> The users have no real control over the network layer.
  - The network service is generally **unreliable**.
  - The only possibility is to put on top of the network layer another layer that improves the quality of the service.
    - In a **connectionless** network, if packets are lost or mangled, the transport entity can detect the problem and compensate for it by using retransmissions.
    - In a **connection-oriented** network, if a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated, with no indication of what has happened to the data currently in transit, it can set up a new network connection to the remote transport entity. Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and knowing where it was, pick up from there it left off.

Transit Units of Different Layers

- Transport layer: segment or TPDU (Transport Protocol Data Unit)
- Network layer: packet
- Data link layer: frame
- Physical layer: bit

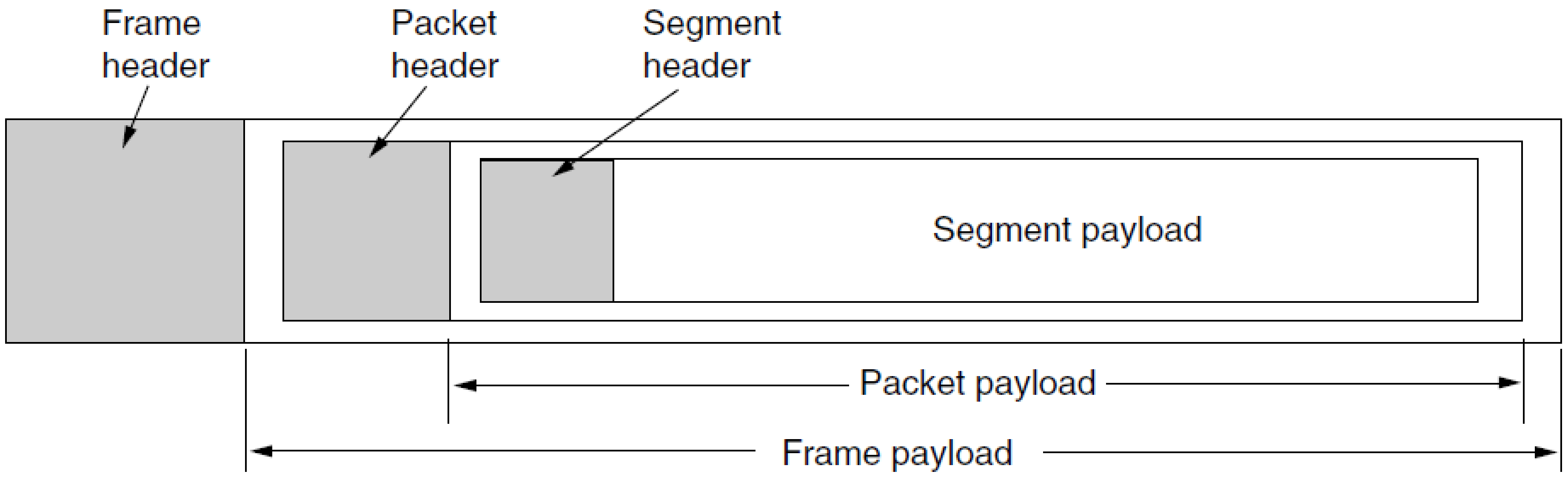


Figure 6-3. Nesting of segments, packets, and frames.

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

The Internet Transport Protocols

- The Internet has *two main protocols* in the transport layer
  - **UDP** (**User Datagram Protocol**, *connectionless* protocol): It does nothing beyond sending packets between applications. It typically runs in the operating system.
  - **TCP** (*connection-oriented* protocol): It does almost everything. It makes connections and adds reliability with retransmission, along with flow control and congestion control.

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
    - RTP (Real-Time Transport Protocol) & RTCP (Real-time Transport Control Protocol)
  - TCP

UDP

- UDP: **connectionless** transport protocol
  - RFC768
  - UDP header (8 bytes)

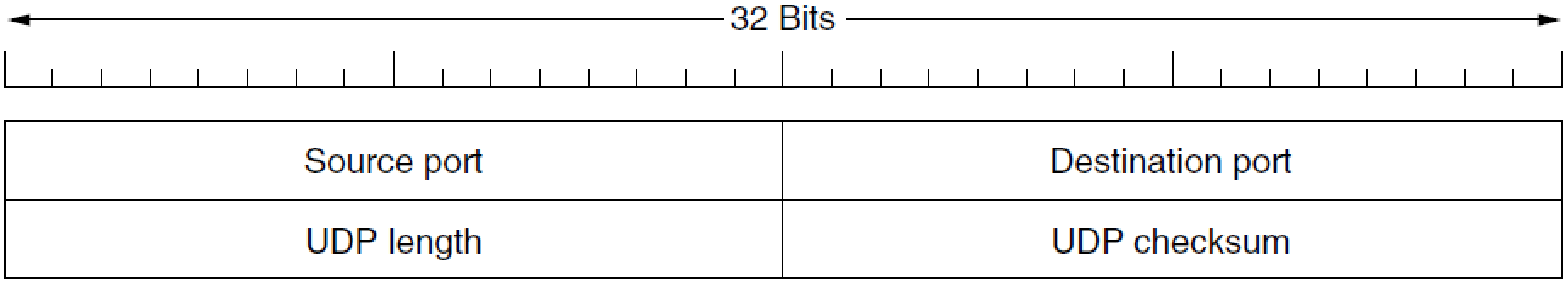


Figure 6-27. The UDP header.

- **The two ports** serve to identify the endpoints within the source and destination machines.
  - With these two ports, it delivers the embedded segment to the correct application.
  - The source port is primarily needed when a reply must be sent back to the source. By copying the Source Port field from the incoming segment into the Destination Port field of the outgoing segment.

UDP (II)

- UDP header (8 bytes)
  - The UDP length field includes the 8-byte header and the data.
    - The minimum length is 8 bytes, to cover the header.
    - The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.
  - The UDP checksum field (optional) is to provide extra reliability.
    - It checksums the header, data, and a *conceptual IP pseudoheader*.

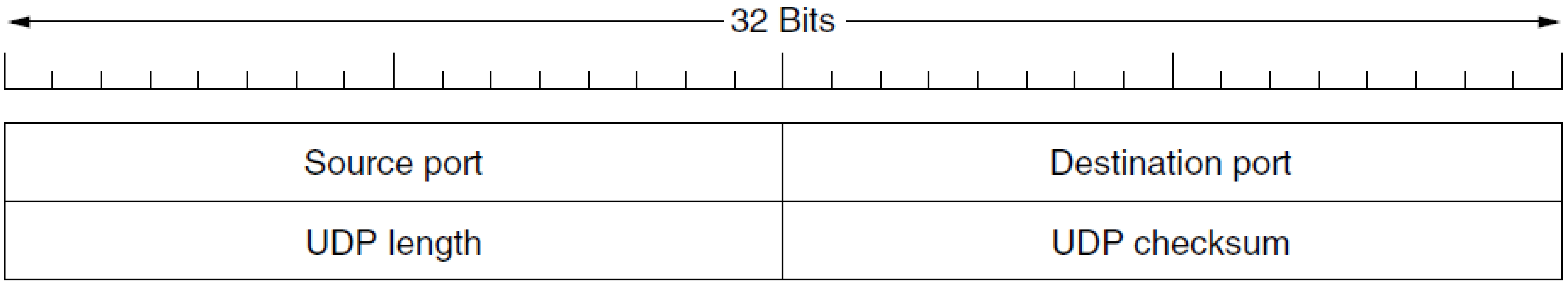


Figure 6-27. The UDP header.

UDP (III)

- Performing the checksum computation
  - The Checksum field is set to zero.
  - The data field is padded out with an additional zero byte if its length is an odd number of bytes.
  - The checksum algorithm is simply to add up all the 16-bit words (note here a word = 16 bits = 2 bytes) in one's complement and to take the one's complement of the sum.

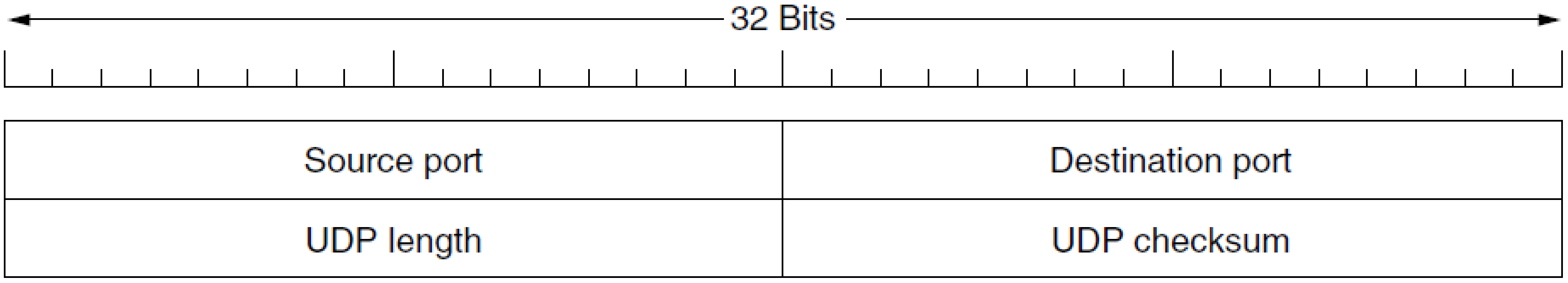


Figure 6-27. The UDP header.

UDP (IV)

- The IPv4 pseudoheader
  - The 32 bit IPv4 addresses of the source and destination machines.
    - Including the pseudoheader in the UDP checksum computation helps detect mis-delivered packets, but including it also violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not to the UDP layer.
  - The protocol number of UDP (17)
  - The byte count of the UDP segment (including the header).

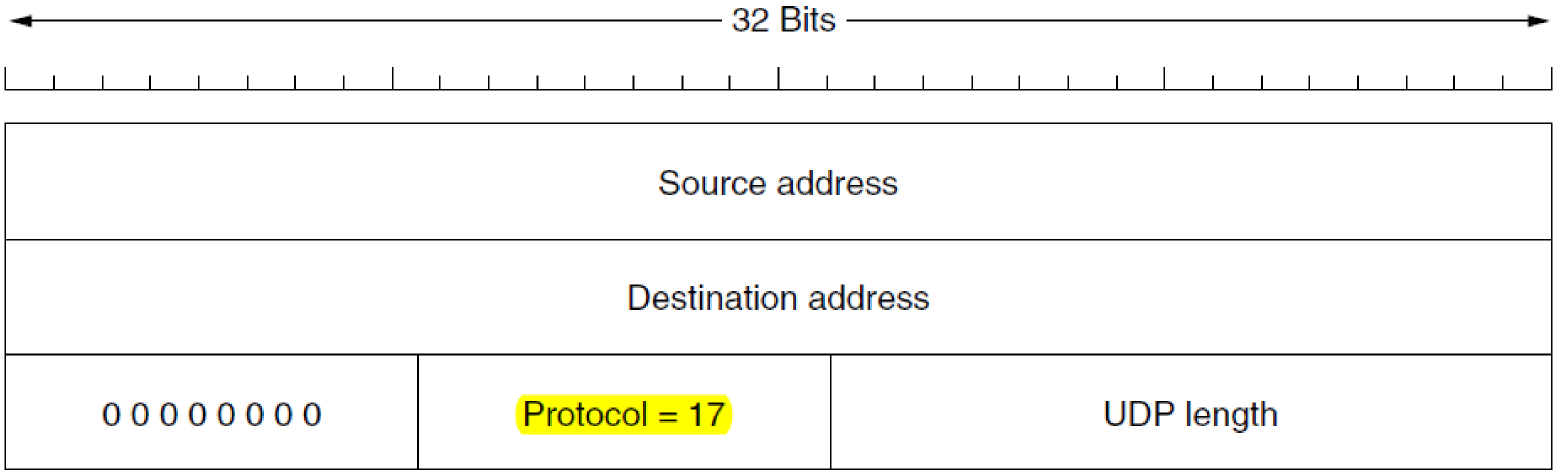
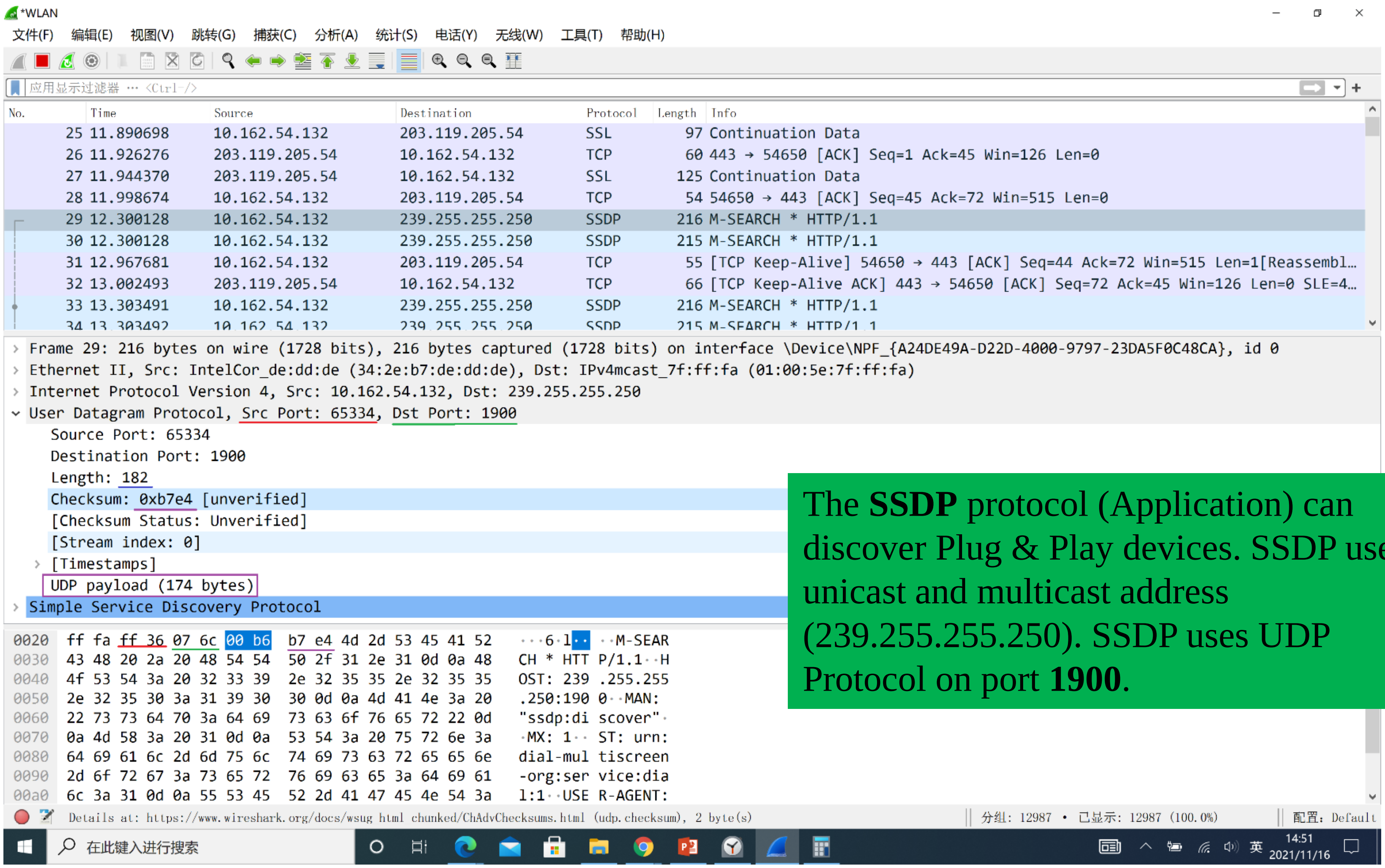


Figure 6-28. The IPv4 pseudoheader included in the UDP checksum.

An UDP Example

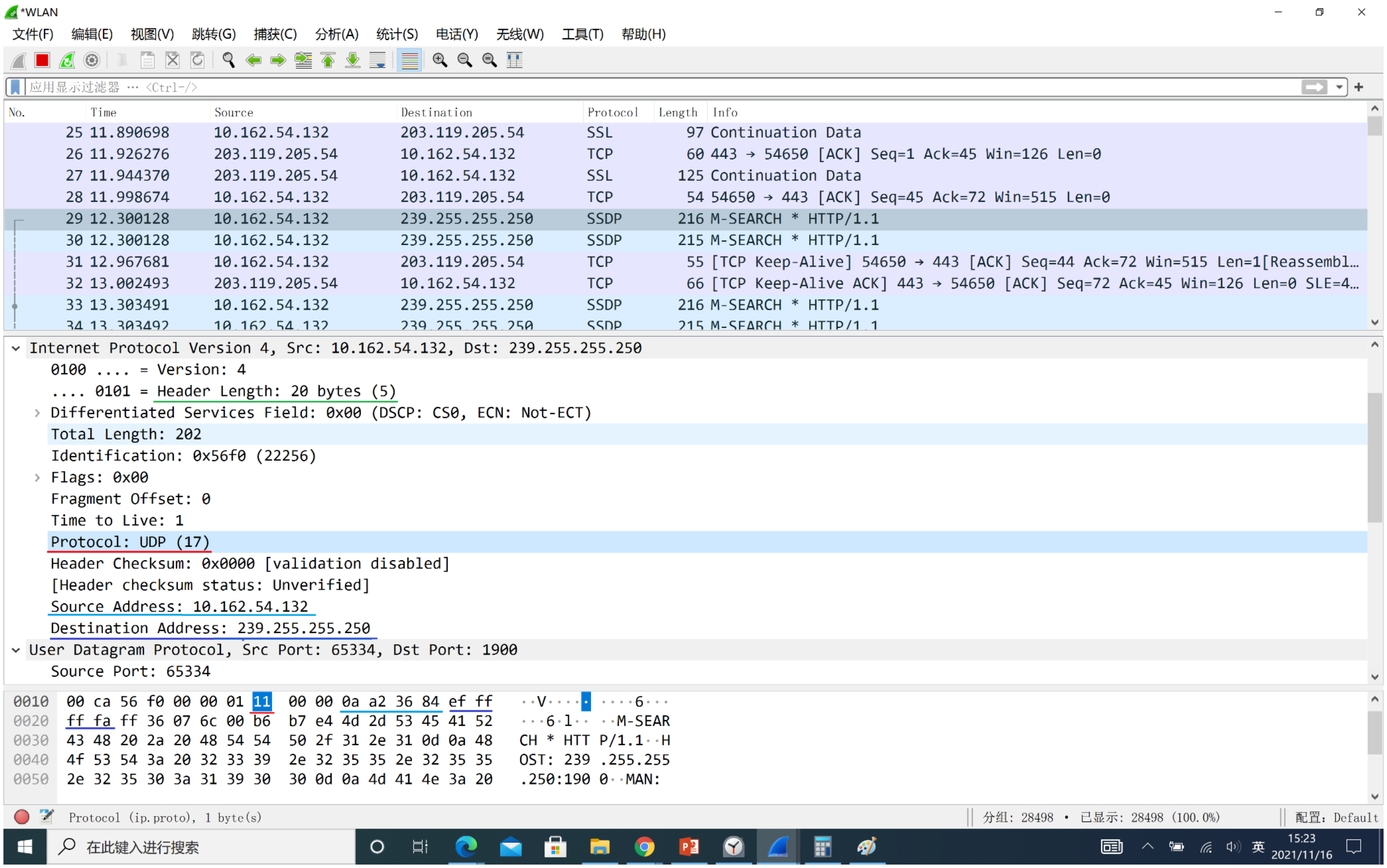


```
No.  Time       Source          Destination      Protocol Length Info
  25 11.890698  10.162.54.132   203.119.205.54   SSL       97 Continuation Data
  26 11.926276  203.119.205.54  10.162.54.132    TCP       60 443 → 54650 [ACK] Seq=1 Ack=45 Win=126 Len=0
  27 11.944370  203.119.205.54  10.162.54.132    SSL      125 Continuation Data
  28 11.998674  10.162.54.132   203.119.205.54   TCP       54 54650 → 443 [ACK] Seq=45 Ack=72 Win=515 Len=0
  29 12.300128  10.162.54.132   239.255.255.250  SSDP     216 M-SEARCH * HTTP/1.1
  30 12.300128  10.162.54.132   239.255.255.250  SSDP     215 M-SEARCH * HTTP/1.1
  31 12.967681  10.162.54.132   203.119.205.54   TCP       55 [TCP Keep-Alive] 54650 → 443 [ACK] Seq=44 Ack=72 Win=515 Len=1[Reassembl…
  32 13.002493  203.119.205.54  10.162.54.132    TCP       66 [TCP Keep-Alive ACK] 443 → 54650 [ACK] Seq=72 Ack=45 Win=126 Len=0 SLE=4…
  33 13.303491  10.162.54.132   239.255.255.250  SSDP     216 M-SEARCH * HTTP/1.1
  34 13.303492  10.162.54.132   239.255.255.250  SSDP     215 M-SEARCH * HTTP/1.1

> Frame 29: 216 bytes on wire (1728 bits), 216 bytes captured (1728 bits) on interface \Device\NPF_{A24DE49A-D22D-4000-9797-23DA5F0C48CA}, id 0
> Ethernet II, Src: IntelCor_de:dd:de (34:2e:b7:de:dd:de), Dst: IPv4mcast_7f:ff:fa (01:00:5e:7f:ff:fa)
> Internet Protocol Version 4, Src: 10.162.54.132, Dst: 239.255.255.250
v User Datagram Protocol, Src Port: 65334, Dst Port: 1900
    Source Port: 65334
    Destination Port: 1900
    Length: 182
    Checksum: 0xb7e4 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  > [Timestamps]
    UDP payload (174 bytes)
> Simple Service Discovery Protocol

0020  ff fa ff 36 07 6c 00 b6  b7 e4 4d 2d 53 45 41 52   ···6·l·· ··M-SEAR
0030  43 48 20 2a 20 48 54 54  50 2f 31 2e 31 0d 0a 48   CH * HTT P/1.1··H
0040  4f 53 54 3a 20 32 33 39  2e 32 35 35 2e 32 35 35   OST: 239 .255.255
0050  2e 32 35 30 3a 31 39 30  30 0d 0a 4d 41 4e 3a 20   .250:190 0··MAN:
0060  22 73 73 64 70 3a 64 69  73 63 6f 76 65 72 22 0d   "ssdp:di scover"·
0070  0a 4d 58 3a 20 31 0d 0a  53 54 3a 20 75 72 6e 3a   ·MX: 1·· ST: urn:
0080  64 69 61 6c 2d 6d 75 6c  74 69 73 63 72 65 65 6e   dial-mul tiscreen
0090  2d 6f 72 67 3a 73 65 72  76 69 63 65 3a 64 69 61   -org:ser vice:dia
00a0  6c 3a 31 0d 0a 55 53 45  52 2d 41 47 45 4e 54 3a   l:1··USE R-AGENT:

Details at: https://www.wireshark.org/docs/wsug_html_chunked/ChAdvChecksums.html (udp.checksum), 2 byte(s)   分组: 12987 · 已显示: 12987 (100.0%)   配置: Default
```

*WLAN
文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)
应用显示过滤器 … <Ctrl-/>
在此键入进行搜索
英
14:51
2021/11/16

The **SSDP** protocol (Application) can discover Plug & Play devices. SSDP uses unicast and multicast address (239.255.255.250). SSDP uses UDP Protocol on port **1900**.

UDP in IPv4 Packet


*WLAN
文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)
应用显示过滤器 … <Ctrl-/>

```
No.  Time       Source          Destination      Protocol Length Info
  25 11.890698  10.162.54.132   203.119.205.54   SSL        97 Continuation Data
  26 11.926276  203.119.205.54  10.162.54.132    TCP        60 443 → 54650 [ACK] Seq=1 Ack=45 Win=126 Len=0
  27 11.944370  203.119.205.54  10.162.54.132    SSL       125 Continuation Data
  28 11.998674  10.162.54.132   203.119.205.54   TCP        54 54650 → 443 [ACK] Seq=45 Ack=72 Win=515 Len=0
  29 12.300128  10.162.54.132   239.255.255.250  SSDP      216 M-SEARCH * HTTP/1.1
  30 12.300128  10.162.54.132   239.255.255.250  SSDP      215 M-SEARCH * HTTP/1.1
  31 12.967681  10.162.54.132   203.119.205.54   TCP        55 [TCP Keep-Alive] 54650 → 443 [ACK] Seq=44 Ack=72 Win=515 Len=1[Reassembl…
  32 13.002493  203.119.205.54  10.162.54.132    TCP        66 [TCP Keep-Alive ACK] 443 → 54650 [ACK] Seq=72 Ack=45 Win=126 Len=0 SLE=4…
  33 13.303491  10.162.54.132   239.255.255.250  SSDP      216 M-SEARCH * HTTP/1.1
  34 13.303492  10.162.54.132   239.255.255.250  SSDP      215 M-SEARCH * HTTP/1.1
```

```
v Internet Protocol Version 4, Src: 10.162.54.132, Dst: 239.255.255.250
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 202
    Identification: 0x56f0 (22256)
  > Flags: 0x00
    Fragment Offset: 0
    Time to Live: 1
    Protocol: UDP (17)
    Header Checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 10.162.54.132
    Destination Address: 239.255.255.250
v User Datagram Protocol, Src Port: 65334, Dst Port: 1900
    Source Port: 65334
```

```
0010  00 ca 56 f0 00 00 01 11  00 00 0a a2 36 84 ef ff   ··V·····  ····6···
0020  ff fa ff 36 07 6c 00 b6  b7 e4 4d 2d 53 45 41 52   ···6·l··  ··M-SEAR
0030  43 48 20 2a 20 48 54 54  50 2f 31 2e 31 0d 0a 48   CH * HTT P/1.1··H
0040  4f 53 54 3a 20 32 33 39  2e 32 35 35 2e 32 35 35   OST: 239 .255.255
0050  2e 32 35 30 3a 31 39 30  30 0d 0a 4d 41 4e 3a 20   .250:190 0··MAN:
```

Protocol (ip.proto), 1 byte(s) | 分组: 28498 • 已显示: 28498 (100.0%) | 配置: Default

UDP (V)

- What UDP does **not** do
  - Flow control, congestion control, or retransmission upon receipt of a bad segment.
- What UDP does **do**
  - To provide an interface to the IP protocol with the added feature of de-multiplexing multiple processes using the ports.
  - Optional end-to-end error detection (~ checksum)
- Which application uses the UDP protocol
  - **DNS** (Domain Name System, Chapter 7)
  - SSDP (Simple Service Discovery Protocol)
    - The SSDP protocol can discover Plug & Play devices.
    - SSDP is HTTP like protocol and work with NOTIFY and M-SEARCH methods.

Real-Time Transport Protocol (I)

- **RTP** (Real-time Transport Protocol)
  - RFC3550
  - It is **a transport protocol** but just happens to be implemented in the application layer.
- Two aspects of real-time transport
  - The RTP protocol for transporting audio and video data in packets
  - How the receiver plays out the audio and video at the right time?

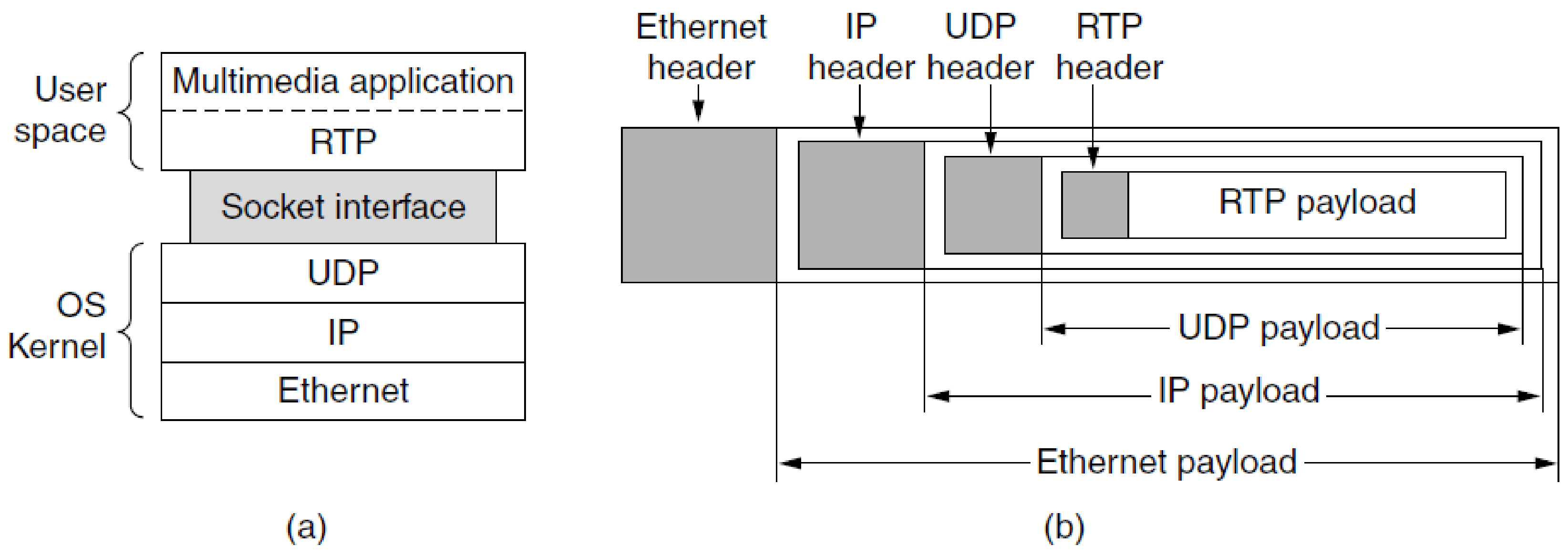


Figure 6-30. (a) The position of RTP in the protocol stack. (b) Packet nesting.

RTP (II)

- The basic function of RTP is to **multiplex** several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destination (multicasting).
- Since there is no guarantees about delivery, and packets may be lost, delayed, corrupted, etc. Each packet sent in an RTP stream is given a number one higher than its predecessor.
  - This numbering allows the destination to determine if any packets are missing.
  - If a packet is missing, either skip (a video frame) or interpolation (audio data).
  - RTP has no acknowledgements.
- Each RTP payload may contain multiple samples, and they may be coded any way that the application wants.
  - RTP provides a header field to specify the encoding scheme.
- To associate a time-stamping with the first sample in each packet. This mechanism allows the destination to do a small amount of buffering and play each sample the right number of millisecond after the start of the stream.
  - To reduce the effect of variation in network delay
  - To synchronize multiple streams.

The RTP Header (I)

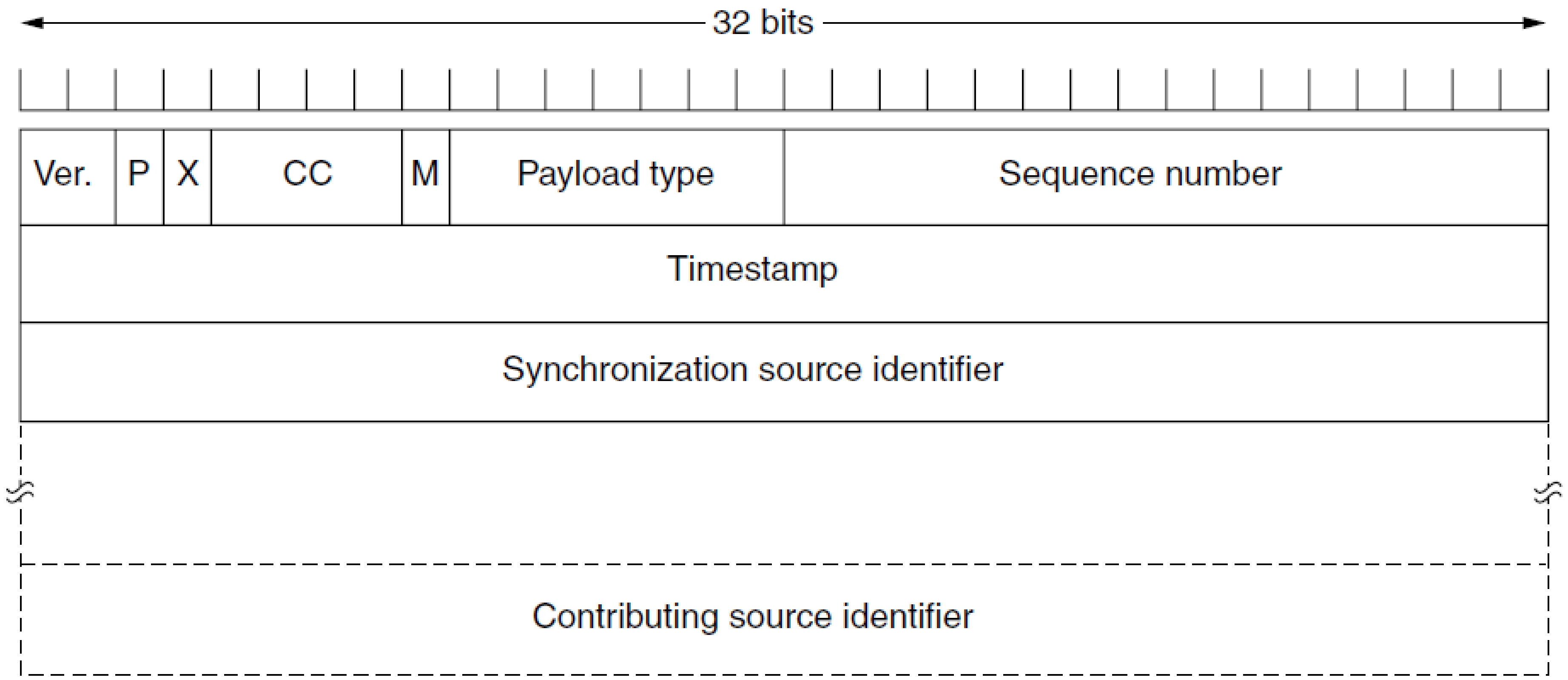

32 bits
Ver.
P
X
CC
M
Payload type
Sequence number
Timestamp
Synchronization source identifier
Contributing source identifier

Figure 6-31. The RTP header.

It consists of three 32- bit words and potentially some extensions.

The RTP Header (II)

- 1) The Version field: 2
- 2) The P bit indicates that the packet has been padded to a multiple of 4 bytes. The last padding byte tells how many bytes were added.
- 3) The X bit indicates that an extension header is present.
- 4) The CC field tells how many contributing sources are present, from 0 to 15.
- 5) The M bit field is an application-specific marker bit.
- 6) The Payload type field tells which encoding algorithm has been used.
- 7) The Sequence number is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.
- 8) The Timestamp is produced by the stream's source to note when the 1st sample in the packet was made.
- 9) The Synchronization source identifier tells which stream the packet belongs to.
- 10) The Contributing source identifier, if any, are used when mixers are present in the studio.

RTCP

- The RTCP (Real-time Transport Control Protocol) is a little sister protocol of RTP.
  - RFC3550
  - To handle feedback, synchronization, and the user interface.
  - It does not transport any media samples.

Playout with Buffering and Jitter Control

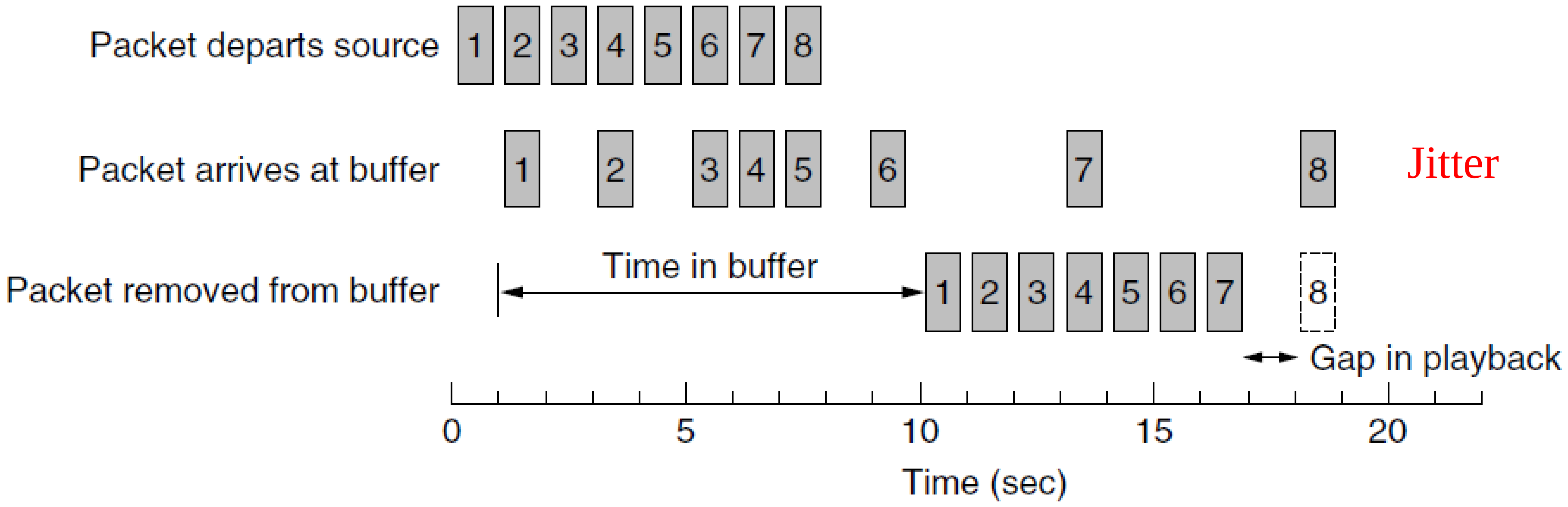


Figure 6-32. Smoothing the output stream by buffering packets.

- A key consideration for smooth playout is the **playback point**, or how long to wait at the receiver for media before playing it out. Deciding how long to wait depends on the jitter.
- To pick a good playback point, the application can measure the jitter by looking at the difference between the RTP timestamps and the arrival time, and can adapt the playback point according to the change of delay over time.

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP
    - TCP segment header
    - TCP connection establish
    - TCP sliding window
    - TCP timer management
    - TCP congestion control
    - BBR (Bottleneck Bandwidth and Round-trip propagation time)

TCP

- TCP (Transmission Control Protocol) was designed to provide a **reliable** end-to-end byte stream over an unreliable internetwork.
- An internetwork may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters in different parts.
- TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.
  - RFC793, RFC793+, RFC1122 (clarifications and bug fixes), RFC1323 (extensions for high-performance), **RFC2018** (selective acknowledgement), **RFC 2581** (congestion control), RFC2873 (repurposing of header fields for quality of service), RFC2988 (improved retransmission timers), **RFC 3168** (explicit congestion control)

The TCP Service Model

- TCP service is obtained by both the sender and the receiver creating end points, called **sockets**.
- Each socket has **a socket number (address)** consisting of **the IP addressing of the host** and a 16-bit number local to that host, called **a port**.
- Connections are identified by the socket identifiers at both ends, that is, (*socket*1, *socket*2).
- All TCP connections are full duplex and point-to-point.
- TCP does **not** support multicasting and broadcasting.

The TCP Service Model (II)

- A TCP connection is **a byte stream**, not a message stream. Message boundaries are not preserved end to end.
  - For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk.

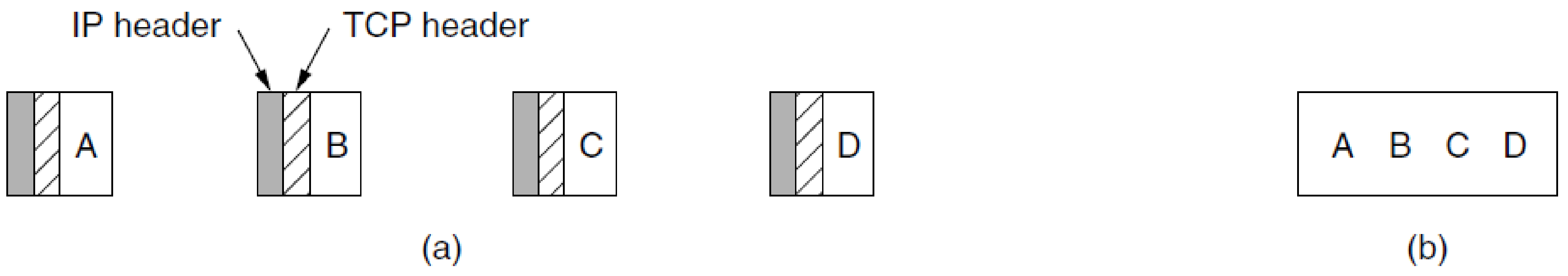


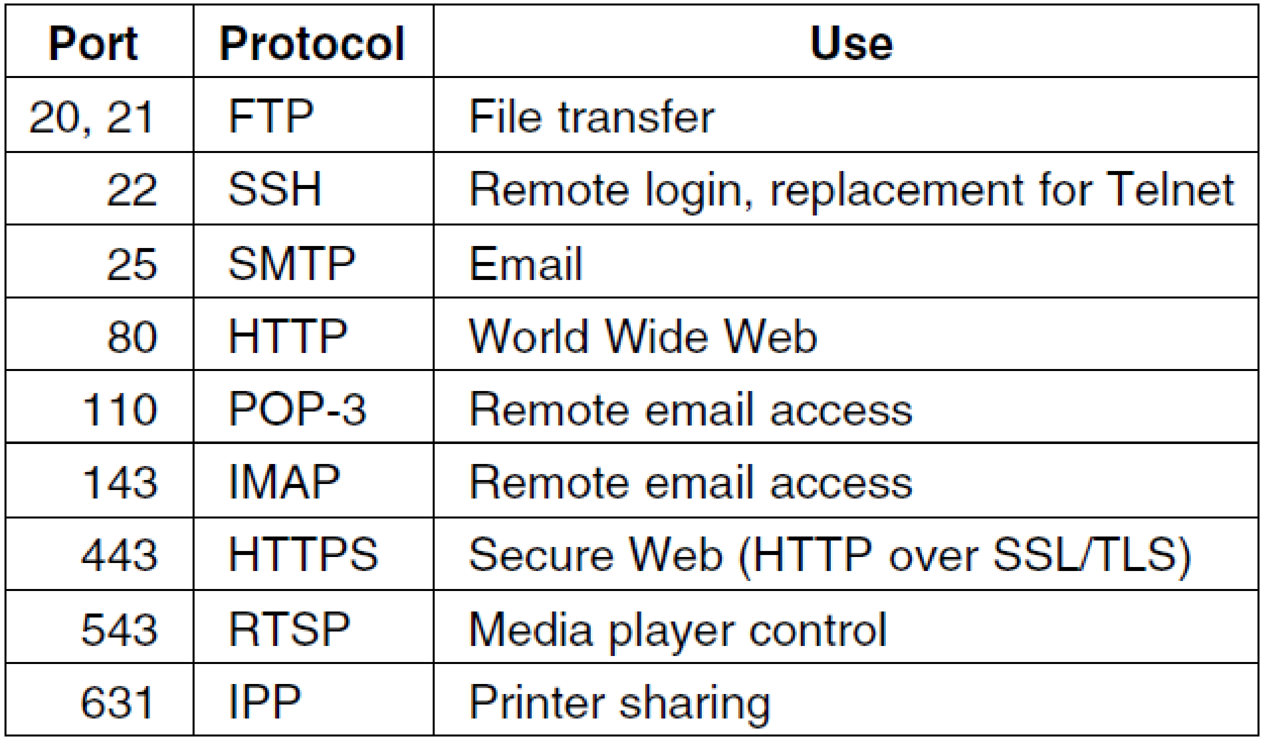
Figure 6-35. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

The TCP Service Model (III)

- When an application passes data to TCP, TCP may send it immediately or buffer it.
  - TCP may send data immediately (with the **PUSH** flag)
- When an application has high priority data that should be processed immediately, the sending application can put some control information in the data stream and give it to TCP along with the **URGENT** flag.
- When the **urgent** data are received at the destination, the receiving application is interrupted so it can stop whatever it was doing and read the data stream to find the urgent data.
- The end of the urgent data is marked so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out.

Some Assigned Ports for Well-known Applications

- The list of well-known ports is given at www.iana.org.



Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Figure 6-34. Some assigned ports.

TCP

- The TCP Protocol
  - The form of data exchange: segment
  - Include: a fixed 20-byte header + <optional> + <0-N data bytes>
  - Two limits restrict the segment size:
    - Each segment, including the TCP header, must fit in the 65515-byte IP payload (65535 – 20).
    - Each link has an MTU (Maximum Transfer Unit)

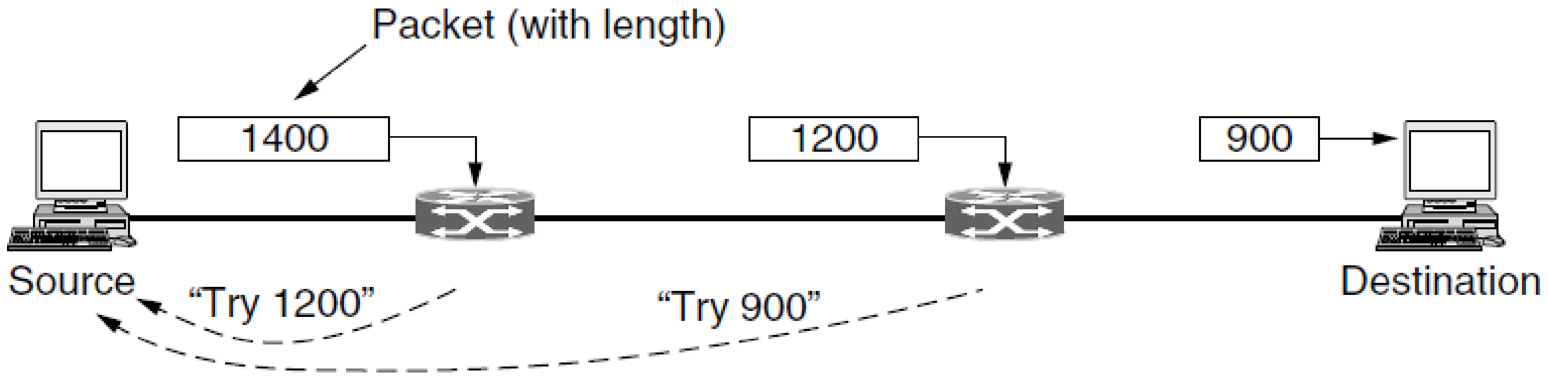


Figure 5-44. Path MTU discovery.

TCP

- The TCP Protocol
  - The basic TCP protocol: **the sliding window protocol with dynamic window size**
    - Although this protocol sounds simple, there are many problems to solve.
    - 1) Segment can arrive out of order, so bytes 3072-4095 can arrive but cannot be acknowledged because bytes 2048-3071 have not turned up yet.
    - 2) Segments can also be delayed so long in transit that the sender times out and retransmits them. The retransmissions may include different byte ranges than the original transmission, requiring careful administration to keep track of which bytes have been correctly received so far.
    - …

The TCP Segment Header

- A key feature of TCP, and one that dominants the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number.
- Every segment begins with a fixed-format, 20-byte header.
- The fixed header may be followed by header options..
- After the options, if any, up to 65535 – 20 (the IP header)– 20 (the TCP header) = 65495 data bytes may follow.
- Segments without any data are legal and are commonly used for acknowledgements and control messages.

The TCP Segment Header

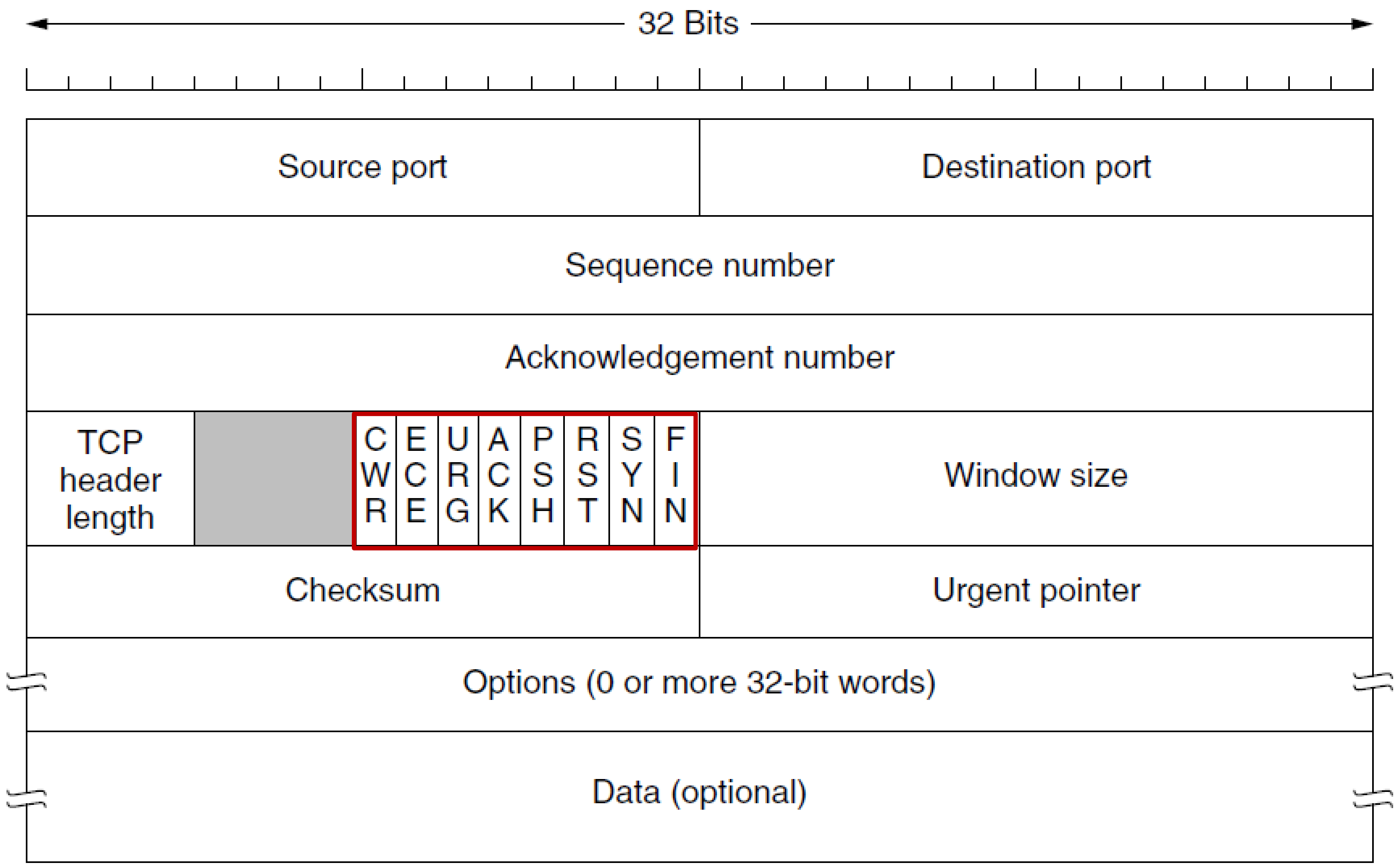


Figure 6-36. The TCP header.

The TCP Segment Header (I)

- 1) The Source port (16 bits) and Destination port (16 bits) fields identify the local end points of the connection
  - A TCP port (16 bits) plus its host’s IP address (32 bits) forms a 48-bit unique end point.
  - The connection identifier is **a 5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP, and source port, and destination IP and destination port.
- 2) The Sequence number (32 bits) and Acknowledgement number (32 bits) fields
  - The Acknowledgement number specifies the next in-order byte expected, not the last byte correctly received.
  - It is **a cumulative acknowledgement** because it summarizes the received data with a single number.

The TCP Segment Header (II)

- 3) The <u>TCP header length</u> (4 bits): tells how many 32-bit words are contained in the TCP header.
  - This information is needed because the Options field is of variable length, so the header is, too.
  - Technically, this field really indicates the start of the data within the segment, measured in 32-bit words.
- 4) The not used 4-bit field

The TCP Segment Header (III)

- 5) eight 1-bit fields
  - **CWR** and **ECE** are used to signal congestion when **ECN** (Explicit Congestion Notification) is used. [RFC 3168]
    - **ECE** is set to signal an ECN-Echo to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network.
    - **CWR** is set to signal *Congestion Window Reduced* from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the ECN-Echo.
  - **URG** is set to 1 if the Urgent pointer is in use. The Urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
  - The **ACK** bit is set to 1 to indicate that the Acknowledgement number is valid. *If ACK is 0, the segment does not contain an acknowledgement, so the Acknowledgement number field is ignored.*

The TCP Segment Header (IV)

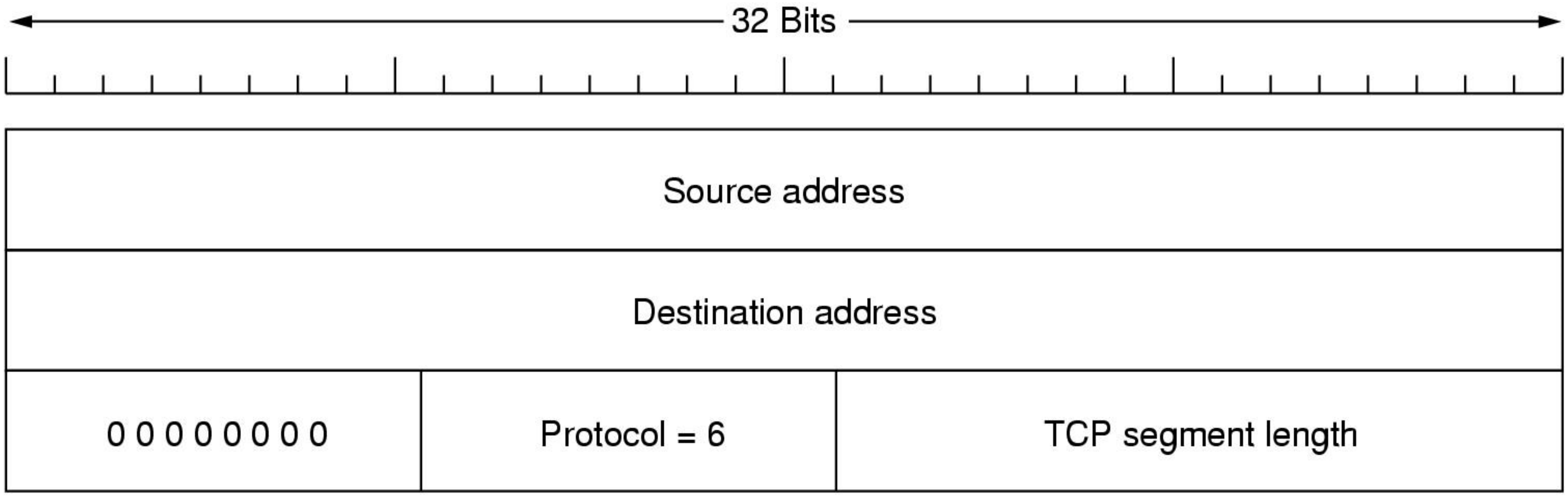
- 5) eight 1-bit fields
  - The **PSH** bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer.
  - The **RST** bit is used to abruptly reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection.
  - The **SYN** bit is used to establish connections.
    - The connection request has **SYN** = 1 and **ACK**= 0.
    - **SYN** = 1 and **ACK** = 1: the connection reply does bear an acknowledgement.
    - In essence, the **SYN** bit is used to denote both CONNECTION REQUEST and CONNECTION ACCEPTED, with the **ACK** bit used to distinguish between those two possibilities.

The TCP Segment Header (V)

- 5) eight 1-bit fields
  - The **FIN** bit is used to release a connection. It specifies that the sender has no more data to transmit. However, after closing a connection, the closing process may continue to receive data indefinitely. Both SYN and FIN segments have sequence numbers and are thus guaranteed to be processed in the corrected order.
- 6) The Window size field (16 bits) tells how many bytes may be sent starting at the byte acknowledged.
  - A window size field of 0 is legal and says that the bytes up to and including Acknowledgement number – 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment.
  - In TCP, acknowledgement and permission to send additional data are completely decoupled.

The TCP Segment Header (VI)

- 7) Checksum (16 bits) provides extra reliability. It checksum the header, the data and a conceptual pseudo-header in exactly the same way as UDP, except that the pseudo-header has the protocol number for TCP (6) and the checksum is mandatory.



The pseudoheader of TCP

The TCP Segment Header (VII)

- 8) The Options field: the options are of variable length, fill a multiple of 32 bits by using padding with zeros, and may extended to 40 bytes to accommodate the longest TCP header that can be specified.
  - A widely used option is the one that allows each host to specify the **MSS** (**Maximum Segment Size**) it is willing to accept.
    - If a host does not use this option, it defaults to a 536-byte load. All Internet hosts are required to accept TCP segments of 536+20 = 556 bytes.
  - The **window scale** option allows the sender and receiver to negotiate a window scale factor at the start of a connection.
    - This option is especially useful for lines with high bandwidth, high delay, or both. Large window size would allow the sender to keep pumping data out.
  - The **timestamp** option carries a timestamp sent by the sender and echoed by the receiver.
    - It is used to compute round-trip time samples that are used to estimate when a packet has been lost.
    - It is also used as a logical extension of the 32-bit sequence number.
  - The **SACK** (**Selective ACKnowledgement**) option
    - It supplements the Acknowledgement number and is used after a packet has been lost but subsequent (or duplicate) data has arrived.

Outline

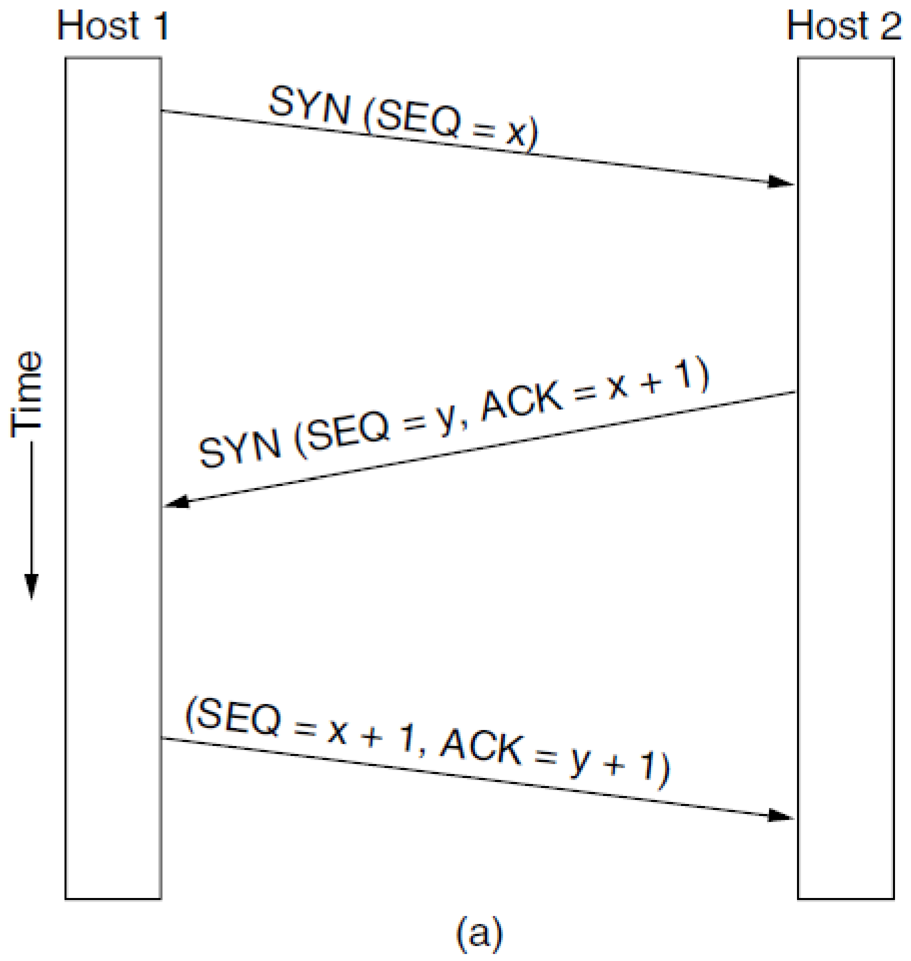
- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP
    - TCP segment header
    - TCP connection establish
    - TCP sliding window
    - TCP timer management
    - TCP congestion control
    - BBR (Bottleneck Bandwidth and Round-trip propagation time)

TCP Connection Establishment: Three Way Handshake

- Connections are established in TCP by means of the three-way handshake.



1) Note that a SYN segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.
2) The initial sequence number chosen by each host should **cycle slowly**. This rule is to protect against **delayed duplicated packets**.

TCP Connection Establishment (I)

- Step 1: **SYN** — for establishing a connection
  - Client sends a request segment to the server
  - Request segment consists only of TCP Header with an empty payload.
    - Maybe?
  - Then, it waits for a reply segment from the server.

- **Request segment** contains the following information in TCP header:
  - 1. Initial sequence number (randomly chosen by the client)
  - 2. SYN bit set to 1. (to indicate the server that this segment contains the initial sequence number used by the client)
  - 3. Maximum segment size (the largest data chunk that client can send and receiver from the server, contained in the Options field)
  - 4. Receiving window size (the limit of unacknowledged data that can be sent to the client, contained in the window size field)

TCP Connection Establishment (II)

- Step 2: **SYN + ACK** — After receiving the request segment
  - Server responds to the client by sending the reply segment.
  - It informs the client of the parameters at the server side.

- **Reply segment** contains the following information in TCP header:
  - 1. Initial sequence number (randomly chosen by the server)
  - 2. SYN bit set to 1. (to indicate the client that this segment contains the initial sequence number used by the server)
  - 3. Maximum segment size (the largest data chunk that server can send and receive from the client, contained in the Options field)
  - 4. Receiving window size (the limit of unacknowledged data that can be sent to the server, contained in the window size field)
  - 5. Acknowledgement number (the initial sequence number in the request segment sent by the client incremented by 1 as an acknowledgement number, or it indicates the sequence number of the next data byte that server expects to receive from the client)
  - 6. ACK bit set to 1. (to indicate the client that the acknowledgment number field in the current segment is valid)

TCP Connection Establishment (III)

- Step 3: **ACK** — After receiving the reply segment
  - Client acknowledges the response of server.
  - It acknowledges the server by sending a pure acknowledgement.
    - Not necessary.

TCP Connection Establishment: Important Points (I)

- Connection establishment phase consume 1 sequence number of both sides.
  - Request segment consumes 1 sequence number of the requester.
  - Reply segment consumes 1 sequence number of the responder.
  - Pure acknowledgement do not consume any sequence number.

- Pure acknowledgement for the reply segment is <u>not necessary</u>. This is because
  - If client sends the data packet immediately, then it will be considered as an acknowledgement.
  - It means that in the first two steps only, the full duplex connection is established.

TCP Connection Establishment: Important Points (II)

- For all the segments *except* the request segment, ACK bit is always set to 1. This is because
  - For the request segment, acknowledgement number field will always be **invalid**.
  - For all other segments, acknowledgement number field will always be **valid**.

- Certain parameters are negotiated during connection establishment. The negotiation can be on setting the values of the following parameters
  - 1. Window size
  - 2. Maximum segment size
  - 3. Timer values

TCP Connection Establishment: Important Points (III)

- In any TCP segment,
  - If SYN bit = 1 and ACK bit = 0, then it must be **the request segment**.
  - If SYN bit = 1 and ACK bit = 1, then it must be **the reply segment**.
  - If SYN bit = 0 and ACK bit = 1, then it can be the pure ACK or segment meant for data transfer.
  - If SYN bit = 0 and ACK bit = 0, then this combination is not possible.

The IPv4 Datagram

- The header has a 20-byte fixed part and a variable-length optional part.
- The bits are transmitted from left to right and top to bottom. This is "big-endian" network byte order.

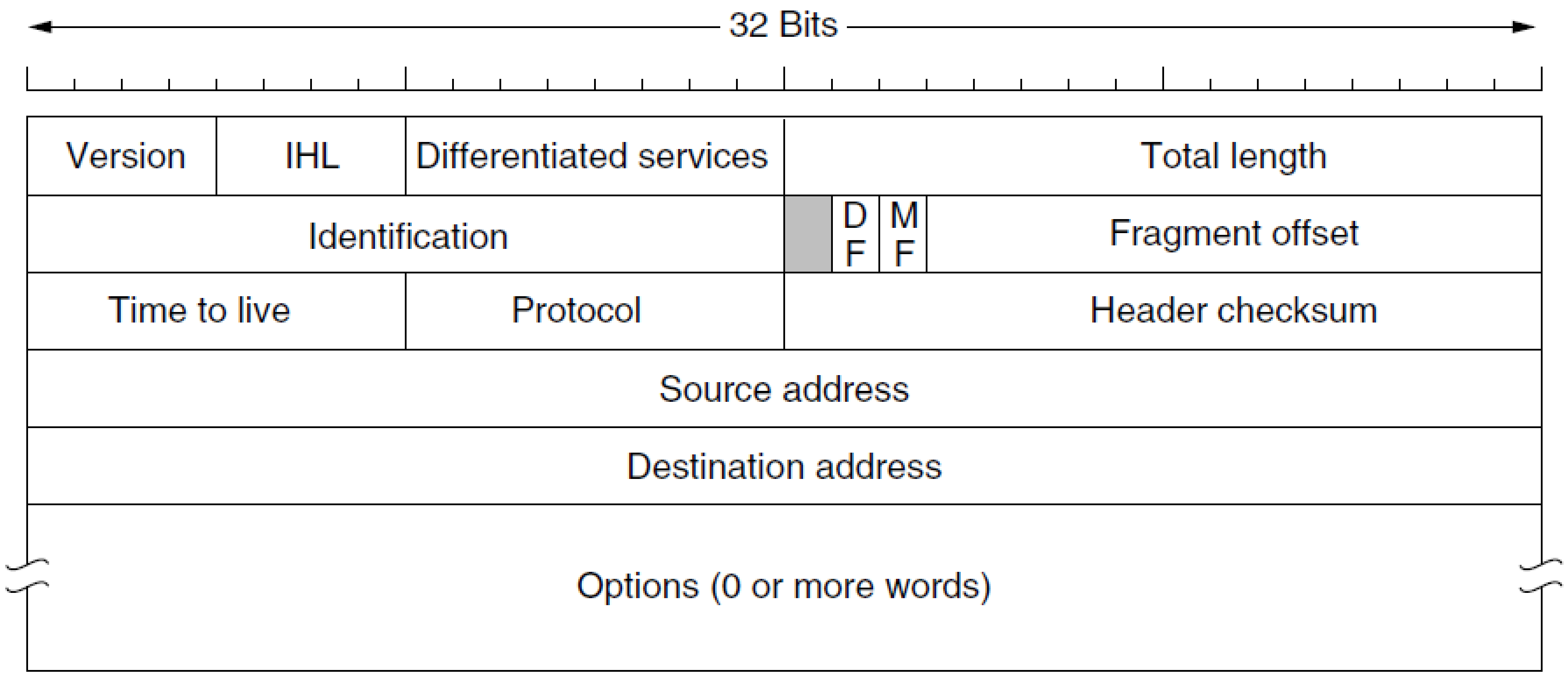


Figure 5-46. The IPv4 (Internet Protocol) header.

The TCP Segment Header

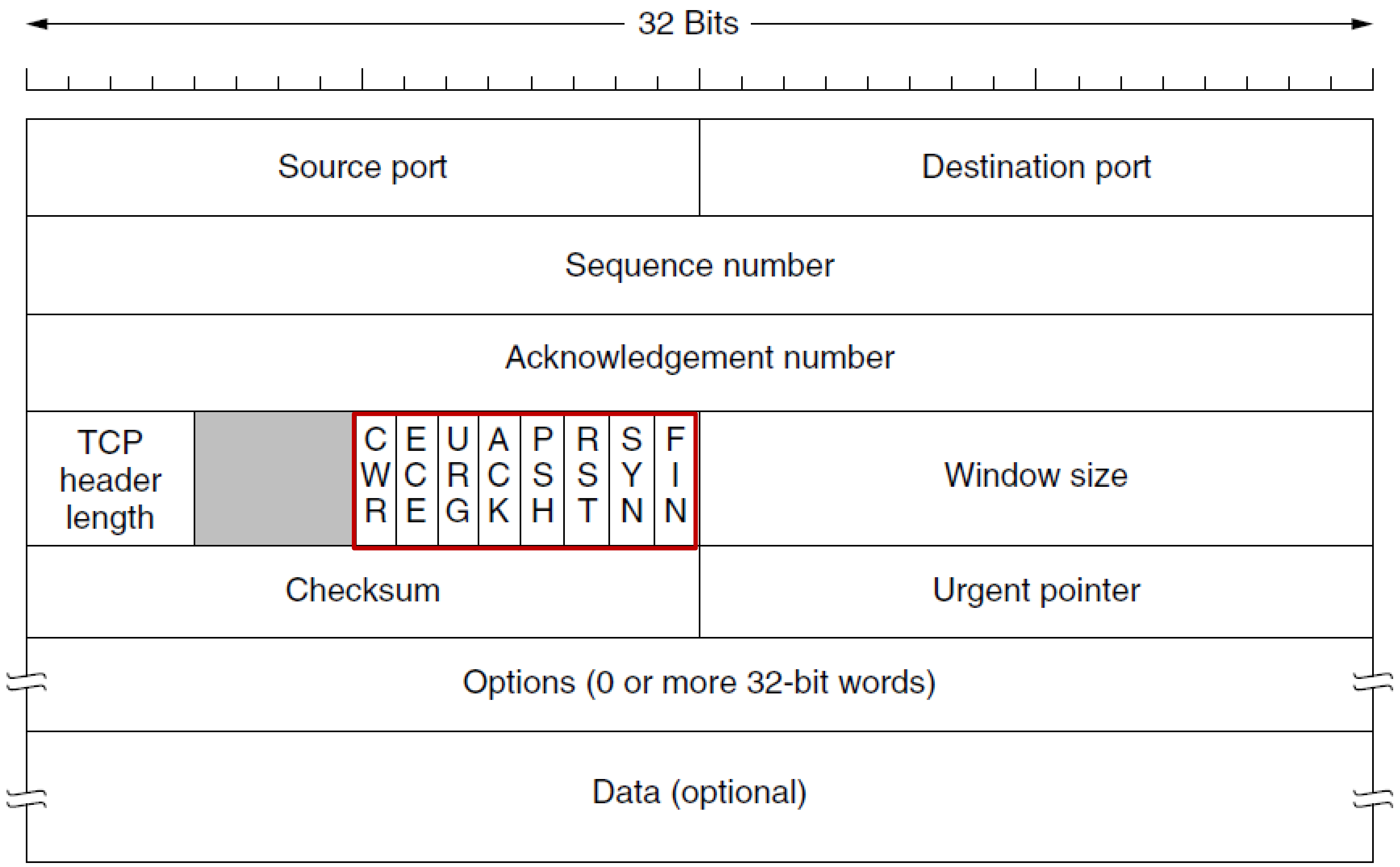
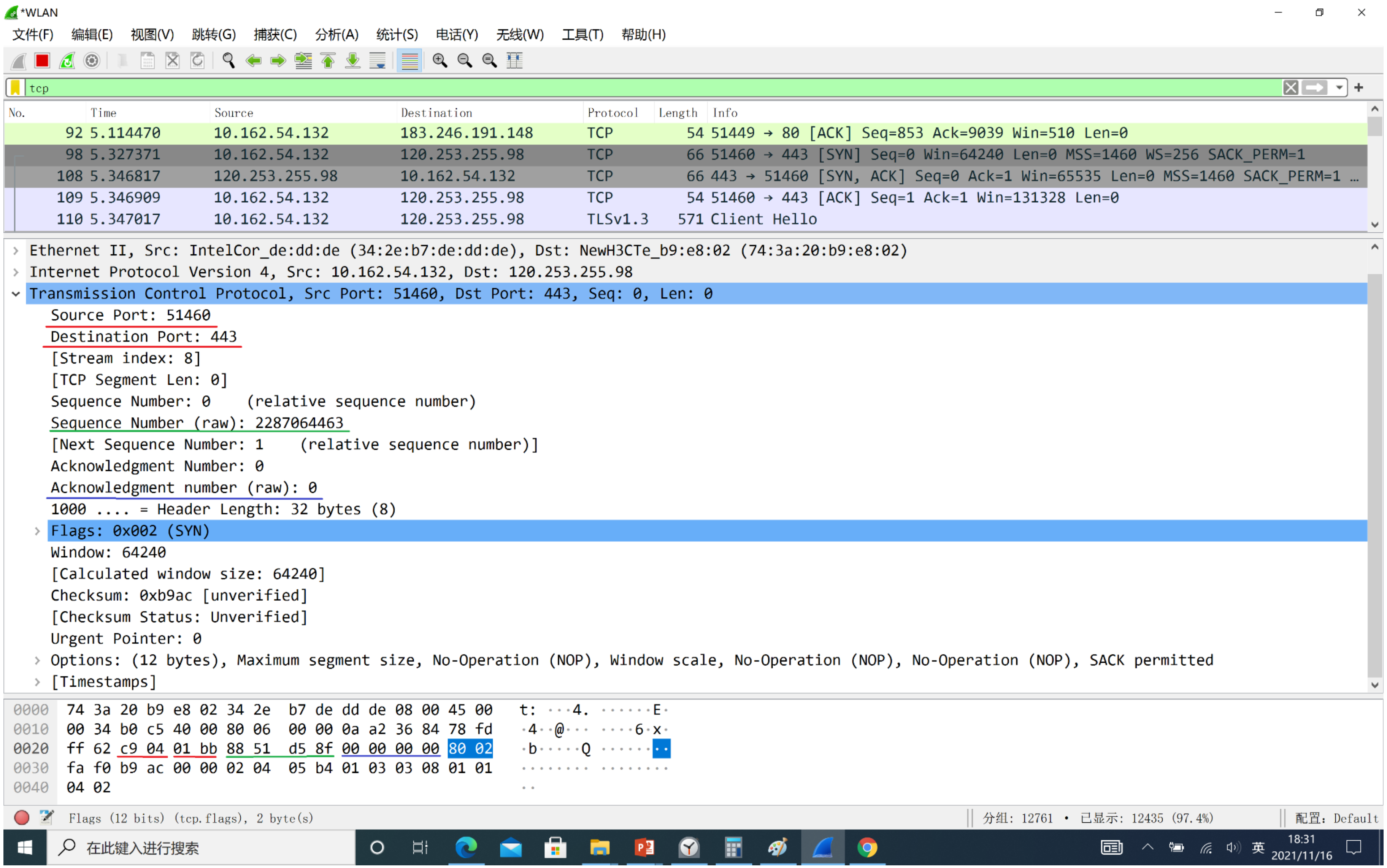


Figure 6-36. The TCP header.

A TCP-SYN Example



```
No.   Time       Source          Destination      Protocol  Length  Info
  92 5.114470   10.162.54.132   183.246.191.148  TCP        54 51449 → 80 [ACK] Seq=853 Ack=9039 Win=510 Len=0
  98 5.327371   10.162.54.132   120.253.255.98   TCP        66 51460 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
 108 5.346817   120.253.255.98  10.162.54.132    TCP        66 443 → 51460 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 …
 109 5.346909   10.162.54.132   120.253.255.98   TCP        54 51460 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0
 110 5.347017   10.162.54.132   120.253.255.98   TLSv1.3   571 Client Hello
```

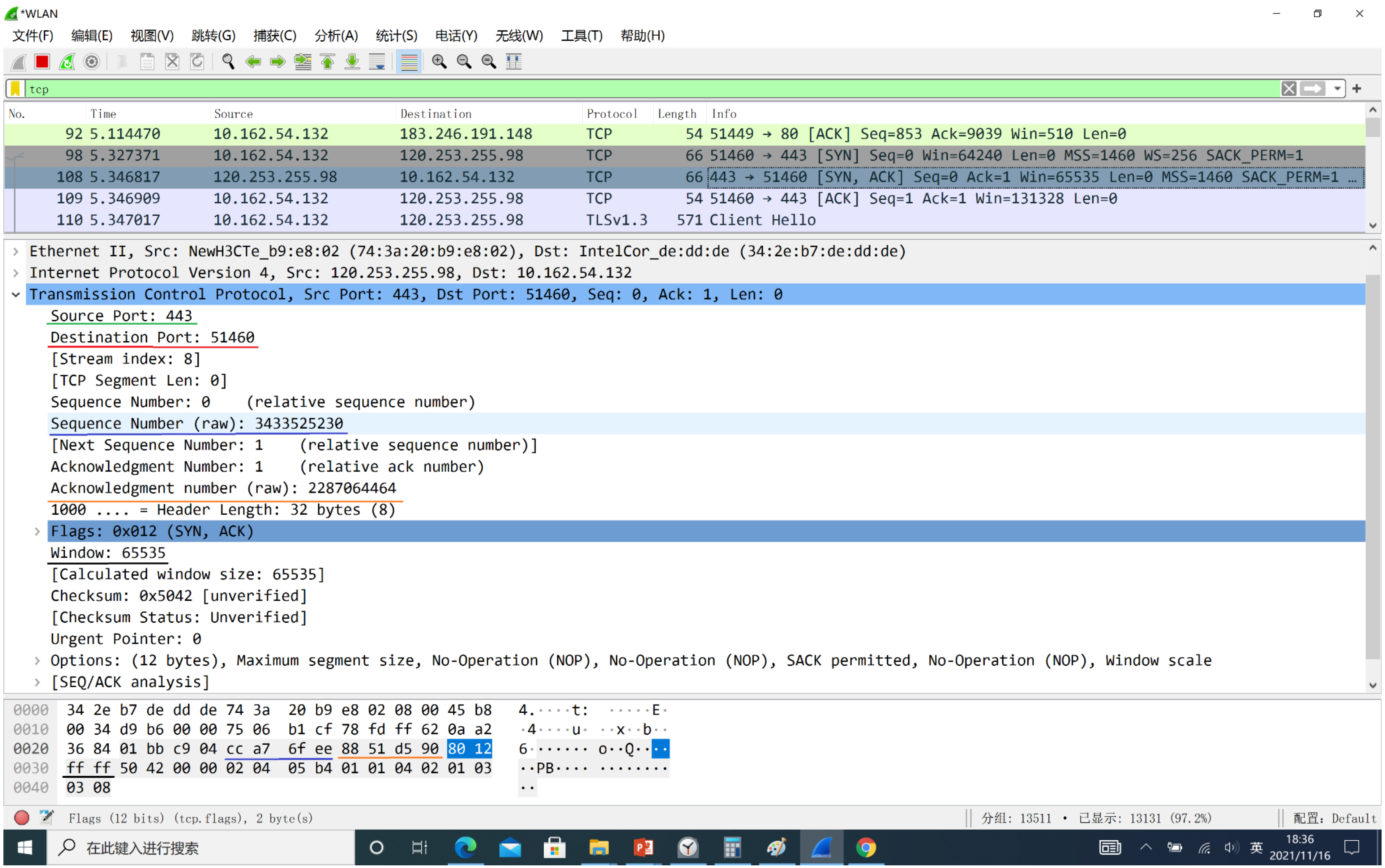
```
> Ethernet II, Src: IntelCor_de:dd:de (34:2e:b7:de:dd:de), Dst: NewH3CTe_b9:e8:02 (74:3a:20:b9:e8:02)
> Internet Protocol Version 4, Src: 10.162.54.132, Dst: 120.253.255.98
v Transmission Control Protocol, Src Port: 51460, Dst Port: 443, Seq: 0, Len: 0
    Source Port: 51460
    Destination Port: 443
    [Stream index: 8]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 2287064463
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x002 (SYN)
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0xb9ac [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), SACK permitted
  > [Timestamps]
```

```
0000  74 3a 20 b9 e8 02 34 2e  b7 de dd de 08 00 45 00   t: ···4. ······E·
0010  00 34 b0 c5 40 00 80 06  00 00 0a a2 36 84 78 fd   ·4··@··· ····6·x·
0020  ff 62 c9 04 01 bb 88 51  d5 8f 00 00 00 00 80 02   ·b·····Q ········
0030  fa f0 b9 ac 00 00 02 04  05 b4 01 03 03 08 01 01   ········ ········
0040  04 02                                              ··
```

Flags (12 bits) (tcp.flags), 2 byte(s)   分组: 12761 • 已显示: 12435 (97.4%)   配置: Default

连接发起端选了个Seq no. 为：2287064463，这个segment中头部字节为32个，不是TCP头部固定的20个字节，说明有Option选项，共占12个字节。

A TCP-SYN-ACK Example


*WLAN
文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)
tcp
在此键入进行搜索
英
18:36
2021/11/16

```
No.    Time        Source            Destination       Protocol  Length  Info
   92  5.114470    10.162.54.132     183.246.191.148   TCP       54  51449 → 80 [ACK] Seq=853 Ack=9039 Win=510 Len=0
   98  5.327371    10.162.54.132     120.253.255.98    TCP       66  51460 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
  108  5.346817    120.253.255.98    10.162.54.132     TCP       66  443 → 51460 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 …
  109  5.346909    10.162.54.132     120.253.255.98    TCP       54  51460 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0
  110  5.347017    10.162.54.132     120.253.255.98    TLSv1.3  571  Client Hello
```

```
> Ethernet II, Src: NewH3CTe_b9:e8:02 (74:3a:20:b9:e8:02), Dst: IntelCor_de:dd:de (34:2e:b7:de:dd:de)
> Internet Protocol Version 4, Src: 120.253.255.98, Dst: 10.162.54.132
v Transmission Control Protocol, Src Port: 443, Dst Port: 51460, Seq: 0, Ack: 1, Len: 0
    Source Port: 443
    Destination Port: 51460
    [Stream index: 8]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 3433525230
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2287064464
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x012 (SYN, ACK)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0x5042 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted, No-Operation (NOP), Window scale
  > [SEQ/ACK analysis]
```

```
0000  34 2e b7 de dd de 74 3a  20 b9 e8 02 08 00 45 b8   4.····t:  ·····E·
0010  00 34 d9 b6 00 00 75 06  b1 cf 78 fd ff 62 0a a2   ·4····u·  ··x··b··
0020  36 84 01 bb c9 04 cc a7  6f ee 88 51 d5 90 80 12   6·······  o··Q····
0030  ff ff 50 42 00 00 02 04  05 b4 01 01 04 02 01 03   ··PB····  ········
0040  03 08                                              ··
```

```
Flags (12 bits) (tcp.flags), 2 byte(s)    分组: 13511 · 已显示: 13131 (97.2%)    配置: Default
```

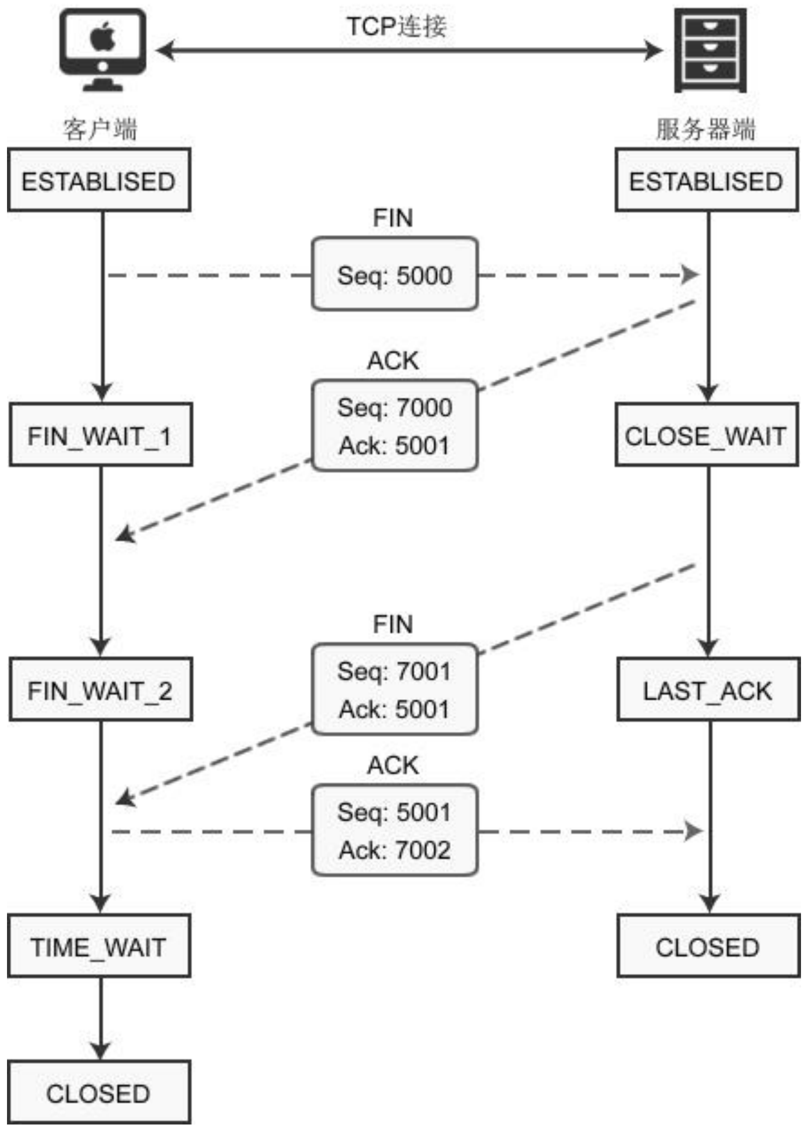
注意到没：这里ACK no (2287064464) 刚好前面SYN segment中Sequence no + 1. 接收端选择的Seq no. 为3433525230， 和发送端的Seq no.没有任何关系！

“SYN Flood” Attack

- A vulnerability with implementing the three-way handshake is that the listening process must remember its sequence number as soon it responds with its own SYN segment.
  - The clock-based ISN proved to have a significant weakness:
  - **A SYN flood** — A malicious sender can tie up resources on a host by sending a stream of SYN segments and never following through to complete the connection. It crippled many Web servers in the 1990s.
  - One way to defend against this attack is to use **SYN cookies**. Instead of remembering the sequence number, a host chooses a cryptographically generated sequence number, puts it on the outgoing segment, and forgets it. If the three-way handshake completes, this sequence number (+ 1) will be returned to the host. It can then regenerate the correct sequence number by running the same cryptographic function, as long as the inputs to that function are known, for example, the other host’s IP address and port, and a local secret.
    - ISN = C(t) + hash(local_addr, local_port, remote_addr, remote_port, key) (Initial Sequence Number)
  - RFC1948

TCP Connection Release

- Each simplex connection is released independently.
- Normally, **four TCP segments** are needed to release a connection: one FIN and one ACK for each direction.
- To avoid the two-army problem (discussed in Sec.6.2.3), *timers are used.*
  - If a response to a FIN is not forthcoming with **two maximum packet lifetimes**, the sender of the FIN releases the connection. The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well.



The Two-army Problem

- Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.
- The two-army problem

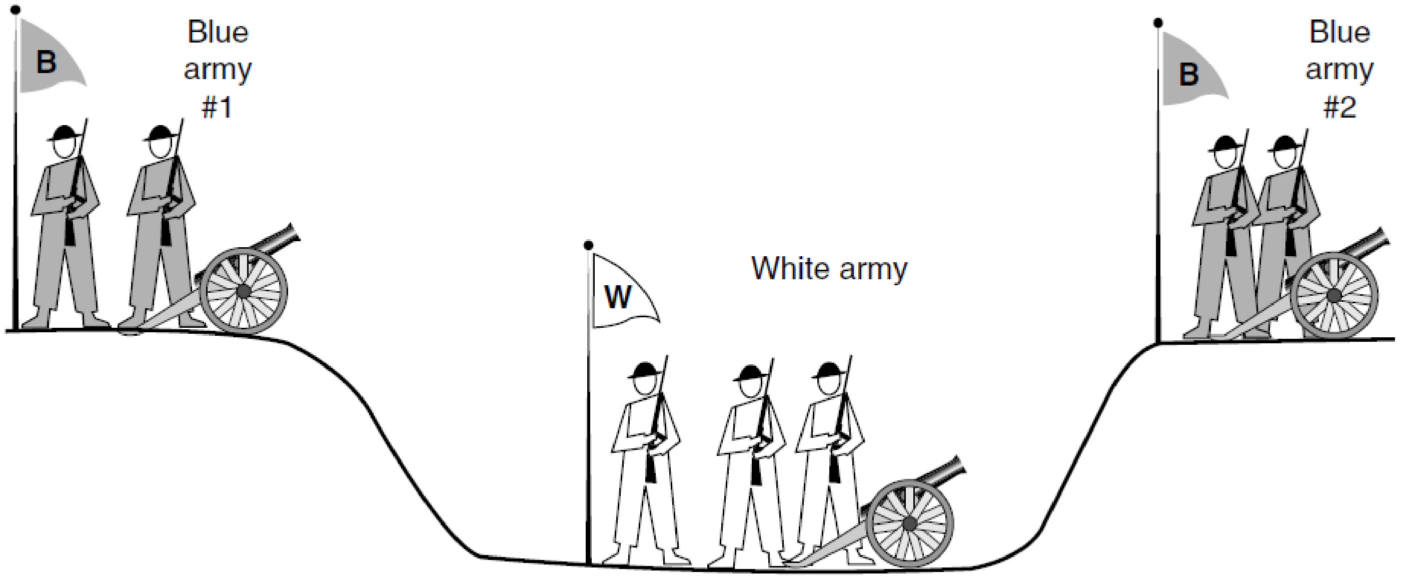
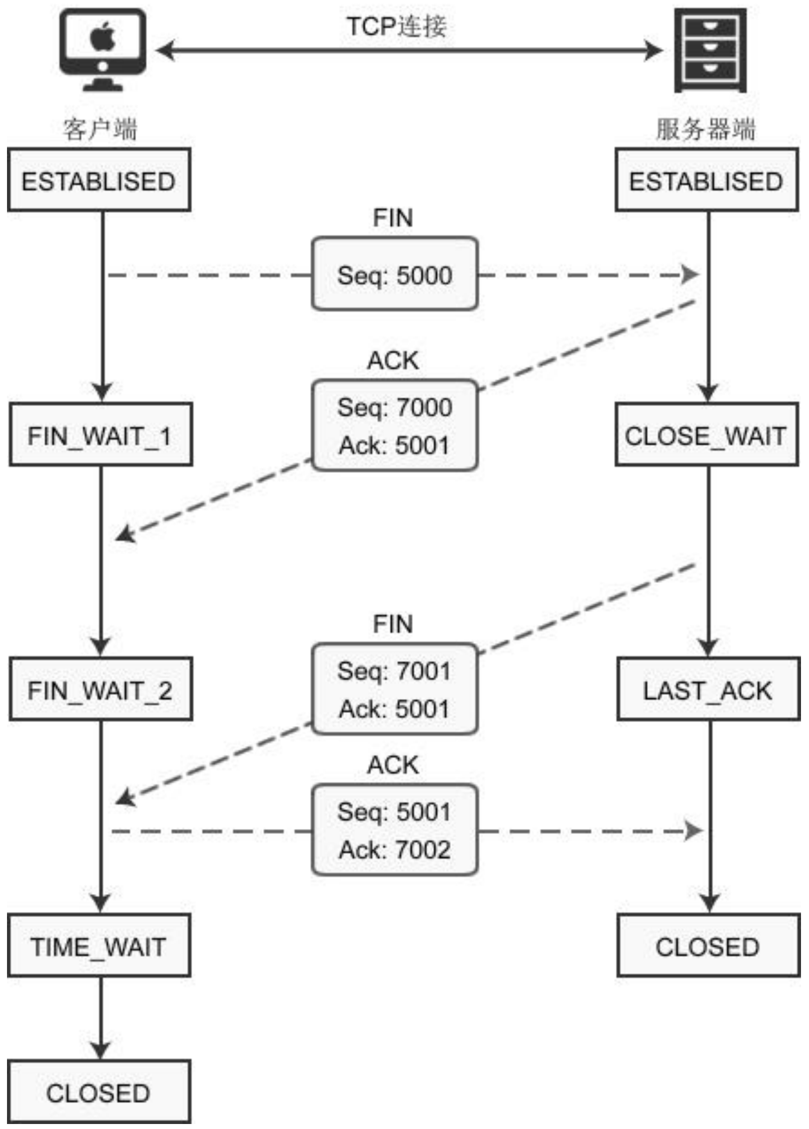


Figure 6-13. The two-army problem.

TCP Connection Release

- Each simplex connection is released independently.
- Normally, **four TCP segments** are needed to release a connection: one FIN and one ACK for each direction.
- To avoid the two-army problem (discussed in Sec.6.2.3), *timers are used.*
  - If a response to a FIN is not forthcoming with **two maximum packet lifetimes**, the sender of the FIN releases the connection. The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well.



TCP Connection Release (I)

- Consider there is a well established TCP connection between the client and server. Client wants to terminate the connection
- The following steps are followed in terminating the connection:
  - Step 1: For terminating the connection
    - Client sends a FIN segment to the server with FIN bit set to 1.
    - Client enters the FIN_WAIT_1 state.
    - Client waits for an acknowledgement from the server.
  - Step 2: after receiving the FIN segment
    - Server frees up its buffers (receiving buffer)
    - Server sends an acknowledgement to the client.
    - Server enters the CLOSE_WAIT state.

TCP Connection Release (II)

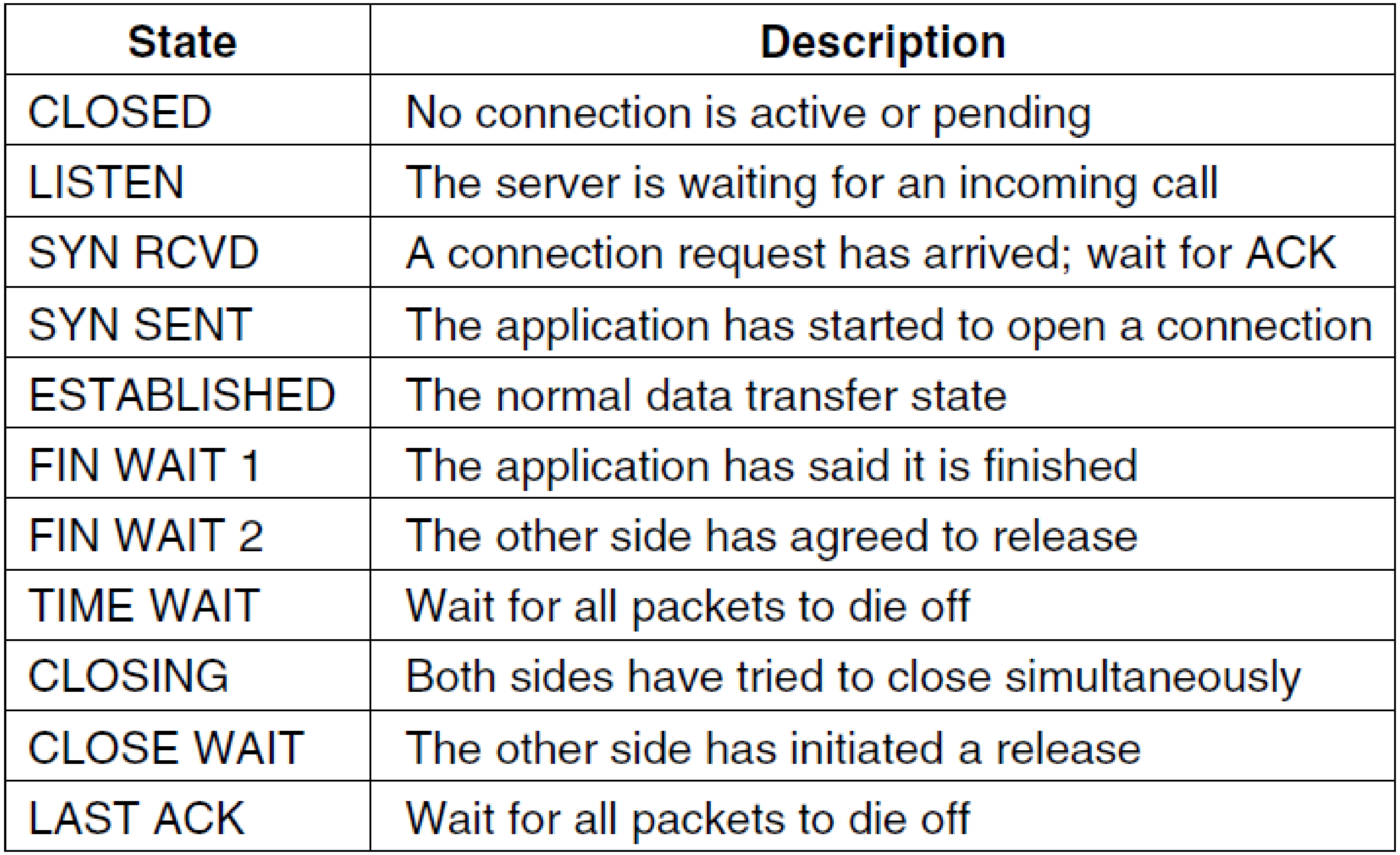
- The following steps are followed in terminating the connection:
  - Step 3: After receiving the acknowledgement, client enters the FIN_WAIT_2 state. Now,
    - The connection from client to server is terminated i.e. one way connection is closed.
    - Client cannot send any data to the server since server has released its buffers.
    - Pure acknowledgements can still be sent from client to server (no data).
    - The connection from server to client is still open i.e. one way connection is still open.
    - Server can send both data and acknowledgements to the client.
  - Step 4: Now suppose server wants to close the connection with the client. For terminating the connection,
    - Server sends a FIN segment to the client with FIN bit set to 1.
    - Server waits for an acknowledgement from the client.

TCP Connection Release (III)

- The following steps are followed in terminating the connection:
  - Step 5: After receiving the FIN segment,
    - Client frees up its buffers (receiving buffer).
    - Client sends an acknowledgement to the server (not mandatory).
    - Client enters the TIME_WAIT state.

- TIME_WAIT state
  - The TIME_WAIT state allows the client to resend the final acknowledgement if it gets lost.
  - After the wait, the connection gets formally closed.

TCP Connection Management Modeling



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 application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Figure 6-38. The states used in the TCP connection management finite state machine.

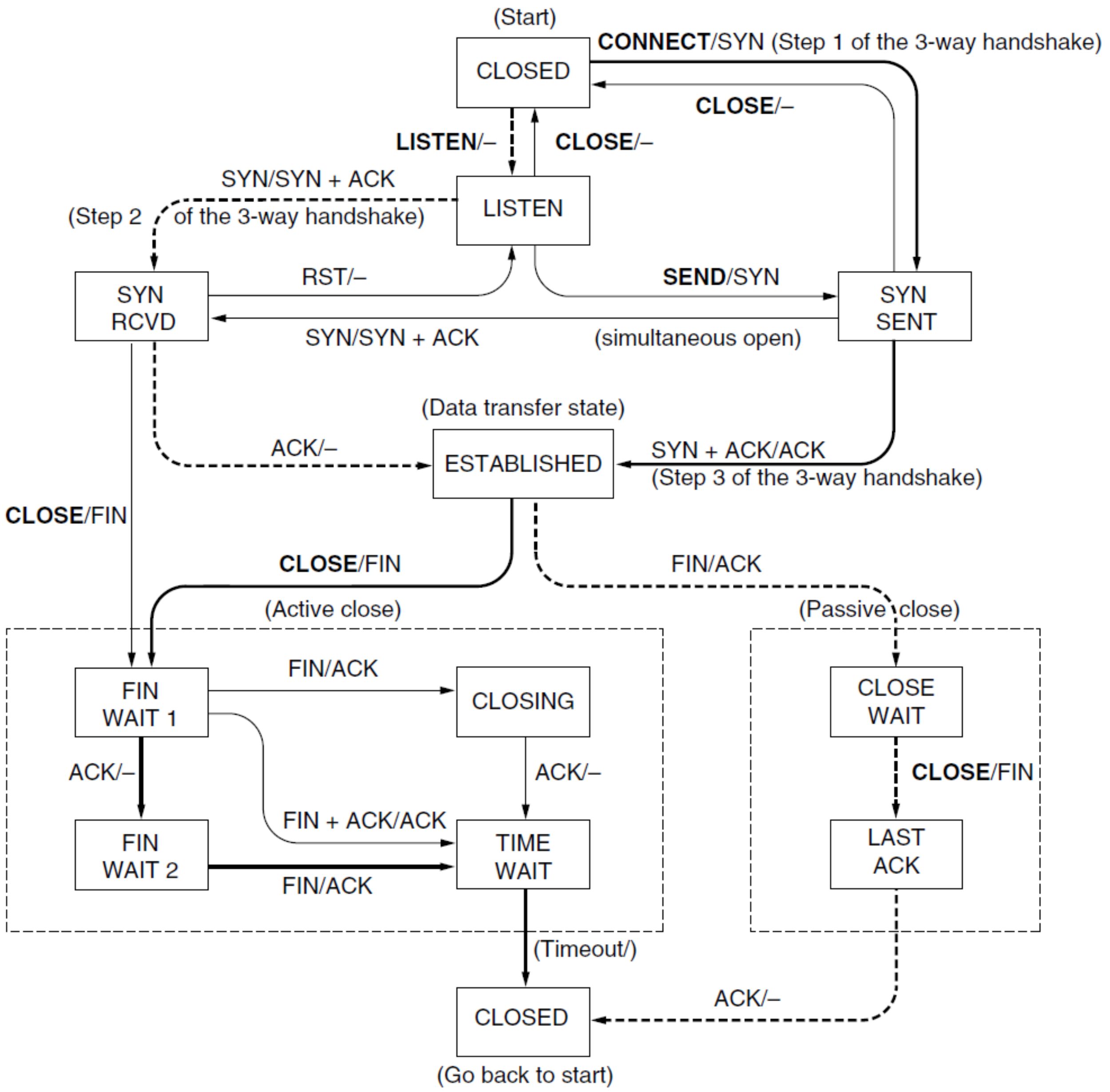


Figure 6-39. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

Each line is marked by an event/action pair

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP
    - TCP segment header
    - TCP connection establish
    - TCP sliding window
    - TCP timer management
    - TCP congestion control
    - BBR (Bottleneck Bandwidth and Round-trip propagation time)

TCP Sliding Window

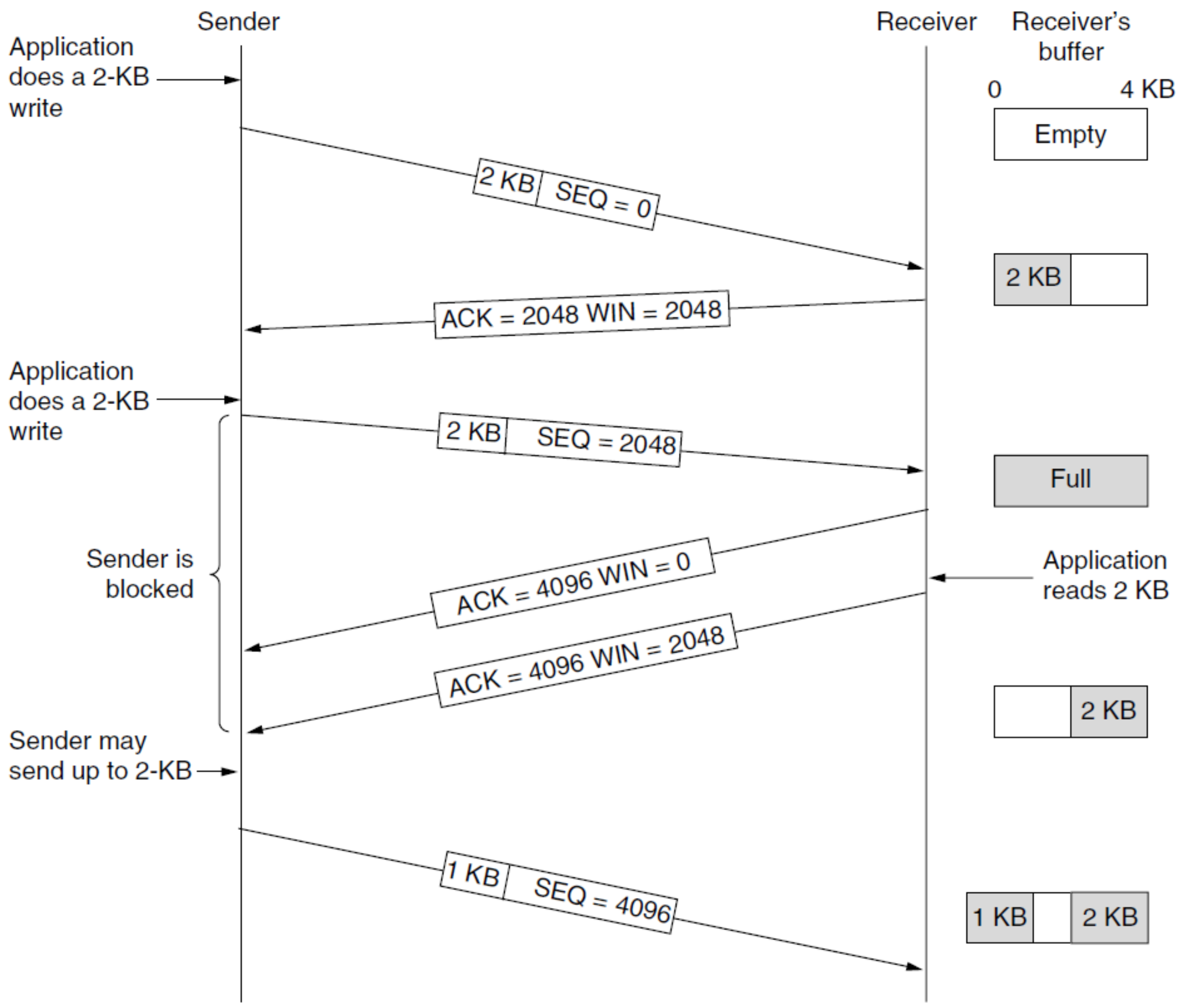
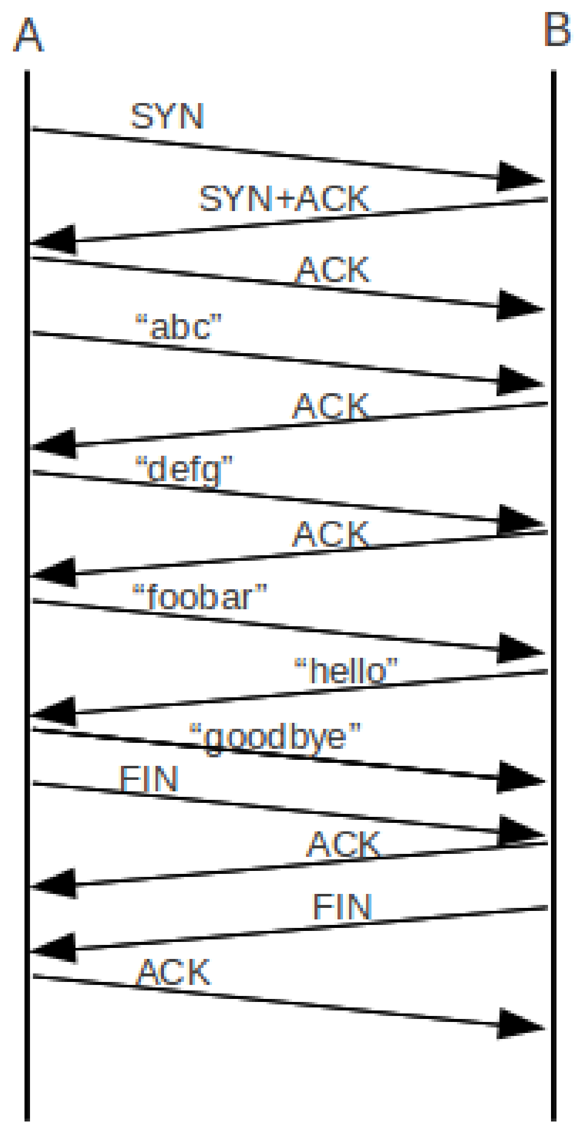


Figure 6-40. Window management in TCP.

ACK = 2048指下个期待的字节号 (It indicates the sequence number of the next data byte that receiver **expects** to receive from the sender)

An connection for data transmission example (I)



In the following table, **relative sequence numbers** are used, which is to say that sequence numbers begin with 0 on each side. The SEQ numbers on the A side correspond to the ACK numbers on the B side; they both count data flowing from A to B.

An Ladder Example (II)

	A sends	B sends
1	SYN: **SEQ = 0**	
2		SYN+ACK: SEQ = 0, **ACK = 1** (expecting)
3	ACK: **SEQ = 1**， ACK = 1 (ACK of SYN)	
4	“abc”: **SEQ = 1**, ACK = 1	
5		ACK of “abc”: SEQ = 1, **ACK = 4** (no data)
6	“dfeg”: **SEQ = 4**, ACK = 1	
7		ACK of “dfeg”: SEQ = 1, **ACK = 8** (no data)
8	“foobar”: **SEQ = 8**, ACK = 1	
9		“hello”: SEQ = 1, **ACK = 14**
10	“goodbye”: **SEQ = 14**, ACK = 6	
11	FIN: **SEQ = 21**, ACK = 6	ACK of “goodbye”: SEQ = 6, **ACK = 21**
12		ACK of FIN: SEQ = 6, **ACK = 22**
13		FIN: SEQ = 6, **ACK = 22**
14	ACK of FIN: **SEQ = 22**, ACK = 7	

Another Example

- Suppose A and B create a TCP connection with ISN_A = 20,000 and ISN_B = 5,000. A sends three 1000-byte packets (Data1, Data2, and Data 3 below), and B ACKs each. Then B sends a 1000-byte packet DataB to A and terminates the connection with a FIN. In the table below, fill in the SEQ and ACK fields for each packet shown.

	A sends	B sends
1	SYN: ISN_A = 20000	
2		SYN+ACK: ISN_B = 5000, ACK =
3	ACK: SEQ = , ACK =	
4	Data1: SEQ = , ACK =	
5		ACK: SEQ = , ACK = (no data)
6	Data2: SEQ = , ACK =	
7		ACK: SEQ = , ACK = (no data)
8	Data3: SEQ = , ACK =	
9		ACK: SEQ = , ACK = (no data)
10		DataB: SEQ = , ACK =
11	ACK: SEQ = , ACK =	
12		FIN: SEQ = , ACK =

TCP Sliding Window

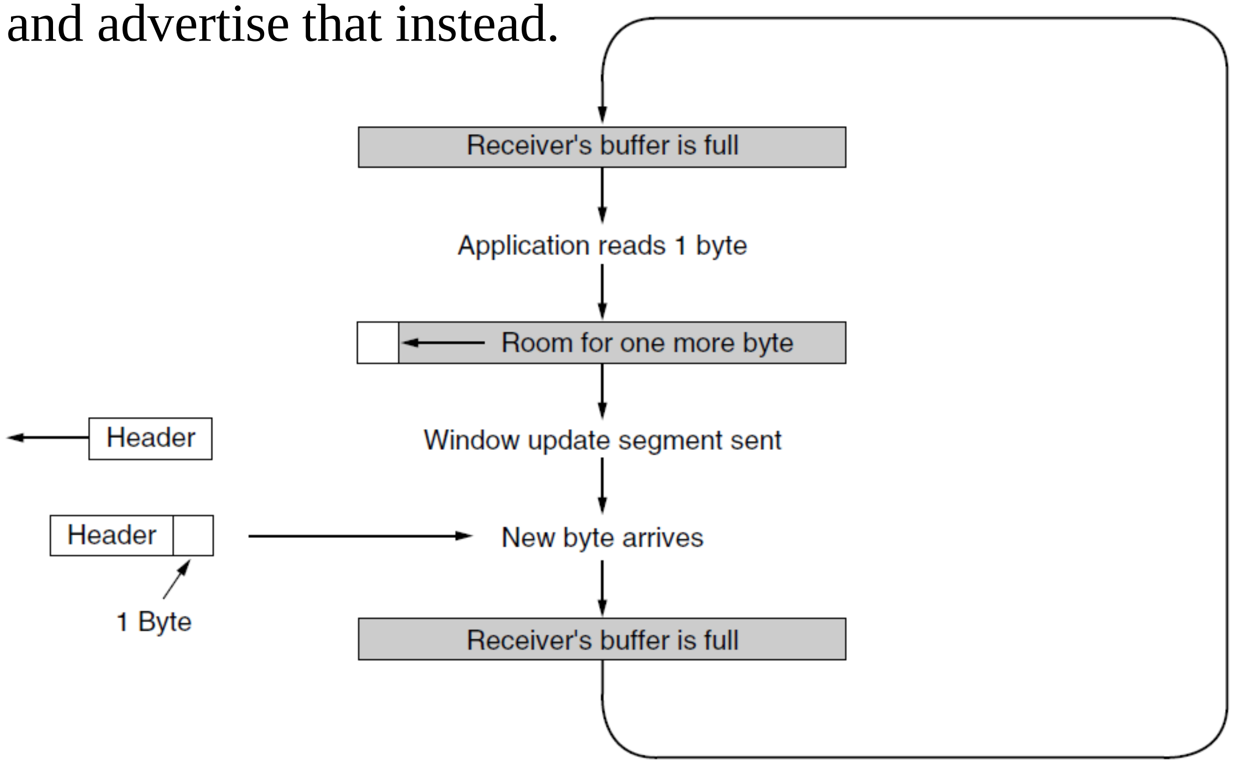
- TCP implements sliding windows, in order to improve throughput.
- Window sizes are measured in terms of bytes rather than packets;
- When the window is 0, the sender may not normally send segments, <u>with two exceptions</u>.
  - 1) Urgent data may be sent, for example, to allow the user to kill the process running on the remote machine.
  - 2) The sender may send a 1-byte segment to force the receiver to re-announce the next byte expected and the window size. This packet is called **a window probe**.
    - The TCP standard explicitly provides this option to prevent **deadlock** if a window update ever gets lost.

Nagle's Algorithm

- To reduce the load placed on the network by the receiver
  - **Delayed acknowledgements** is to delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride.
- To reduce the bandwidth used by a sender that <u>sends multiple short packets</u>.
  - **Nagle's algorithm** (Nagle, 1984): when data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged. Then send all the buffered data in one TCP segment and start buffering again until the next segment is acknowledged. ($\Rightarrow$ burstness)
  - Nagle's algorithm is not suitable for *interactive games*. A subtle problem is that Nagle's algorithm can sometimes interact with delayed acknowledgements to cause a temporary **deadlock**.
    - The receiver waits for data on which to piggyback an acknowledgement, and the sender waits on the acknowledgement to send more data.

Clark's Solution

- Another problem that can degrade TCP performance is **the silly window syndrome** (Clark 1982).
  - The problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data only 1 byte at a time.
  - **Clark's solution** is to force the receiver to wait *until it has a decent amount of space available* and advertise that instead.



The Silly Window Syndrome

- Nagle’s algorithm and Clark’s solution to the silly window syndrome are complementary.
  - Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at time.
  - Clark was trying to solve the problem of the receiving application sucking the data up from TCP a byte at a time.
  - Both solutions are valid and can work together. <u>The goal is for the sender not to send small segments and the receiver not to ask for them</u>.

TCP Sliding Window

- Another issue that the receiver must handle is that <u>segments may arrive out of order</u>.
  - The receiver will **buffer** the data until it can be passed up to the application in order.
  - Acknowledgements can be sent only when all the data up to byte acknowledged have been received.
    - This is called **a cumulative acknowledgement**.
    - Example: If the receiver gets segments 0, 1, 2, 4, 5, 6, and 7, it can acknowledge everything up to and including the last byte in segment 2. When the sender times out, it then retransmits segment 3. As the receiver has buffered segments 4 through 7, upon receipt of segment 3 it can acknowledge all bytes up the end of segment 7.

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP
    - TCP segment header
    - TCP connection establish
    - TCP sliding window
    - TCP timer management
    - TCP congestion control
    - BBR (Bottleneck Bandwidth and Round-trip propagation time)

TCP Timer Management

- TCP uses multiple timers (at least conceptually) to do its work.
  - The RTO (Retransmission TimeOut)
    - How long should the RTO be ? This problem is much more difficult in the transport layer than in data link protocols such as 802.11.
  - The Persistence timer
  - The Keepalive timer
  - The one used in TIME WAIT state

TCP Timer Management: RTO

- The RTO (Retransmission TimeOut)
  - How long should RTO be? This problem is much more difficult in the transport layer than in data link protocols such as 802.11.
  - In the data link layer, the expected delay is measured in microseconds and is highly predictable (i.e., has a low variance)

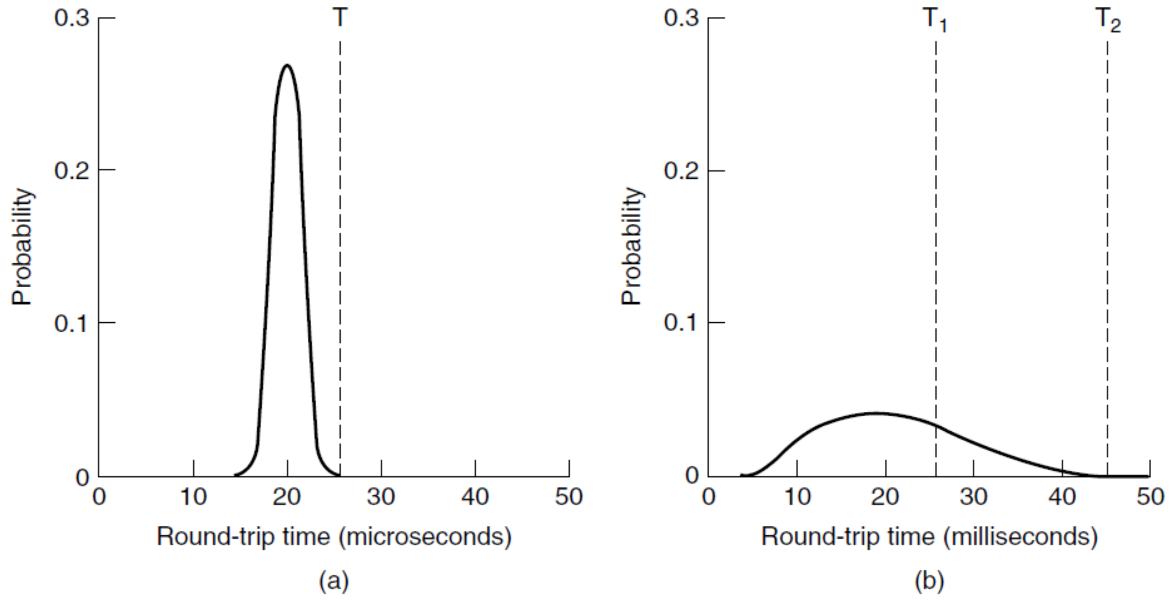


Figure 6-42. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

TCP Timer Management: RTO

- TCP is faced with a radically different environment. The pdf for the time it takes for a TCP acknowledgement to come back is <u>larger and more variable</u>.
  - If the timeout is set too short, say T1, in Fig.6-42(b), unnecessary retransmissions will occur.
  - If the timeout is set too long, say T2, in Fig.6-42(b), performance will suffer due to the long retransmission delay whenever a packet is lost.
  - Furthermore, <u>the mean and variance of the acknowledgement arrival distribution *can change rapidly within a few seconds* as congestion builds up or is resolved</u>.

TCP Timer Management: RTO

- The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.
- SRTT (Smoothed Round-Trip Time, **Jacobson**,1988)
  - Exponentially Weighted Moving Average (EWMA, R is the current estimate of the RTT)

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R \text{ where } \alpha = 7/8.$$

- RTTVAR (Round-Trip Time VARiration)
  - To make the timeout value sensitive to the variance in round-trip times as well as the smoothed round-trip time.

$$RTTVAR = \beta\, RTTVAR + (1 - \beta)\, |SRTT - R| \text{ where } \beta = 3/4.$$

$$RTO = SRTT + 4 \times RTTVAR$$

$$RTO = \min(1\text{sec}, RTO)$$

- RFC 2988

TCP Timer Management: RTO

- One problem that occurs with gathering the samples, R, of the round-trip time is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one.
- **Karn's algorithm**: do not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time.

TCP Timer Management: the persistent timer

- It is designed to prevent the following **deadlock**.
  - The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, <u>but the packet with the update is lost</u>. Now the sender and the receiver are each waiting for the other to do something.
  - When the persistence timer goes off, the sender transmits **a probe** to the receiver. The response to the probe gives the window size.
    - If it is still 0, the persistent timer is set again and the cycle repeats.
    - If it is nonzero, data can now be sent.

TCP Timer Management: the keepalive timer

- When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check whether the other side is still there.

TCP Timer Management: the TIME WAIT timer

- This timer is used in the TIME WAIT state while closing.
- It runs for **twice the maximum packet lifetime** to make sure that when a connection is closed, all packets created by it have died off.

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP
    - TCP segment header
    - TCP connection establish
    - TCP sliding window
    - TCP timer management
    - TCP congestion control
    - BBR (Bottleneck Bandwidth and Round-trip propagation time)

TCP Congestion Control (I)

- **The network layer** detects congestion when queues grow large at routers and tries to manage it, if only by *dropping packets*.
- It is up to **the transport layer** to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network.
  - In the Internet, TCP plays the main role in controlling congestion, as well as the main role in reliable transport.

TCP Congestion Control (II)

- TCP congestion control is based on a **AIMD** (**Additive Increase Multiplicative Decrease**) control law using a window and with **packet loss** as the binary signal.
- TCP maintains *a congestion window* (the sending window) and *a flow control window* (the receiving window)
  - **The congestion window (the sending window)** whose size is the number of bytes the sender may have in the network at any time.
    - The Congestion window is known only to the sender and is not sent over the links.
  - The corresponding rate is the window size divided by the round-trip time of the connection.
  - TCP adjusts the size of the congestion window according to the AIMD rule.

TCP Congestion Control (III)

- TCP congestion control is based on a **AIMD** (**Additive Increase Multiplicative Decrease**) control law using a window and with **packet loss** as the binary signal.
- TCP maintains *a congestion window* and *a flow control window*
  - **The flow control window** (or **the receiver window size**) which specifies the number of bytes that the receiver can buffer. <u>Receiver dictates its window size to the sender through TCP Header</u>.
  - Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows. In other words, <u>the amount of unacknowledged data at a sender</u> ≤ **min** (Receiver Window Size, Congestion Window Size)
  - TCP will **stop** sending data if either the congestion or the flow control window is temporarily full.

The TCP Segment Header

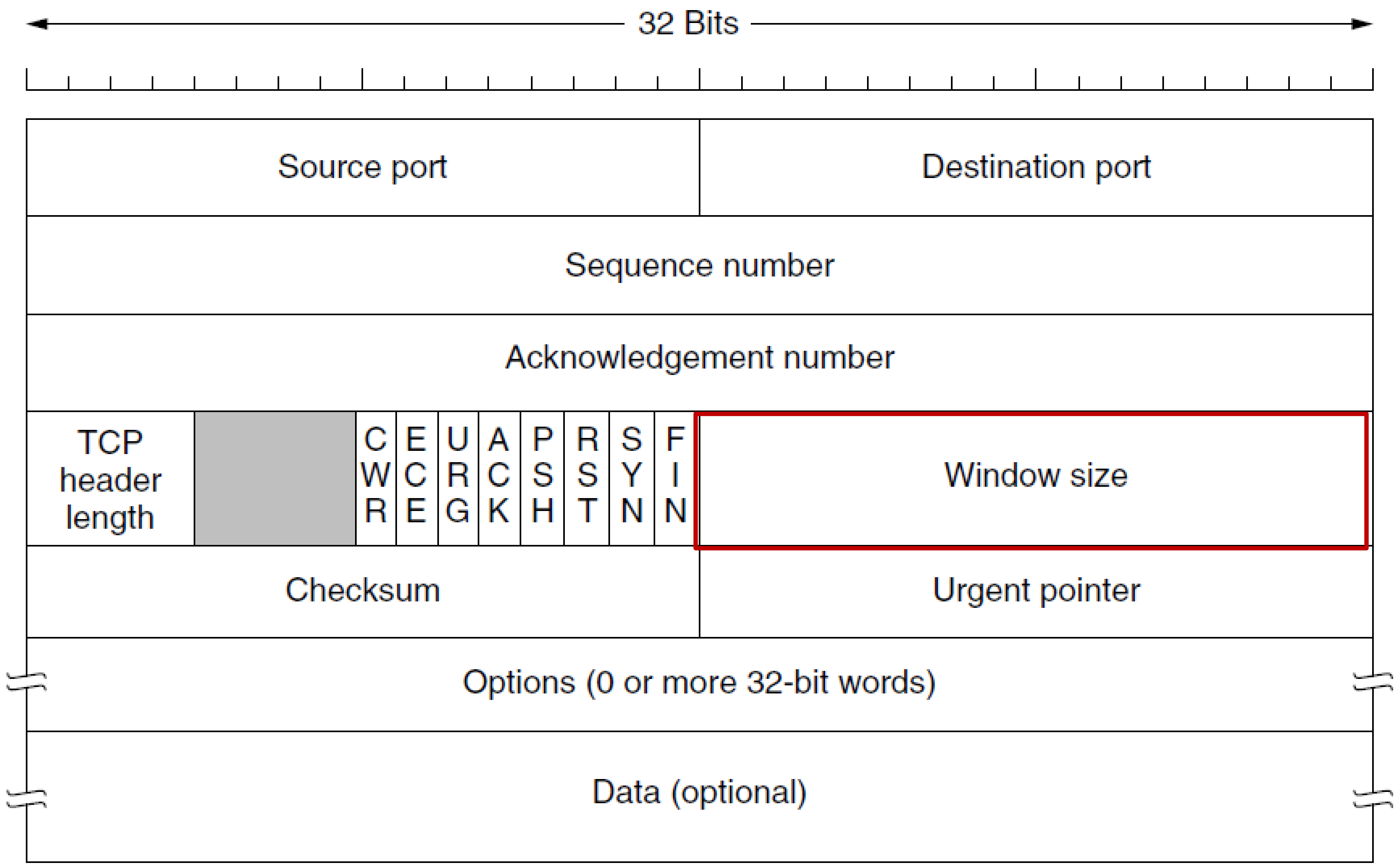


Figure 6-36. The TCP header.

TCP Congestion Control (IV)

- TCP congestion control is based on a **AIMD** (**Additive Increase Multiplicative Decrease**) control law using a window and with **packet loss** as the binary signal.
- All the Internet TCP algorithms assume that **lost packets** are caused by congestion and monitor timeouts.
- But, using packet loss as a congestion signal depends on transmission errors being relatively rare.
  - This is not normally the case for wireless links such as 802.11
    - Retransmission mechanism at the link layer
  - Most wires and optical fibers have lower bit-error rates.

TCP Congestion Control (V)

- During the implementation, we have two important questions:
  - 1) <u>The transmission rate of packets</u> which the sender will use
    - **Ack clock** ~ to estimate RTT
  - 2) <u>The size of the congestion window</u> so that we can take the most advantage of the network path while at the same time will not induce clog quickly
    - **Slow start**

TCP Congestion Control (VI)

- **The key observation** is that the acknowledgements return to the sender at about the rate that packets can be sent over <u>the slowest link in the path</u>. This is precisely the rate that the sender wants to use.
  - The acknowledgements reflect the times at which the packets arrived at the receiver after crossing the slow link.
    - This timing is known as **an ack clock**. It is an essential part of TCP.
  - By using an ack clock, TCP smooths out traffic and avoids unnecessary queues at routers.

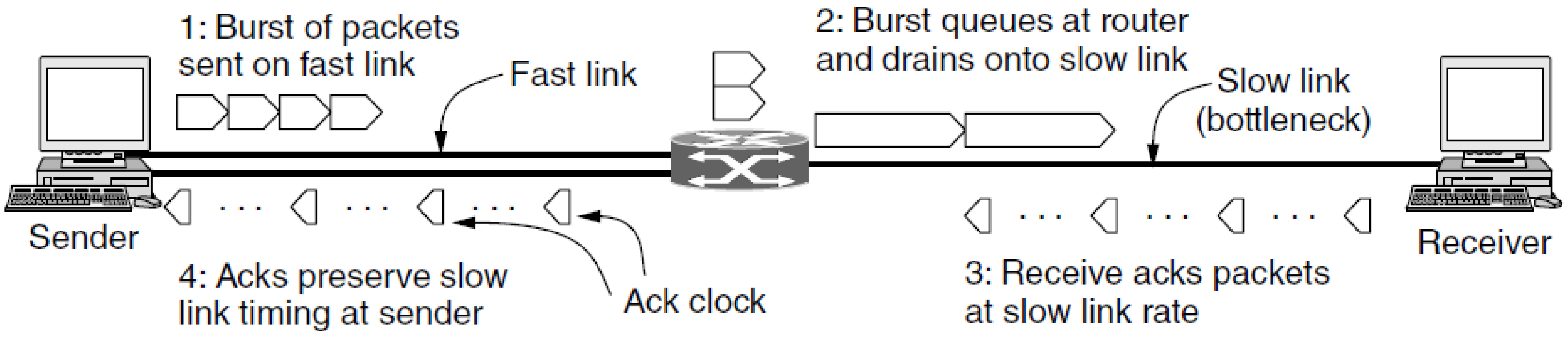

1: Burst of packets sent on fast link
Fast link
2: Burst queues at router and drains onto slow link
Slow link (bottleneck)
Sender
Receiver
4: Acks preserve slow link timing at sender
Ack clock
3: Receive acks packets at slow link rate

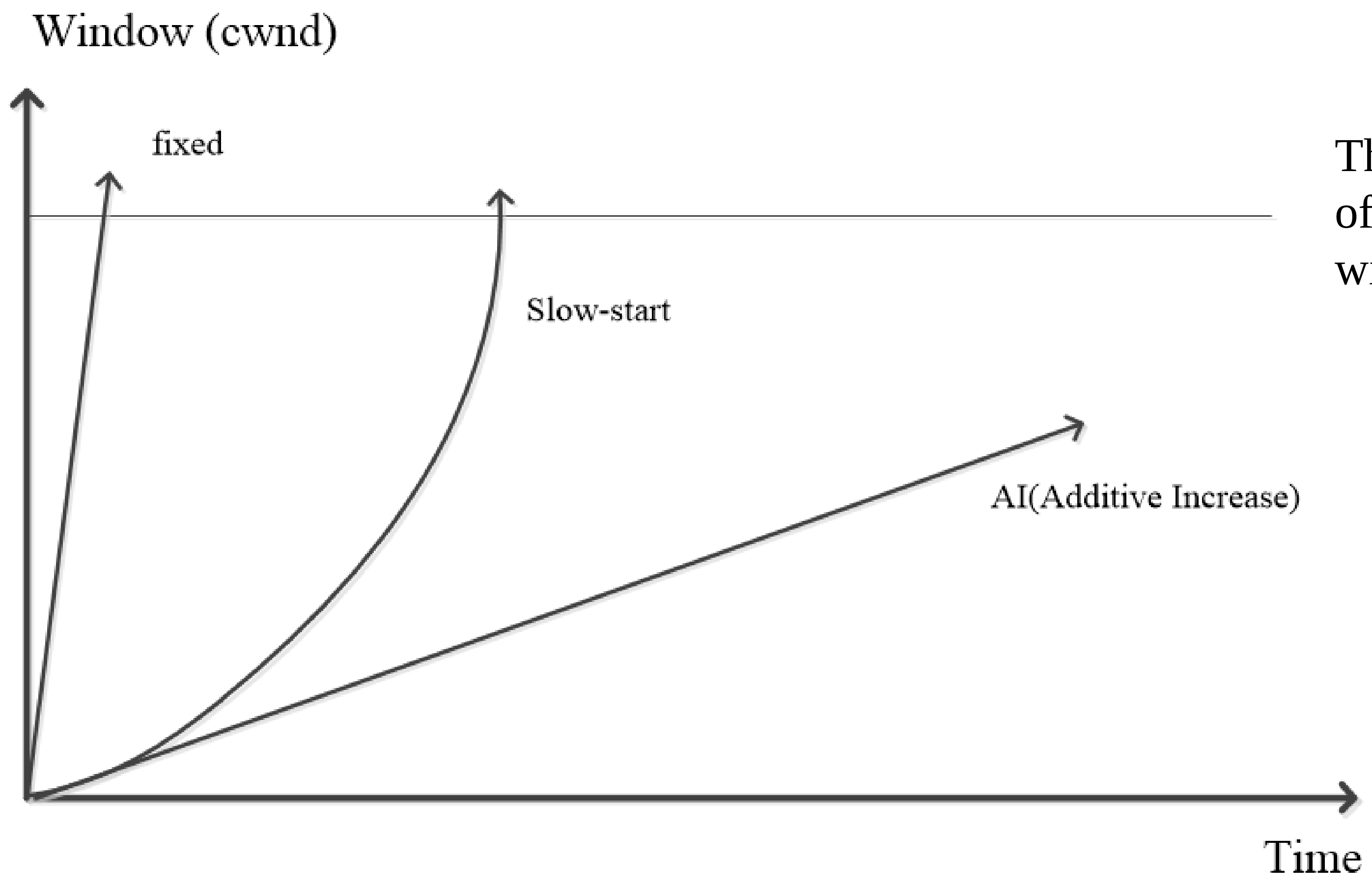
Figure 6-43. A burst of packets from a sender and the returning ack clock.

TCP Start Problem

- We want to quickly near the right rate, $\text{cwnd}_{\text{IDEAL}}$, but it varies greatly because TCP needs to work across a very large range of data rates and RTTs.
  - Fixed sliding window doesn’t adapt and is rough on the network layer (packet loss!)
  - AI (Additive Increase) with small bursts adapts cwnd gently to the network, but might <u>take a long time to become efficient</u>.

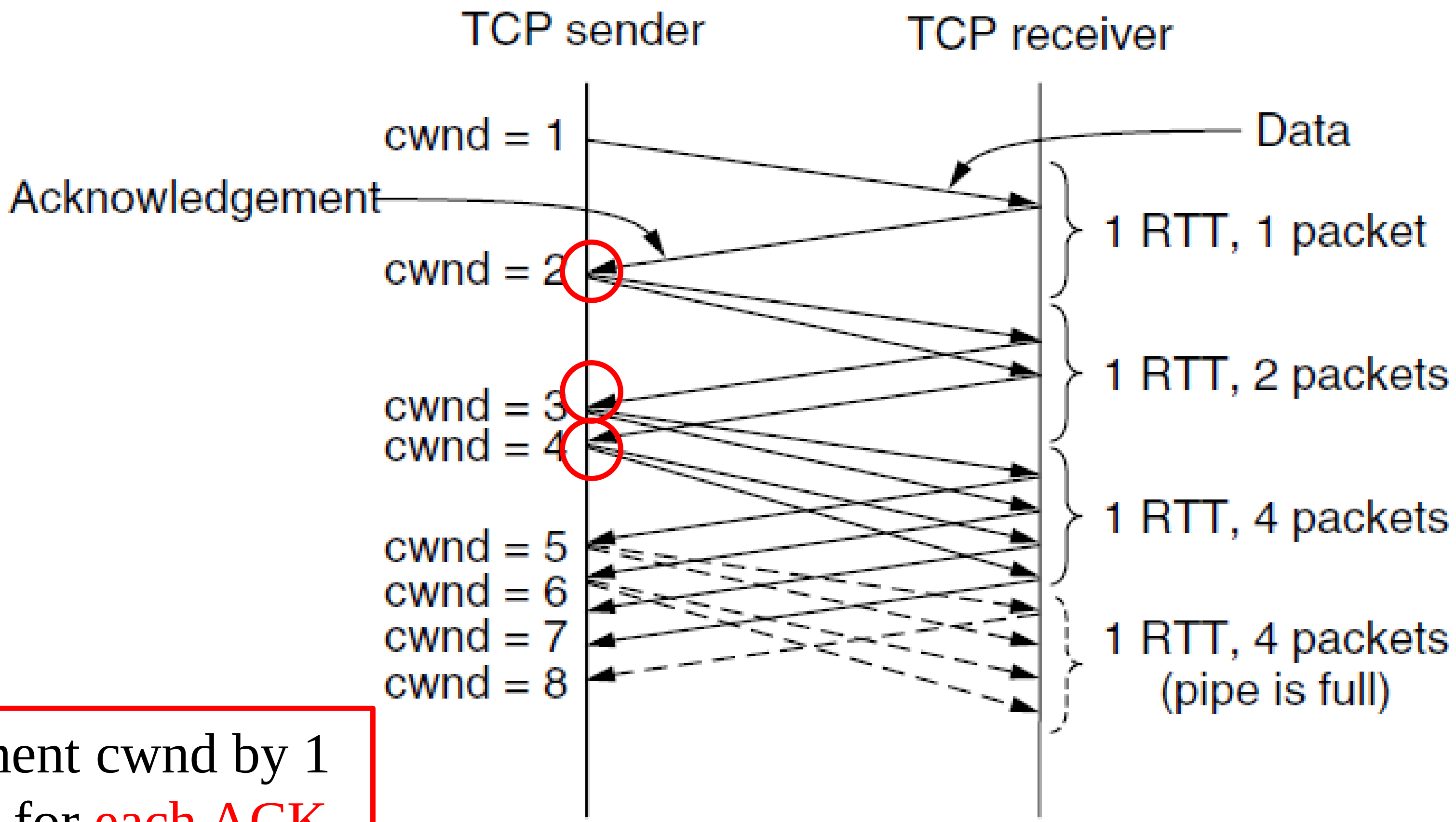
Slow-Start Solution (I)

- Start by **doubling** cwnd (the congestion window) every RTT
  - Exponential growth (1, 2, 4, 8, 16, …)
  - Start slow quickly reach large values.



The Optimal size of the congestion window

Slow-Start (Doubling) Timeline



Increment cwnd by 1 packet for each ACK.

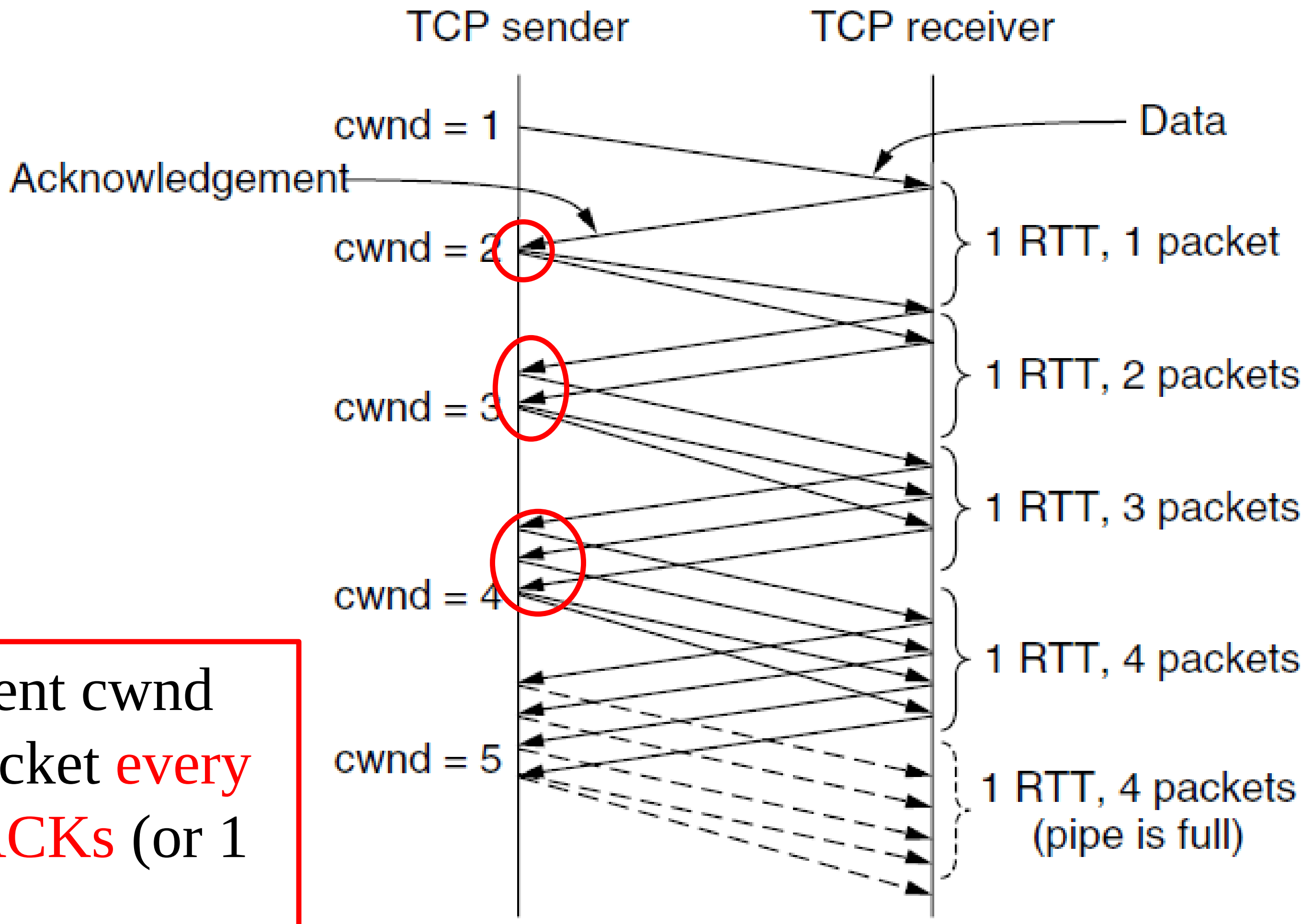
Figure 6-44. Slow start from an initial congestion window of one segment.

每收到一个ACK，就增加一个数据包，也就是一个变两个。

Slow-Start Solution (II)

- Because slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called **the slow start threshold**.
  - Initially the slow start threshold is set arbitrarily high, to the size of the flow control window, so that it will not limit the connection.
  - TCP keeps increasing the congestion window in slow start until
    - 1) **a timeout occurs** (~**packet loss**): the slow start threshold is set to be *half of the congestion window* and the entire process is *restarted*.
    - 2) **the congestion window exceeds the slow start threshold**: TCP switches from slow start to *additive increase*.
      - In this mode, the congestion window is increased by one segment every round-trip time.

Additive Increase Timeline



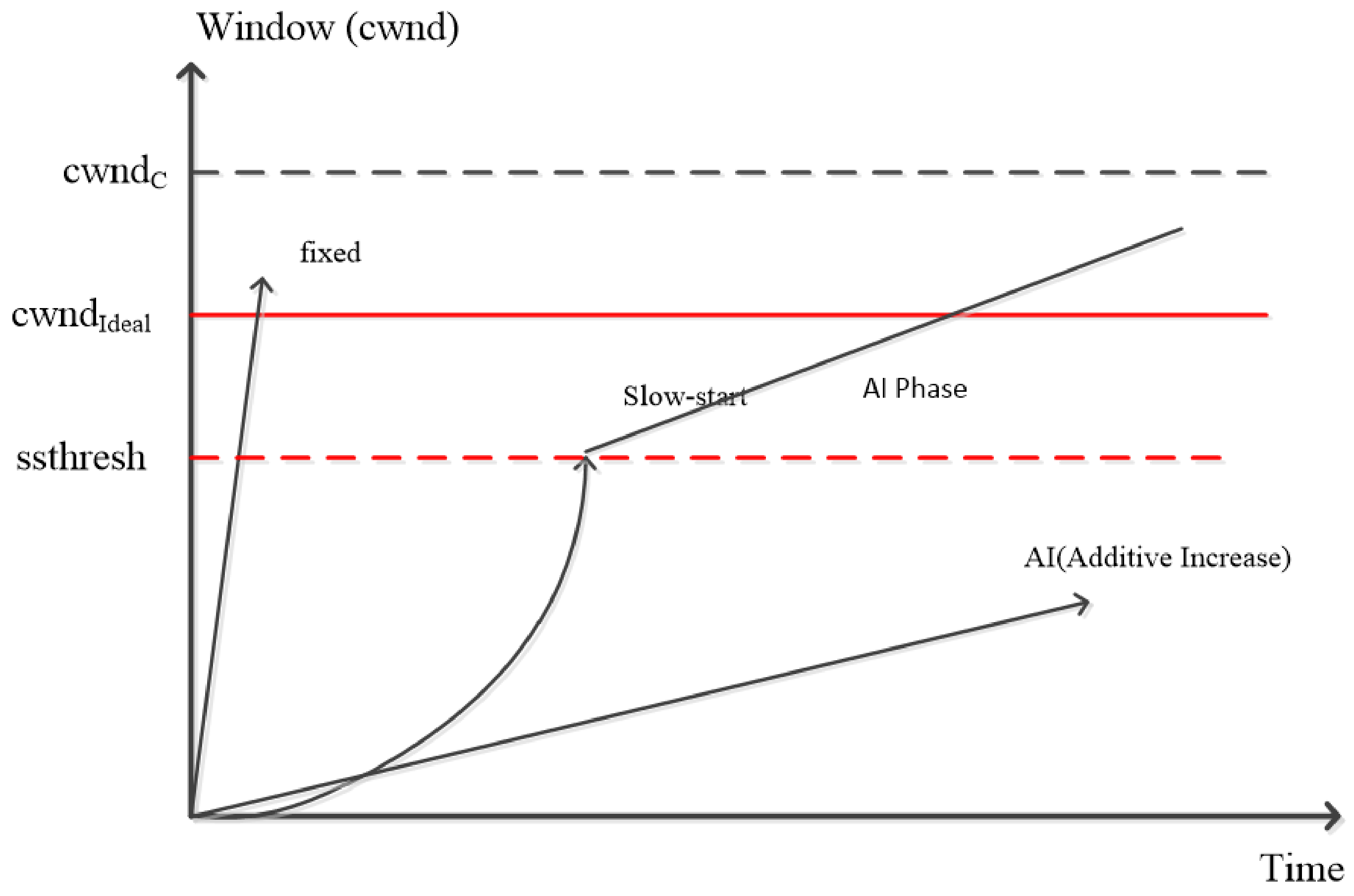
Increment cwnd by 1 packet every cwnd ACKs (or 1 RTT)

Figure 6-45. Additive increase from an initial congestion window of one segment.

从上图看完成一个完整的RTT，才增加一个数据包。如cwnd = 3时，只有收到三个ACKs，才增加一个数据包。

Slow-Start Solution (III)

- A mix of linear and the multiplication increase (Van Jacobson, 1988)
  - The whole idea is for a TCP connection to spend a lot of time with its congestion window close to the optimum value — not so small that throughput will be low, and not so large that congestion will occur.



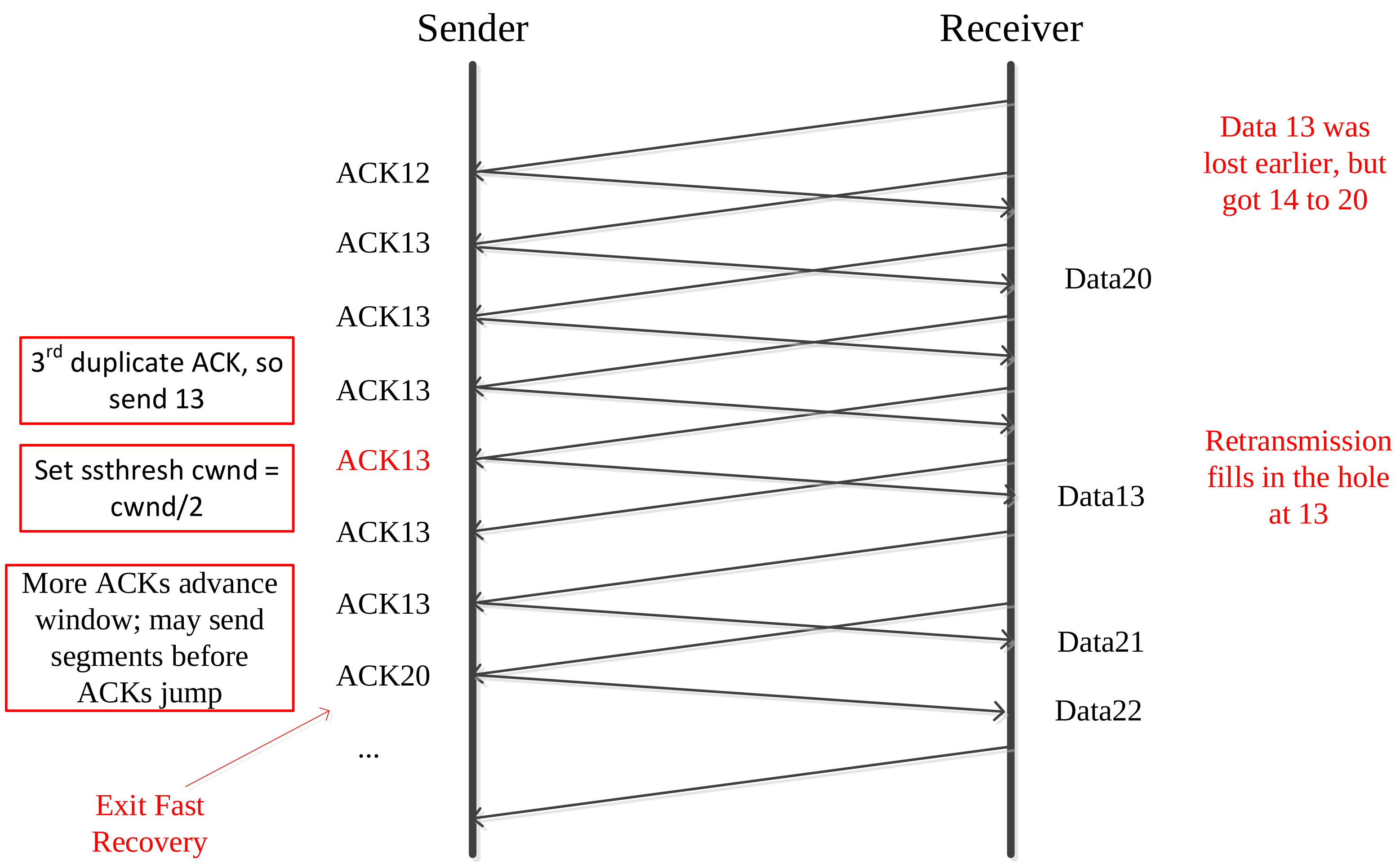
Inferring Loss from ACKs

- TCP uses **a cumulative ACK**
  - Carries highest in-order sequence number
  - Normally a steady advance

- **Duplicate ACKnowledgements** gives us hints about what data hasn't arrived
  - Tell us some new data did arrive, but it was not next expected segment.
  - Thus the next expected segment may be lost.
  - Arbitrarily treat **three duplicate acknowledgements** as a loss
  - Retransmit next expected segment before the retransmission timeout (**Fast Retransmission**)
  - The slow start threshold is still set of be half of the current congestion window. Slow start can be restarted by setting the congestion window to one packet.

Fast Retransmission (I)

- Treat **three duplicate Acknowledgments** as a loss
  - Retransmit next expected segment
  - Some repetition allows for reordering, but still detects loss quickly.



Sender
Receiver
Data 13 was lost earlier, but got 14 to 20
ACK12
ACK13
Data20
ACK13
3rd duplicate ACK, so send 13
ACK13
ACK13
Set ssthresh cwnd = cwnd/2
Data13
Retransmission fills in the hole at 13
ACK13
ACK13
More ACKs advance window; may send segments before ACKs jump
Data21
ACK20
Data22
...
Exit Fast Recovery

Fast Retransmission (II)

- It can repair **single segment loss** quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- The slow start threshold is still set of be half of the current congestion window. Slow start can be restarted by setting the congestion window to one packet.

TCP Congestion Control: Tahoe (1988)

- The maximum segment size is 1KB.
- Initially, the congestion window was 64 KB.
- A **timeout** occurred, so the slow-start threshold is set to be half of the congestion window, 32 KB, and the congestion window to 1KB for transmission 0.
- The congestion window grows exponentially until it hits the slow-start threshold (32KB). Now the window grows linearly. It is increased by one segment every RTT.

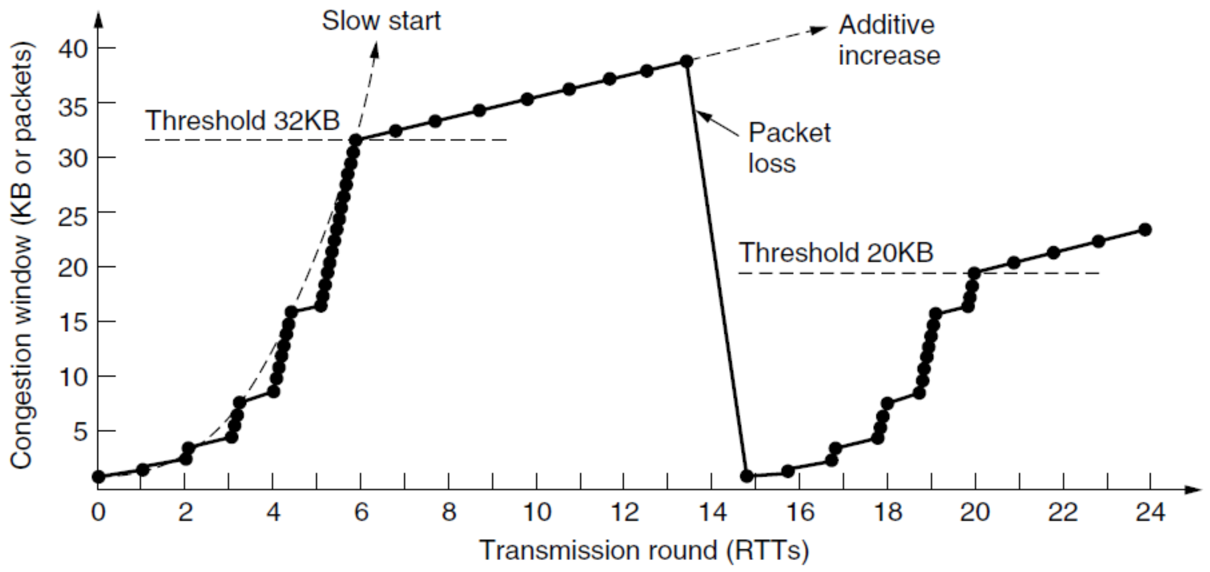


Figure 6-46. Slow start followed by additive increase in TCP Tahoe.

TCP Congestion Control: Tahoe (1988)

- The transmission in round 13, one of packets is lost in the network. This is detected when **three duplicate acknowledgements** arrive. At that time, the lost packet is retransmitted, the slow-start threshold is set to half of the current congestion window (40/2 = 20 KB), and slow start is initiated all over again.

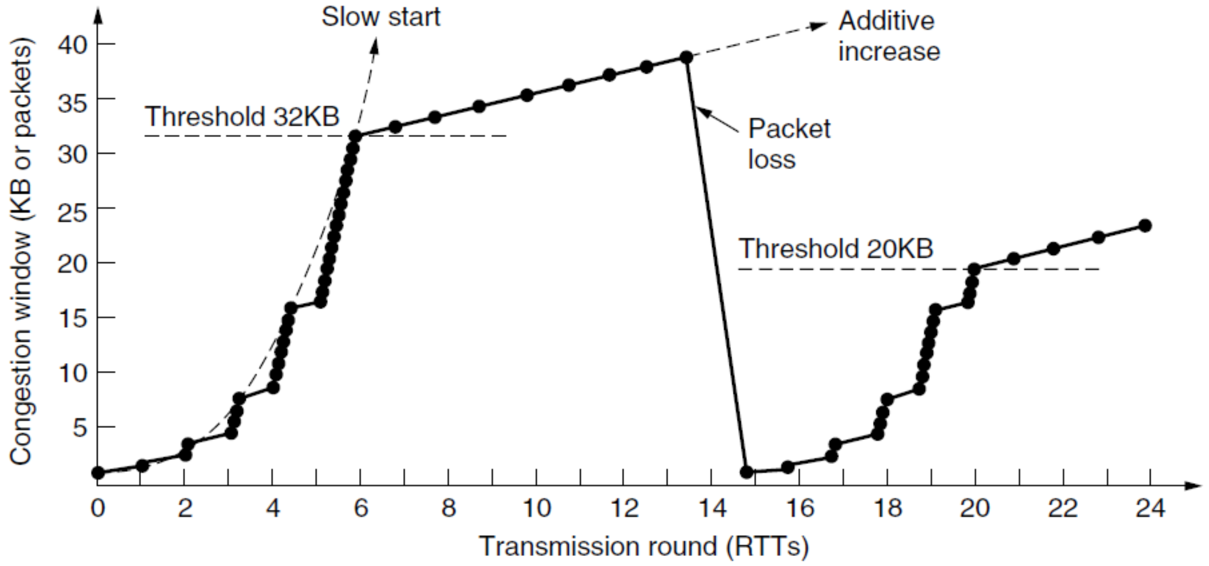


Figure 6-46. Slow start followed by additive increase in TCP Tahoe.

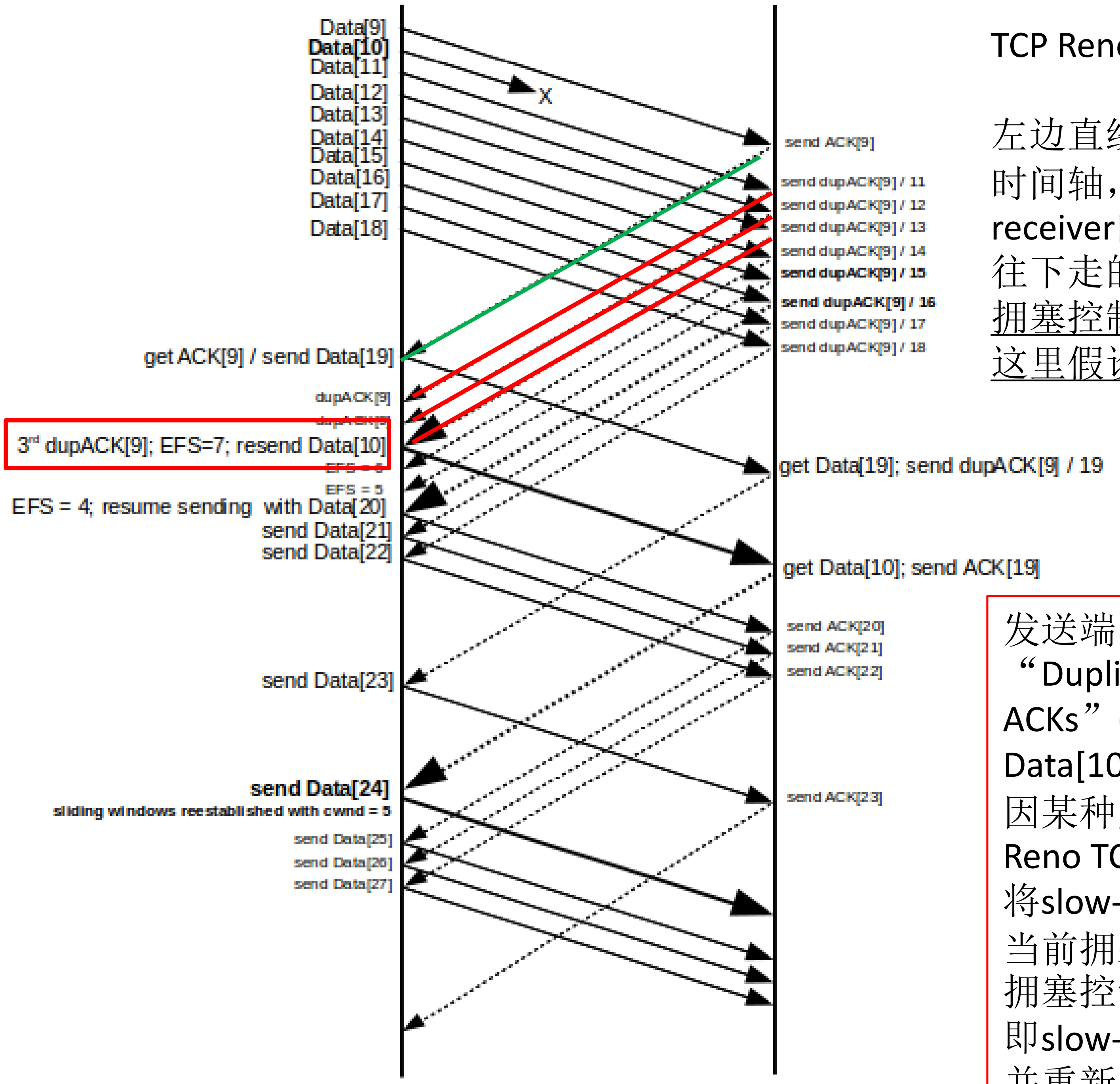
Inferring Non-loss from ACKs

- At the time of the fast retransmission, duplicate Acknowledgements also give us hints about what data has arrived.
  - Every time another duplicate acknowledgement arrives, it is likely that another packet has left the network.
  - It will be the segments after the loss.
  - Thus advancing the sliding window will not increase the number of segments stored in the network

Fast Recovery

- Fast recovery is the heuristic that implements this behavior.
  - Continue to send a new packet for each additional duplicate acknowledgement (<u>pretend further duplicate ACKs are the expected ACKs</u>)
  - To do this, duplicate ACKs are counted (including the three that triggered fast retransmission) until the number of packets in the network has fallen to the new threshold.
  - Reconcile views when the ACK jumps

TCP Reno版本中**Fast Recovery**

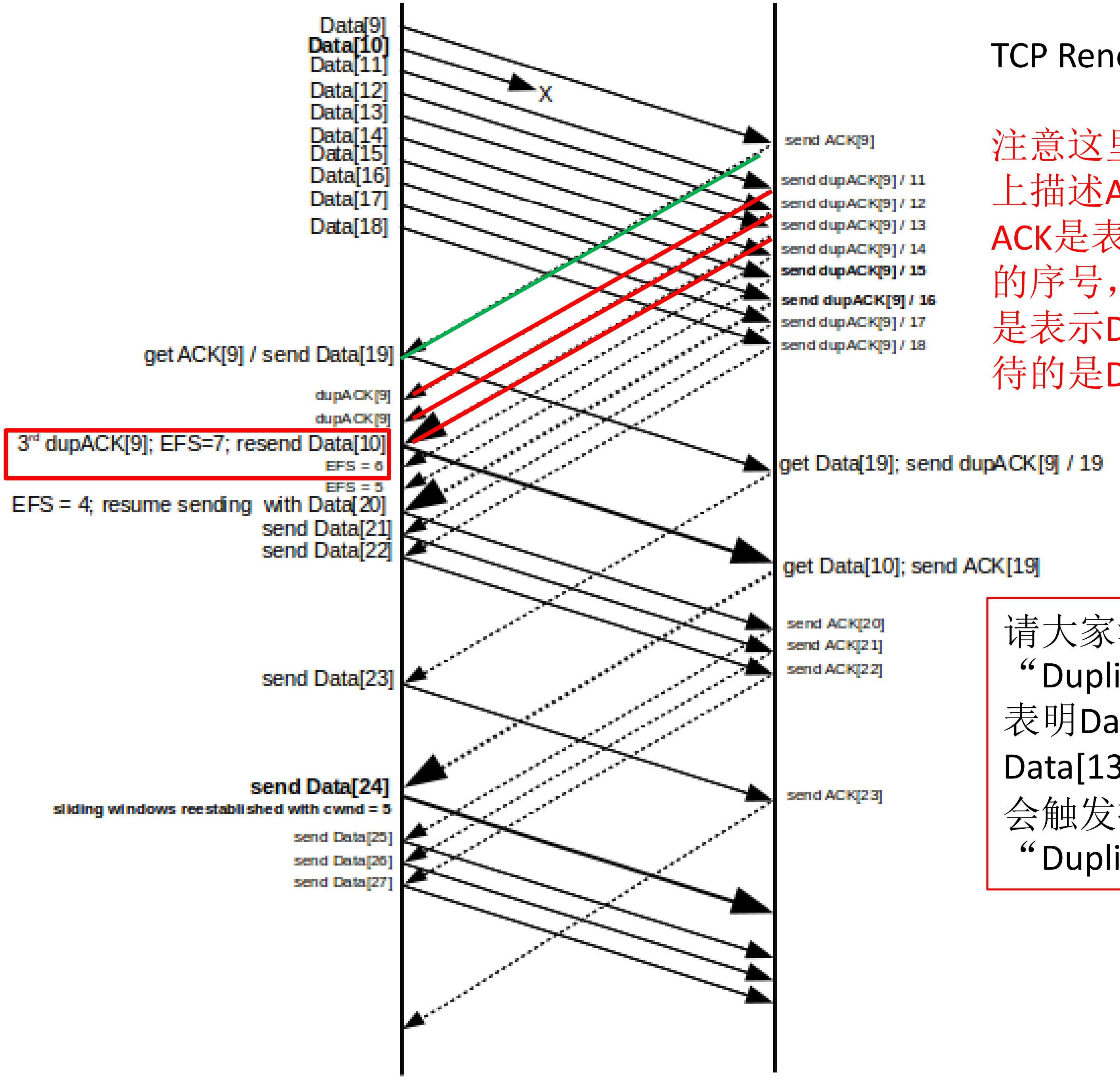


左边直线为发送端sender的时间轴，右边直线为接收端receiver的时间轴，时间轴是往下走的。

<u>拥塞控制窗口cwnd初始大小这里假设为10</u>。

发送端收到了三个“Duplicated ACKs”(dupACK[9])，表明Data[10]可能在传输过程中因某种原因丢失了。按照Reno TCP版马上做出决定：将slow-start threshold降到当前拥塞窗口的一半（注意拥塞控制窗口cwnd=10），即slow-start threshold为5，并重新发送Data[10]。

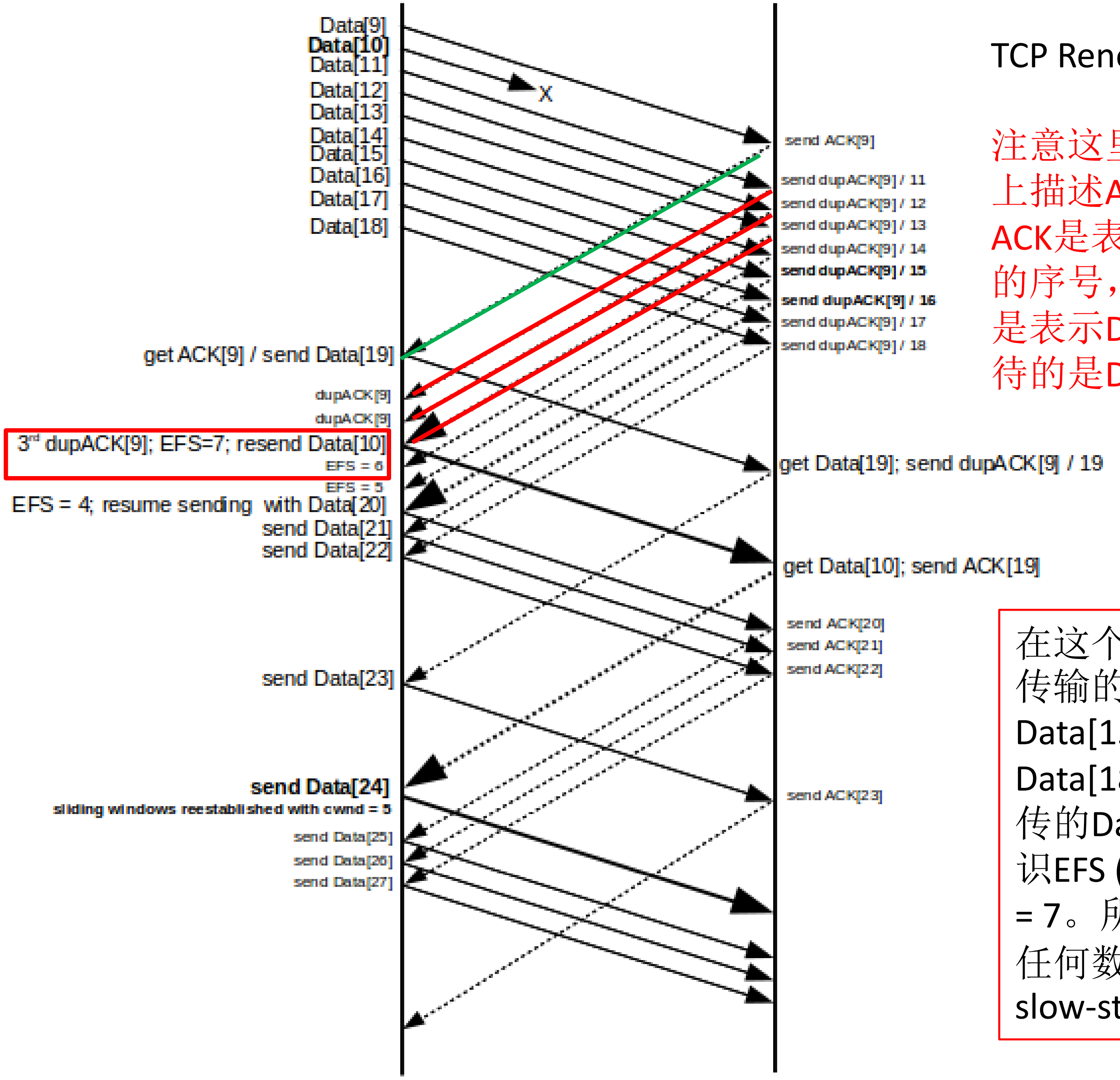
TCP Reno版本中**Fast Recovery**



注意这里图中ACK[9]，和书本上描述ACK有点出入，书本上ACK是表明期待下一个数据包的序号，而这张图中ACK[9]就是表示Data[9]已经收到，期待的是Data[10]

请大家注意这里三个“Duplicated ACKs”其实也表明Data[11]，Data[12]和Data[13]已经抵达，不然不会触发接收端发送三个“Duplicated ACKs”。

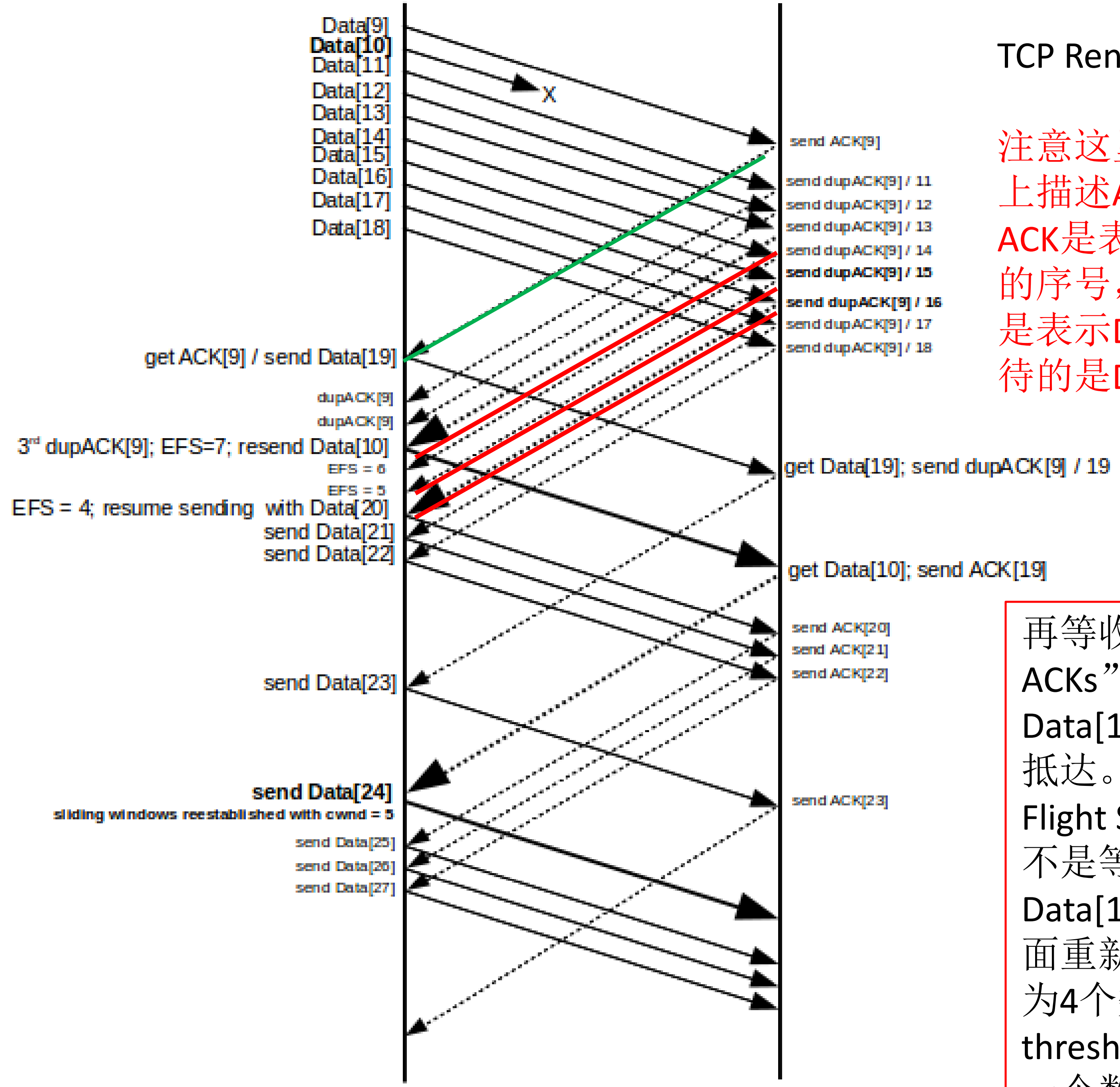
TCP Reno版本中**Fast Recovery**



注意这里图中ACK[9]，和书本上描述ACK有点出入，书本上ACK是表明期待下一个数据包的序号，而这张图中ACK[9]就是表示Data[9]已经收到，期待的是Data[10]

在这个时间点，仍在网络中传输的数据包有： Data[14], Data[15], Data[16], Data[17], Data[18]和Data[19]，以及重传的Data[10]，所以图中标识EFS (Estimated Flight Size) = 7。所以发送端暂停发送任何数据，因为已经超出了slow-start threshold (5)。

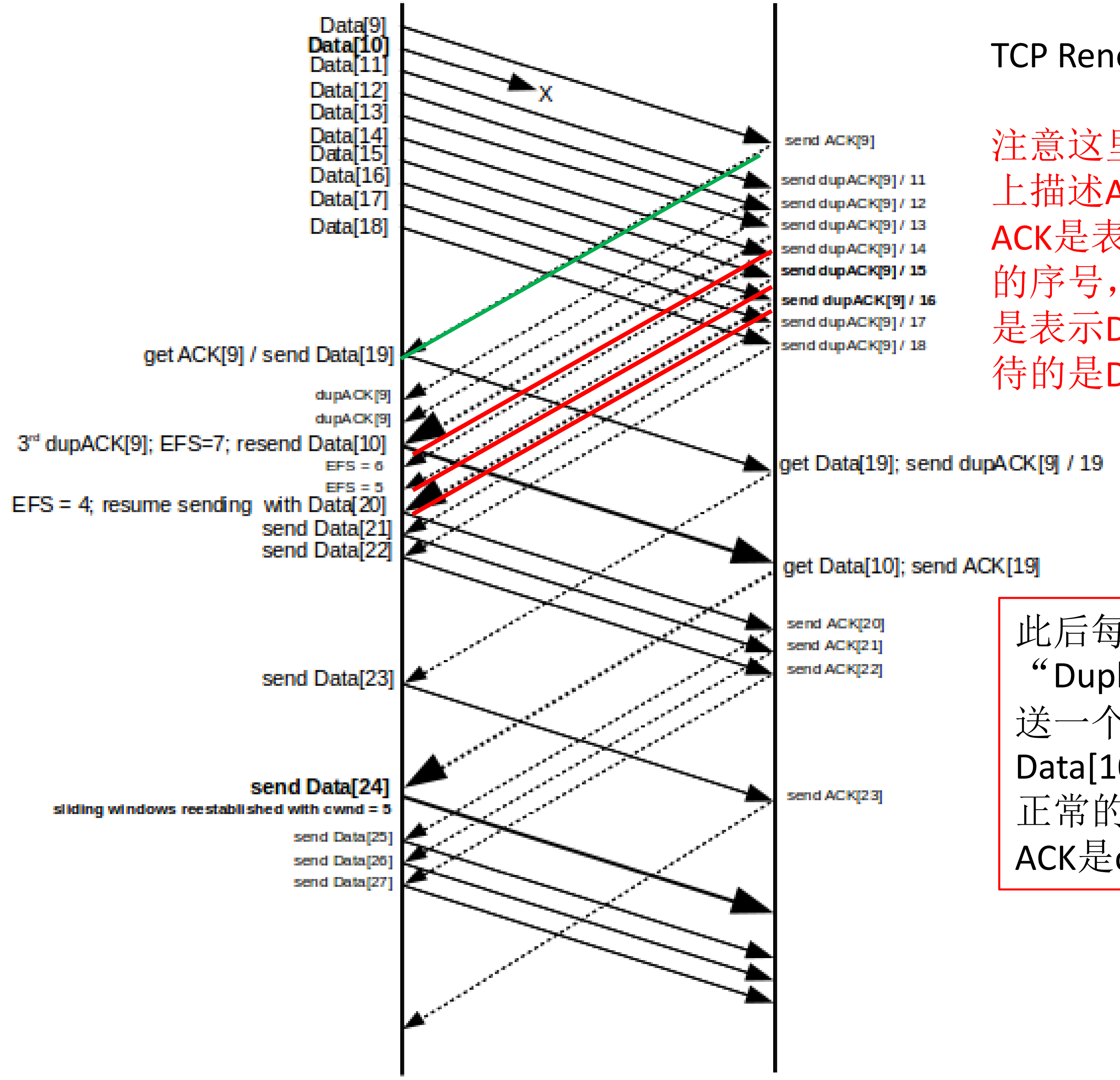
TCP Reno版本中**Fast Recovery**



注意这里图中ACK[9]，和书本上描述ACK有点出入，书本上ACK是表明期待下一个数据包的序号，而这张图中ACK[9]就是表示Data[9]已经收到，期待的是Data[10]

再等收到三个“Duplicated ACKs”后，隐含着Data[14], Data[15]和Data[16]已经安全抵达。现在EFS (Estimated Flight Size) 等于多少呢？是不是等于4，就是Data[17], Data[18]和Data[19]，外加前面重新发送Data[10]，总共为4个数据包。而slow-start threshold为5，所以可以发一个数据包Data[20]。

TCP Reno版本中**Fast Recovery**



注意这里图中ACK[9]，和书本上描述ACK有点出入，书本上ACK是表明期待下一个数据包的序号，而这张图中ACK[9]就是表示Data[9]已经收到，期待的是Data[10]

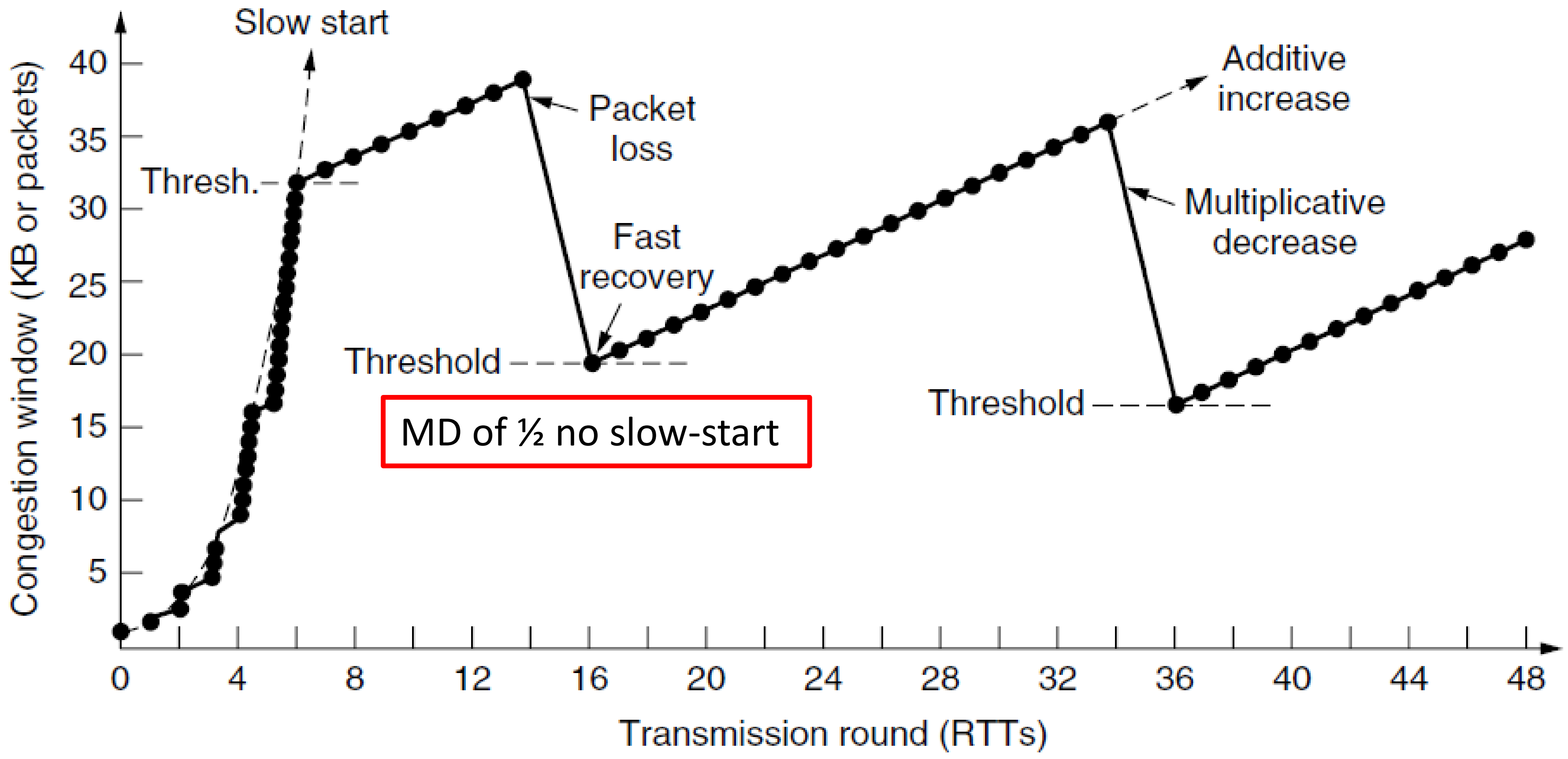
此后每收到一个“Duplicated ACK”，就发送一个新数据包，直至Data[10]抵达，ACK跳跃到正常的情况。不要忘记TCP ACK是cumulative ACK。

TCP Congestion Control: Reno

- **TCP Reno** (1990):
  - *Fast recovery*: to maintain the ack clock running with a congestion window that is the new threshold, or half the value of the congestion window at the time of the fast retransmission.
  - To do this, duplicate acknowledgements are counted until the number of packets in the network has fallen to the new threshold. From then on, *a new packet can be sent for each duplicate acknowledgement that is received*. One RTT after the fast re-transmission, the lost packet will have been acknowledged. At that time, the stream of duplicate acknowledgements will cease and fast recovery mode will be exited. The congestion window will be set to the new slow start threshold and grows by linear increase.
  - TCP avoids slow start, except when the connection is first started and when a timeout occurs.

TCP Congestion Control

- **TCP Reno** with its mechanisms for adjusting the congestion control has formed the basis for TCP congestion control for more than two decades. (TCP Tahoe + fast recovery)



TCP Congestion Control

- Two larger changes have also affect TCP implementations
  - 1) **SACK** (**Selective ACKnowledgements**) lists up to three ranges of bytes that have been received. With this information, the sender can more directly decide which packets to retransmit and track the packets in flight to implement the congestion window.
  - With SACK, TCP can recover more easily from situations in which *multiple packets are lost at roughly the same time*, since the TCP sender knows which packets have not been received.
  - [RFC2883 and RFC3517]

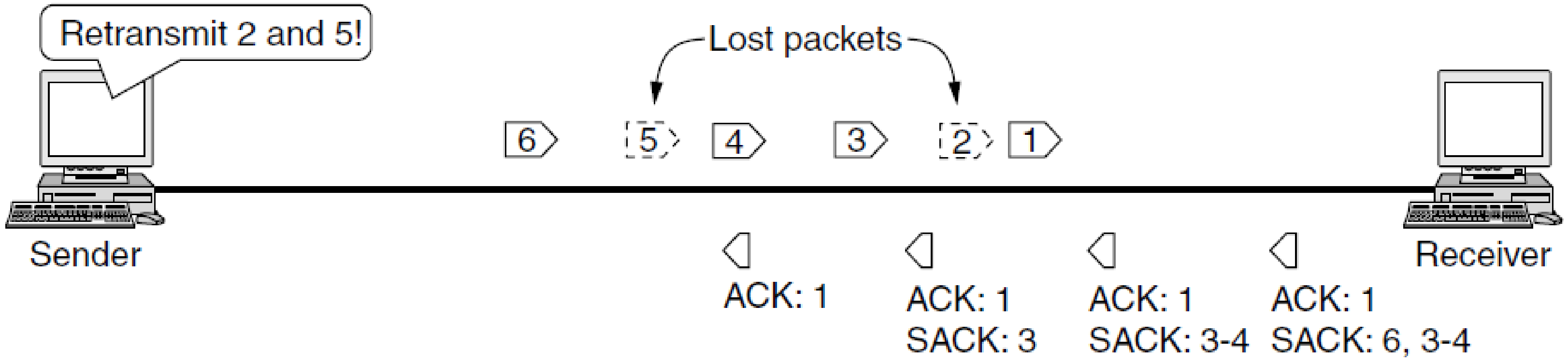


Figure 6-48. Selective acknowledgements.

TCP Congestion Control

- **ECN** (**Explicit Congestion Notification**)
  - ECN is *an IP layer mechanism* to notify hosts of congestion.
  - The use of ECN is enabled for a TCP connection when both the sender and receiver indicate that they are capable of using ECN by setting the ECE and CWR bits during the connection establishment.
    - The TCP receiver uses the **ECE** (**ECN-Echo**) flag to signal the TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network.
    - The sender tells the receiver that it has heard the signal by using the **CWR** (Congestion Window Reduced) flag, so that the TCP receiver knows the sender has slowed down and can stop sending the ECN-Echo.
  - If ECN is used, routers that support ECN will set a congestion signal on packets that can carry ECN flags when congestion is approaching, instead of dropping those packets after congestion has occurred.
  - <u>ECN requires both host and router support</u>. [RFC 3168]

The TCP Segment Header

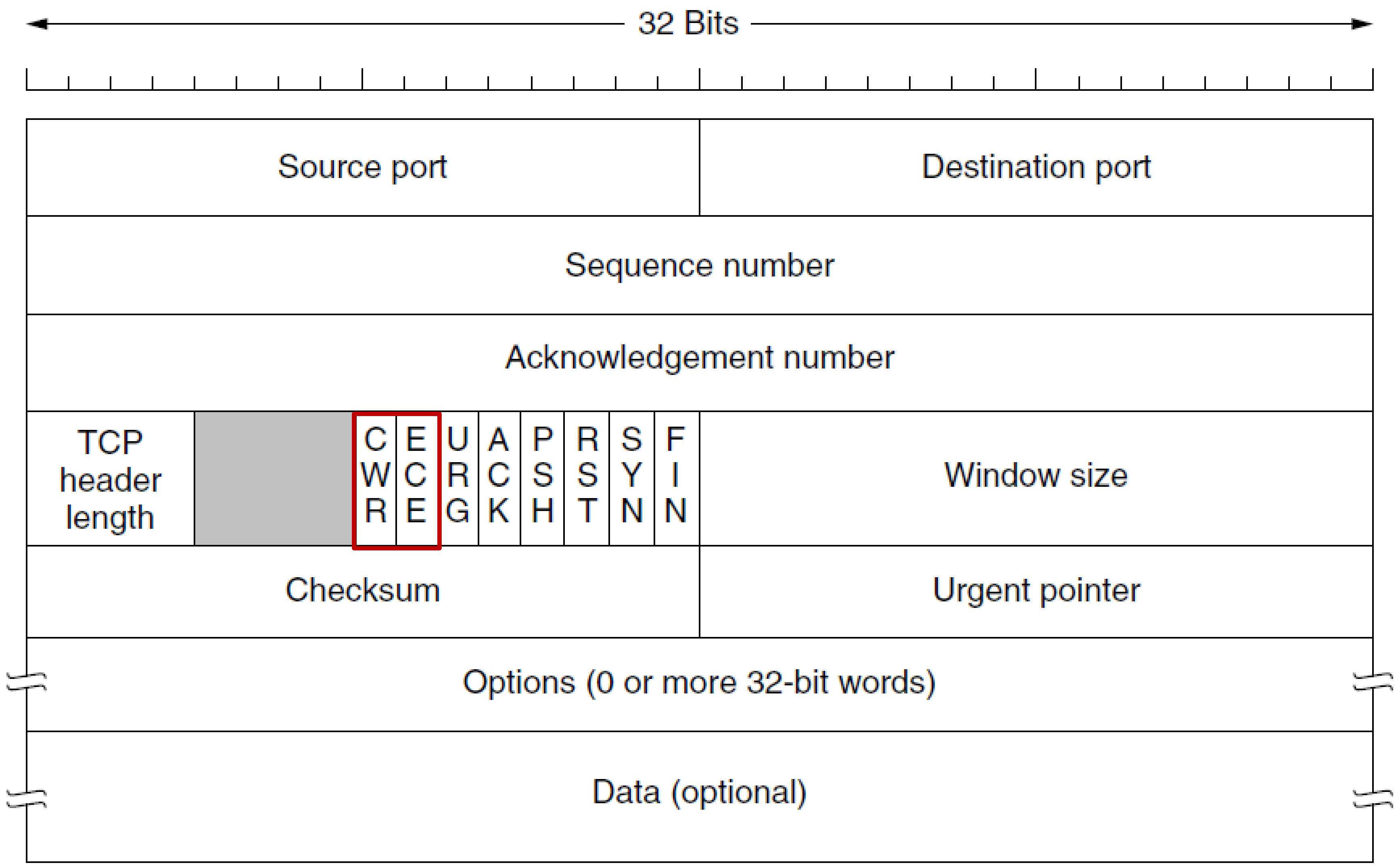


Figure 6-36. The TCP header.

TCP Reno, NewReno, and SACK

- Reno can repair one loss per RTT
  - Multiple losses cause a timeout
- NewReno further refines ACK heuristics
  - Repairs multiple losses without timeout
  - SACK (Selective Acknowledgement) is a better idea
    - Receiver sends ACK ranges so sender can retransmit without guess

Feedback Signals for Congestion Control

- Several possible signals, with different pros/cons

Signal	Example Protocol	Pros/Cons
Packet loss	TCP NewReno	Hard to get wrong/ Hear about congestion late
Packet delay	Compound TCP (Window)	Hear about congestion early/ need to infer congestion
Router indication	TCP with ECN (Explicit Congestion Notification)	Hear about congestion early/ require router support

Outline

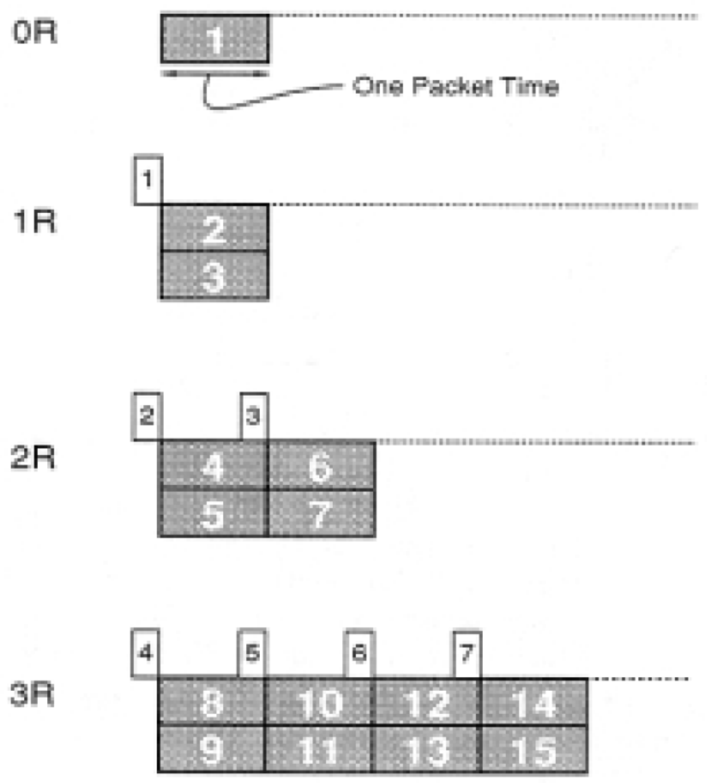
- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP

Outline

- Overview of the transport layer
- The internet transport protocols
  - UDP
  - TCP
    - TCP segment header
    - TCP connection establish
    - TCP sliding window
    - TCP timer management
    - TCP congestion control
    - BBR (Bottleneck Bandwidth and Round-trip propagation time)

BBR Congestion-based Congestion Control

Slow Start [1]



The horizontal direction is time. The continuous time line has been chopped into one-round-trip-time pieces stacked vertically with increasing time going down the page. The grey, numbered boxes are packets. The white numbered boxes are the corresponding acks.

As each ack arrives, two packets are generated: one for the ack (the ack says a packet has left the system so a new packet is added to take its place) and one because an ack opens the congestion window by one packet. It may be clear from the figure why an

Slow Start

- As each ack arrives, two packets are generated.
  - opens the window exponentially in time.
- The slow-start window increase isn't that slow: it takes time $R\log_2 W$ where R is the round-trip-time and W is the window size in packets.

BBR [3]

- TCP congestion control was created in the 1980s — interpreting packet loss as "congestion". This equivalence was true at the time but was because of technology limitations, not first principles.

– As NIC evolved from Mbps to Gbps and memory chips from KB to GB, the relationship between **packet loss** and **congestion** became more tenuous.

  – When bottleneck buffers are large, loss-based congestion control keeps them full, causing **buffer bloat**.

    - <u>Delaying congestion events for senders </u>(when network devices in along a network path have buffers that are too large, a TCP sender with a large congestion window can send at a rate that far exceeds the capacity of the network before it ever receives a loss signal.)

  – When bottleneck buffers are small, loss-based congestion control misinterprets loss as a signal of congestion, <u>leading to low throughput</u>.

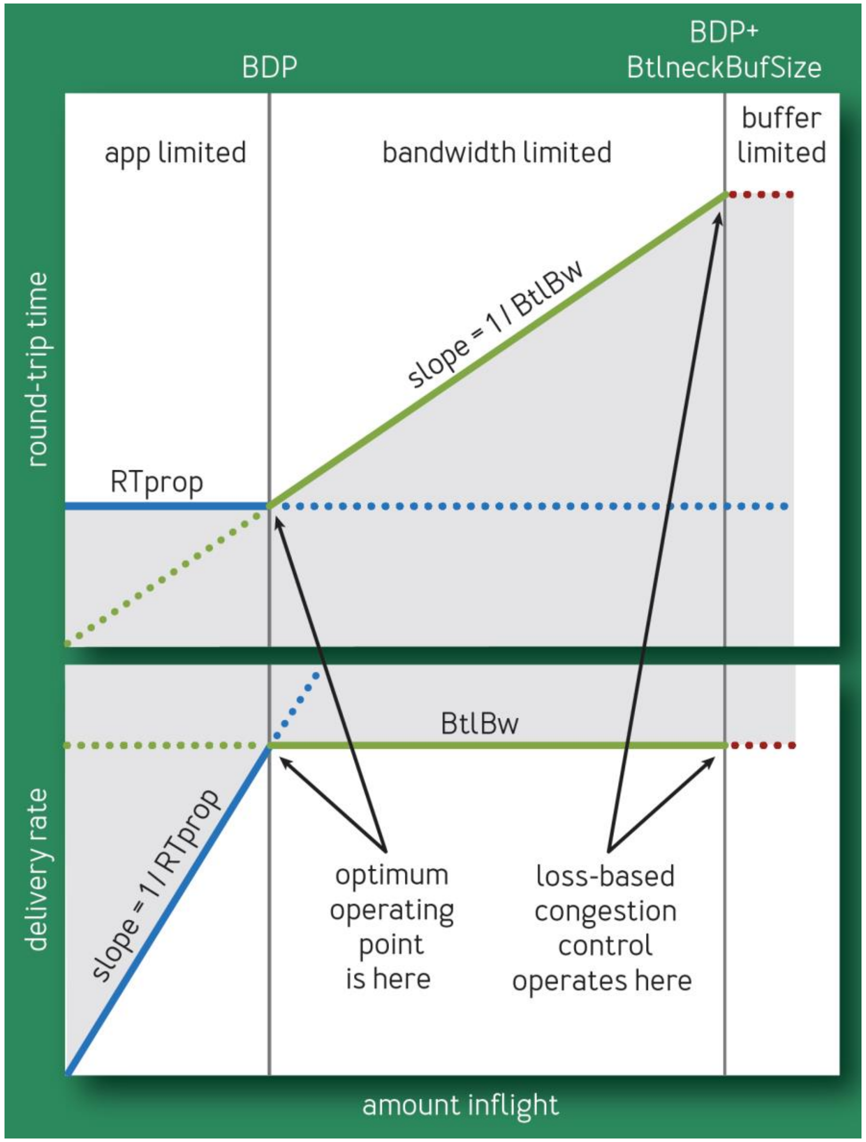
BBR [3]

- Fixing these problems requires an alternative to loss-based congestion control. First we have to understand **where** and **how** network congestion originates.
  - Congestion and bottleneck

– At any time, a full-duplex TCP connection has exactly *one slowest link* or *bottleneck* in each direction. The **bottleneck** is important because
  - It determines the connection's **maximum data-delivery rate**.
  - It is where persistent queues form.

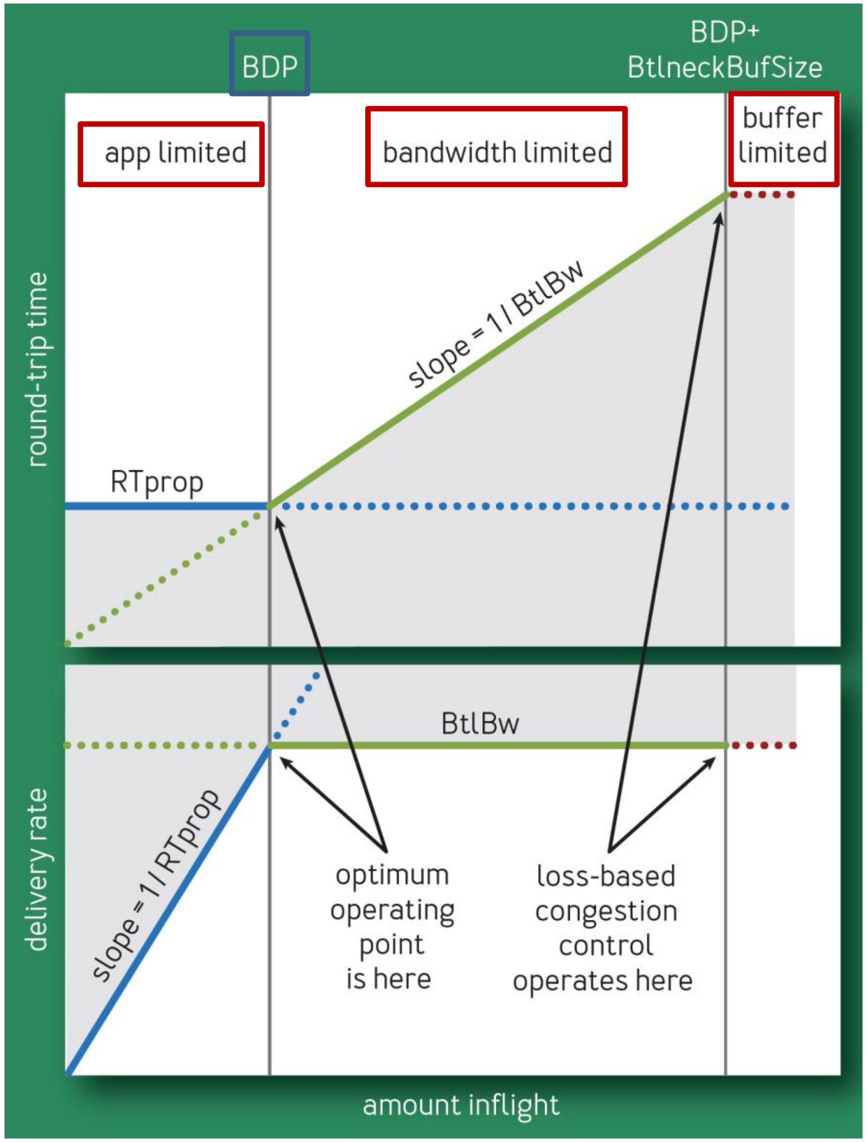
- From TCP's viewpoint, an arbitrary complex path behaves as <u>a single link with the same RTT and bottleneck rate</u>.
  - Two physical constraints, **RTprop** (round-trip propagation time) and **BtlBw** (Bottleneck Bandwidth), bounds transport performance.
    - If the network path were a physical pipe, **RTprop** would be <u>its length</u> and **BtlBW** <u>its minimum diameter</u>.

BBR [3]



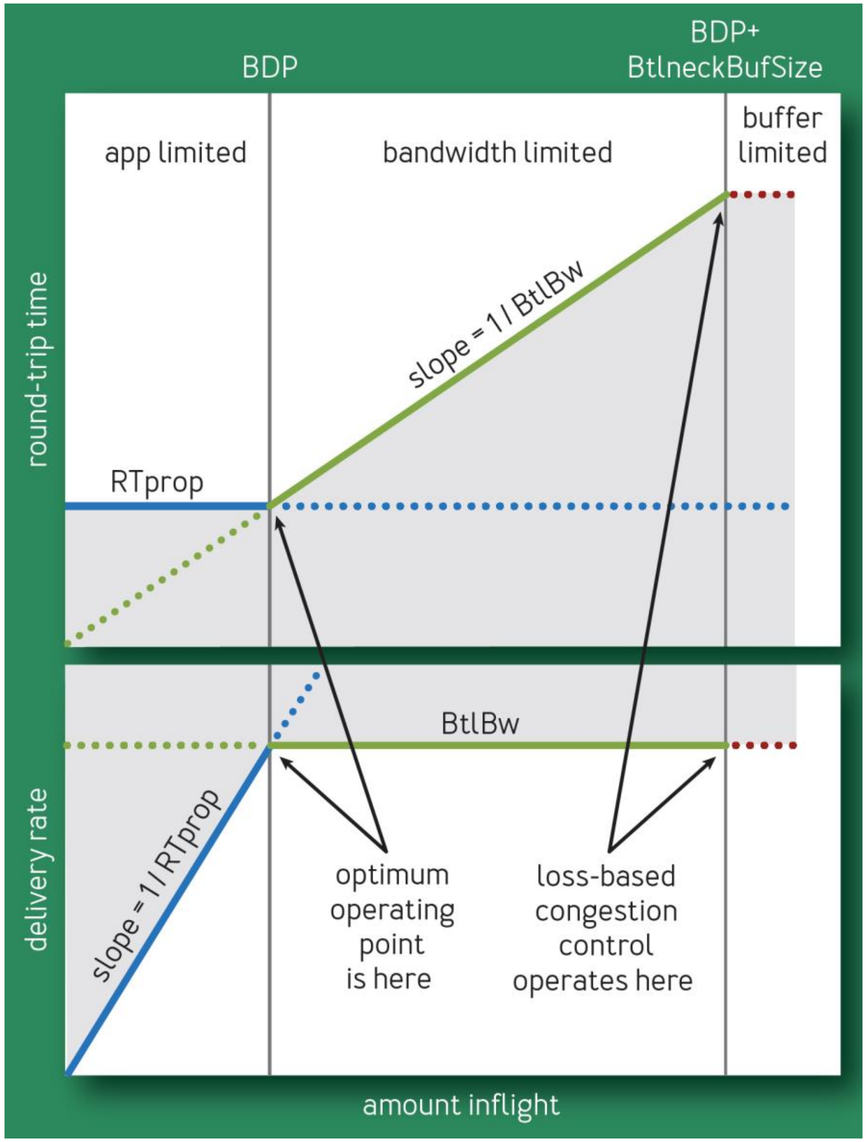
- Fig.1 shows RTT and delivery rate variation with the amount of data in flight (data sent but not yet acknowledged).
- Blue lines show the RTprop constraint, green lines the BtlBw constraint, and red lines the bottleneck buffer.
  - When there isn't enough data in flight to fill the pipe, RTprop determines behavior; otherwise, BtlBw dominates.
- Transitions between constraints result in three different regions {app-limited, bandwidth-limited, and buffer-limited}

BBR [3]



Bandwidth-delay product (**BDP**) is a measurement of how many bits can fill up a network link. It gives the **maximum** amount of data that can be transmitted by the sender at a given time before waiting for acknowledgment. Thus it is the maximum amount of unacknowledged data.
data in flight (data sent but not yet acknowledged) = BtlBw × RTprop.
The pipe is *full* pass this point (BDP).

BBR [3]



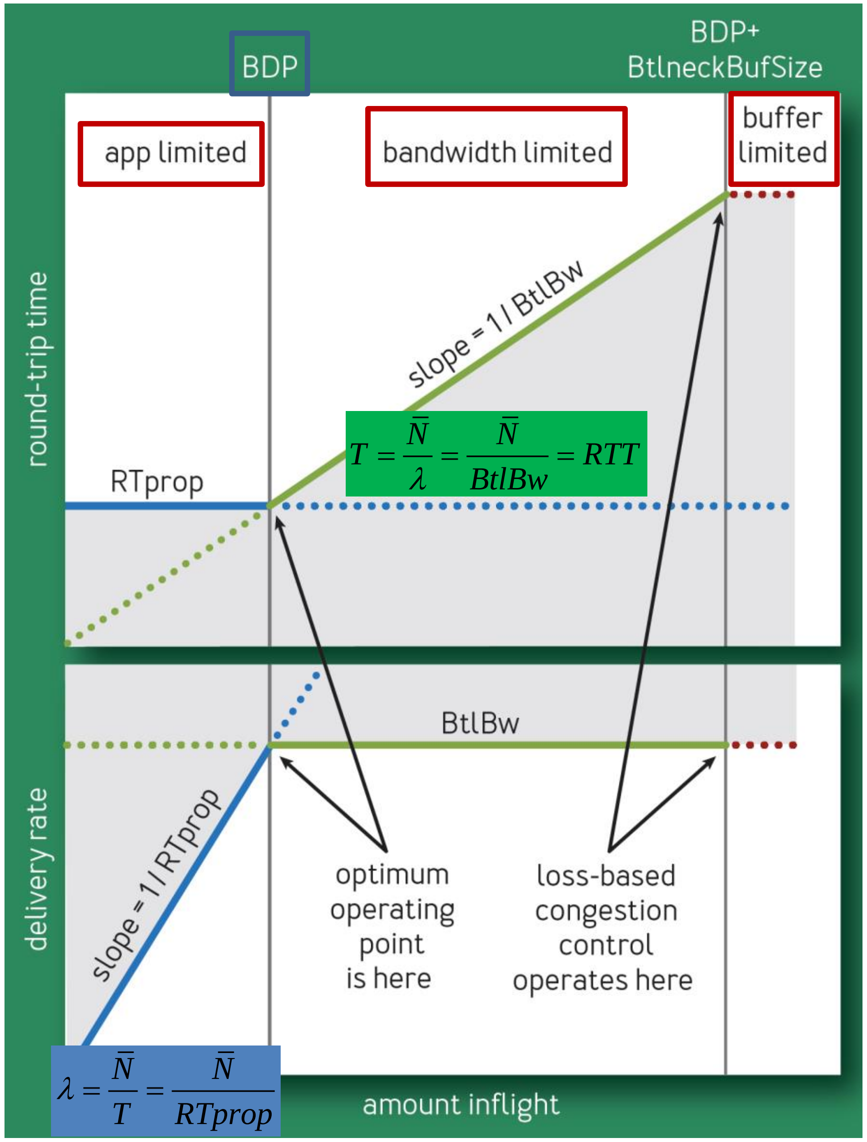
Data in flight (data sent but not yet acknowledged). ~ $\bar{N}$

Little's Result (in steady state) — which relates the average number in the system to the average arrival rate λ and the average time spent in that system T, namely

$$\boxed{\bar{N} = \lambda T}$$

$\bar{N}$ <u>is the data in flight in the network</u>.
T (~ RTT) is proportional to the amount of packets in flight $\bar{N}$, <u>the lower bound is **RTprop**</u> (*min* RTT).
λ (~ the delivery rate) is inversely proportional to the RTT and proportional to the amount of packets in flight $\bar{N}$, <u>the upper bound is **BltBw**</u> (*max* λ or the delivery rate).

BBR [3]



The inflight – BDP excess creates **a queue** at the bottleneck, which results in the linear dependence of RTT on inflight data (Little's Law).
Packets are dropped when the excess exceeds the buffer capacity.

Loss-based congestion control operates at the *right* edge of the bandwidth-limited region, delivering full bottleneck bandwidth at the cost of high delay and frequent packet loss.

BBR operates at the *left* edge of bandwidth-limited region, maximizing delivered bandwidth while minimizing delay and loss

BBR [3]

- "**app-limited**" (application limited) region — The application runs out of data to fill the network.
- When there isn't enough data in flight to fill the pipe, RTprop determines behavior; otherwise BtlBw dominants.
- Rtprop and BtlBw obey **an uncertainty principle** (the Network's Heisenberg Uncertainty Principle): whenever one can measured, the other cannot.
  - The pipe has to be overfilled to find its capacity, which creates a queue that obscures the length of the pipe. (BtlBw but not RTprop)
  - An application running a request/response protocol might never send enough data to fill the pipe and observe BtlBW (RTprop but not BtlBw)

BBR [3]

- Characterizing the bottleneck
  - **Rate balance**: a connection runs with the highest throughput and lowest delay when the bottleneck packet arrival rate equals BtlBw.
    - This condition guarantees that the bottleneck can run at 100% utilization.
  - **Full pipe**: the total data in flight is equal to the BDP = BtlBw × RTprop.
    - This condition guarantees there is enough data to prevent bottleneck starvation but not over fill the pipe.

  - BtlBw and RTprop vary over the life of a connection, so they must be continuously estimated.
  - BtlBw and RTprop are completely independent
    - RTprop can change (for example, on a route change) but still have the same bottleneck, or BtlBw can change (for example, when a wireless link changes rate) without the path changing.

BBR [3]

- How to estimate **RTprop**?
  - TCP currently tracks RTT (the time interval from sending a data packet until it is acknowledged) since it is required for loss detection.
  - At any time *t*, $RTT_t = RTprop_t + \eta_t$, where η ≥ 0 represents the "**noise**" introduced by queues along the path, the receiver's delay ack strategy, ack aggregation, etc.
    - RTprop is a physical property of the connection's path and changes only when the path changes (物理链路能达到的最小时延， 而RTT是实测值).
  - An unbiased, efficient estimator at time T is

$$RTprop = RTprop + \min(\eta_t) = \min(RTT_t) \qquad \forall t \in [T - W_R, T]$$

- a running min over time window W_R which is typically tens of seconds to minutes.

BBR [3]

- How to estimate BtlBw?
  - Unlike RTT, nothing in the TCP requires implementations to track bottleneck bandwidth, but a good estimate results from tracking delivery rate.
  - Average delivery rate between send and ack is the ratio of data delivery to time elapsed: *deliveryRate* = Δdelivered / Δ*t*.
  - This rate must be ≤ the bottleneck rate, the arrival amount is known exactly so all the uncertainty is in the Δ*t*, which must be ≥ the true arrival interval; thus, the ratio must be ≤ the true delivery rate, which is, in turn, **upper-bounded** by the bottleneck capacity). (BtlBw是物理链路能达到的最大速率，传输速率是实测值。)
  - A windowed-max of delivery rate is an efficient, unbiased estimator of BltBw:

$$BltBW = \max\left(deliveryRate_t\right) \qquad \forall t \in \left[T - W_B, T\right]$$

    - where the time window W_R is typically six to ten RTTs.

BBR [3]

- The core BBR algorithm has two parts:
  - 1. When an ack is received

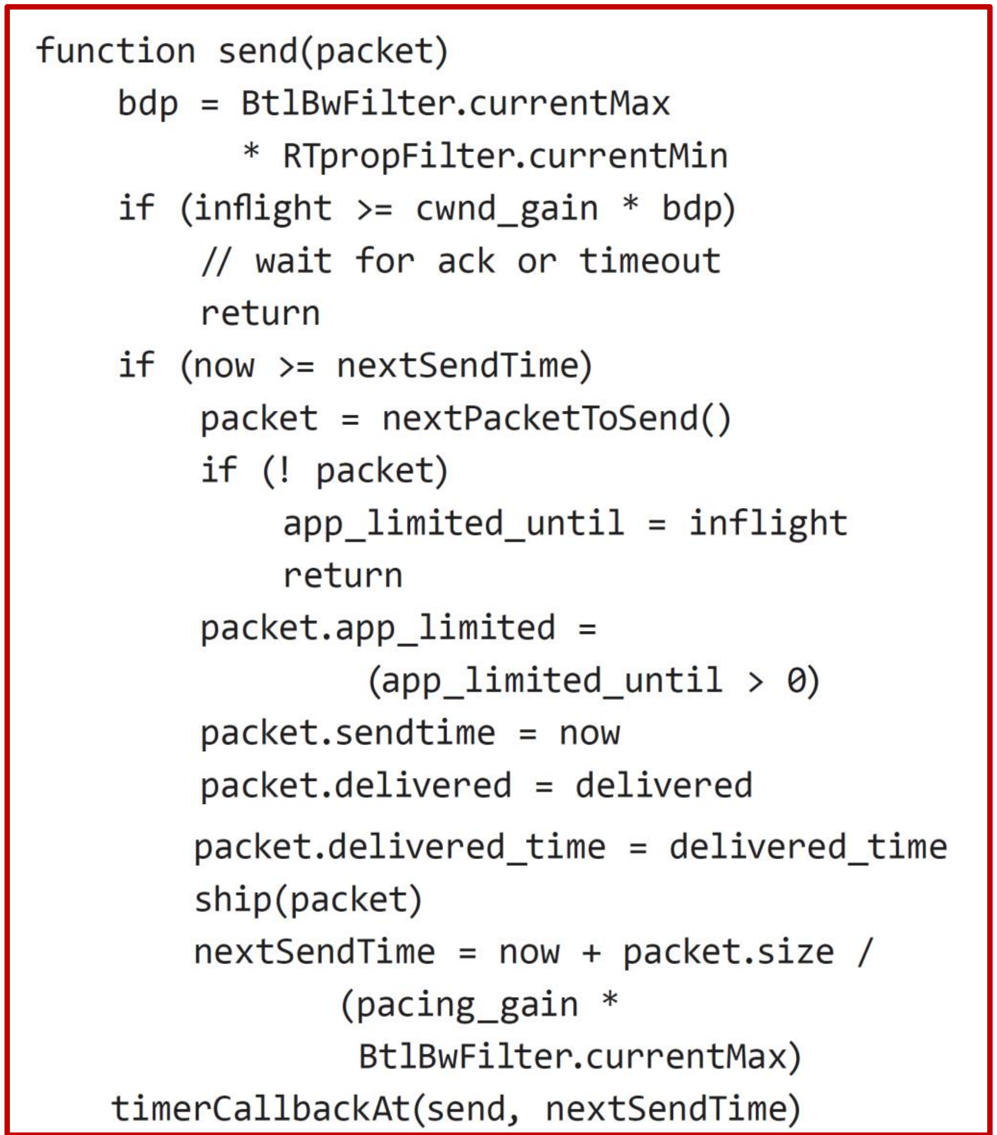
```
function onAck(packet)
  rtt = now - packet.sendtime
  update_min_filter(RTpropFilter, rtt)
  delivered += packet.size
  delivered_time = now
  deliveryRate = (delivered - packet.delivered)
                 /(now - packet.delivered_time)
  if (deliveryRate > BtlBwFilter.currentMax
      || ! packet.app_limited)
      update_max_filter(BtlBwFilter,
                        deliveryRate)
  if (app_limited_until > 0)
      app_limited_until - = packet.size
```

The **if** checks address the uncertainty issue:

1) BtlBw is a hard upper bound on the delivery rate so a measured delivery rate larger than the current BltBw estimate must mean the estimate is too low.
2) The code here decides which samples to include in the bandwidth model so it reflects network, not application limits.

BBR [3]

- The core BBR algorithm has two parts:
  - 2. When data is sent: to match the packet-arrival rate to the bottleneck link’s departure rate, BBR **paces** every data packet.
    - **pacing_rate**: BBR’s primary control parameter, to lower the burstiness.
    - **cwnd_gain**: bounds inflight to a small multiple of the BDP to handle common network and receiver pathologies.



```
function send(packet)
    bdp = BtlBwFilter.currentMax
          * RTpropFilter.currentMin
    if (inflight >= cwnd_gain * bdp)
        // wait for ack or timeout
        return
    if (now >= nextSendTime)
        packet = nextPacketToSend()
        if (! packet)
            app_limited_until = inflight
            return
        packet.app_limited =
                (app_limited_until > 0)
        packet.sendtime = now
        packet.delivered = delivered
        packet.delivered_time = delivered_time
        ship(packet)
        nextSendTime = now + packet.size /
               (pacing_gain *
                BtlBwFilter.currentMax)
    timerCallbackAt(send, nextSendTime)
```

BBR [3]

- Rather than using events such as loss or buffer occupancy, which are only weakly correlated with congestion, BBR starts from Kleinrock's formal model of congestion and its associated optimal operating point.
- The rate and amount BBR sends is solely a function of the estimated BtlBw and RTprop.
  - BtlBw and RTprop can be estimated sequentially
- BBR runs purely on the sender and does not require changes to the protocol, receiver, or network, making it incrementally deployable.

References

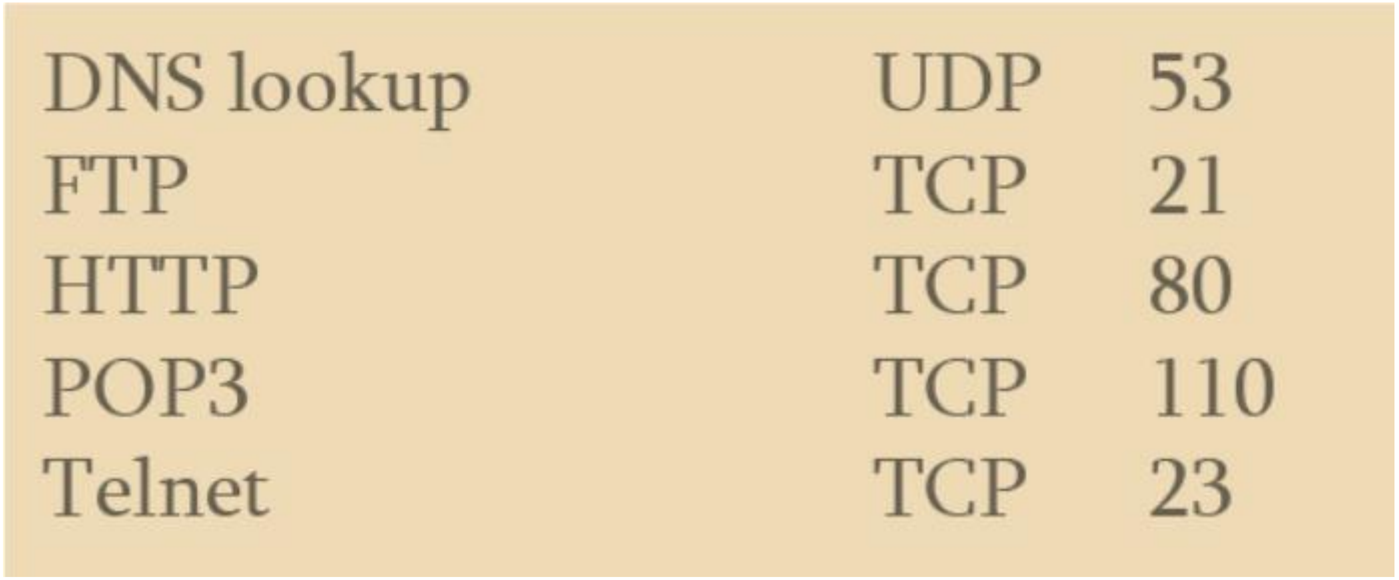
- [1] Jacobson, V. Congestion avoidance and control. ACM SIGCOMM Computer Communication Review, 18(4): 314-329, 1988.
- [2] Ha, S., Rhee I., and Xu L. CUBIC: a new TCP-friendly high-speed TCP variant. ACM SIGOPS, 2008.
- [3] Cardwell N., Cheng Y., Gunn C.S., Yeganeh S.H., and Jacobson,V. BBR congestion-based congestion control, ACM Queue, 2016.

TCP vs. UDP (I)

- Both use port numbers
  - Application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - To provide end-to-end transport
- UDP: User Datagram Protocol
  - connectionless
  - No acknowledgements
  - No retransmissions
  - Out of order, duplicates possible
- TCP: Transmission Control Protocol
  - Connection-oriented
  - Reliable byte-stream channel (in order, all arrive, no duplicates)
  - Flow control
  - bidirectional

TCP vs. UDP (II)

- TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-replay actions.
- Some common examples of TCP and UDP with their default ports:



DNS lookup	UDP	53
FTP	TCP	21
HTTP	TCP	80
POP3	TCP	110
Telnet	TCP	23

References

- [1] A.S. Tanenbaum, and D.J. Wetherall, Computer Networks, 5th Edition, Prentice Hall, 2011.
- [2] S. Ha, I. Rhee, and L. Xu, “CUBIC: a new TCP-friendly high-speed TCP variant,” ACM SIGOPS Operating Systems Review, 42(5): 64-74, 2008. (截至2021年11月18日引用2439次)
- [3] N. Cardwell, Y. Cheng, C.S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR Congestion-based congestion control – Measuring bottleneck bandwidth and round-trip propagation time,” ACM QUEUE, vol.15, no.5, pp.20-53, 2016. (截至2021年11月18日引用588次)