IF2230 Jaringan Komputer

Transport layer

End-to-End Protocols a.k.a Transport Protocols

Robithoh Annur
Andreas Bara Timur
Monterico Andrian

Problem

- How to turn this host-to-host packet delivery service into a process-to-process communication channel

Chapter Outline

- Simple Demultiplexer (UDP)
- Reliable Byte Stream (TCP)

Chapter Goal

- Understanding the demultipexing service
- Discussing simple byte stream protocol

Transport Layer - Introduction

- Provide *logical communication* between app processes running on different hosts
- Transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - receiver side: reassembles segments into messages, passes to app layer

Function of Transport Layer

- Common properties that a transport protocol can be expected to provide
  - Guarantees message delivery
  - Delivers messages in the same order they were sent
  - Delivers at most one copy of each message
  - Supports arbitrarily large messages
  - Supports synchronization between the sender and the receiver
  - Allows the receiver to apply flow control to the sender
  - Supports multiple application processes on each host

Function of Transport Layer

- Provide port number to identify application within a host.
- Chop a long stream of data bits into segments and send them off.
- Reassemble the segments at the receiver.
- Open and close a logical connection
- Provide reliable channel.
- Provide flow control.

End-to-end Protocols

- Typical limitations of the network on which transport protocol will operate
  - Drop messages
  - Reorder messages
  - Deliver duplicate copies of a given message
  - Limit messages to some finite size
  - Deliver messages after an arbitrarily long delay

End-to-end Protocols

- Challenge for Transport Protocols
  - Develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs

Transport Protocols

- The Internet offers two transport protocols available to applications
  - TCP and UDP
- TCP
  - Connection-oriented: Need handshake procedure (require to setup and tear down)
  - Reliable: No error, no data loss, and in proper order
  - Flow control: sender won't overwhelm receiver
  - Congestion control: throttle sender when network overloaded
  - does not provide: timing, minimum bandwidth guarantees
- UDP is
  - Connectionless: No handshake, just send
  - Unreliable: No guarantee that the segment will reach the receiving end.
  - does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee
  - Generally sent faster than TCP

TCP and UDP

UDP - User Datagram Protocol

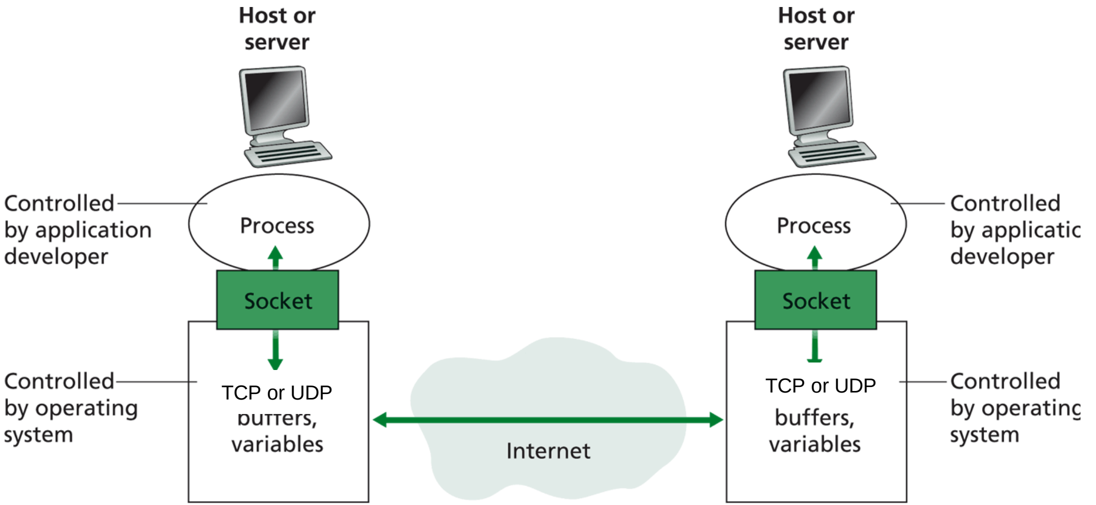
- datagram oriented
- unreliable, connectionless
- simple
- unicast and multicast
- useful only for few applications, e.g., multimedia applications
- used a lot for services
  - network management (SNMP), routing (RIP), naming (DNS), etc.

TCP - Transmission Control Protocol

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

Choice of Transport Protocol via Socket

- You have a choice to send data using either TCP or UDP, in YOUR own program.
- However, you DON'T have much of a choice (to choose between TCP or UDP) if you use "standard" program such as FTP, TFTP, browser or mail



Usage of TCP and UDP

- TCP is used by Application Layer when:
  - Data integrity is an issue
  - Data error is an issue
  - Delay is tolerable
- UDP is used by Application Layer when:
  - Speed is an issue
  - Delay is an issue
  - Lost of data is not so critical
  - Simplicity
  - Reliability of data is not done in transport layer, but in application layer.

UDP and TCP Header

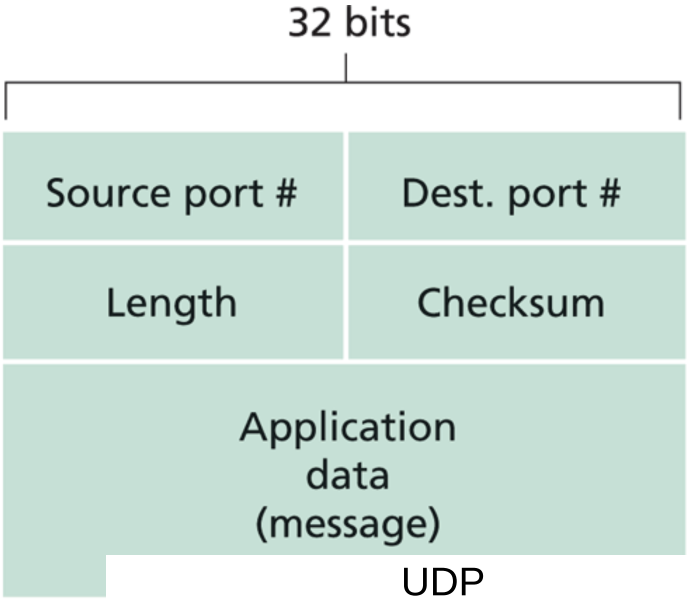


Figure 3.7 ♦ UDP segment structure

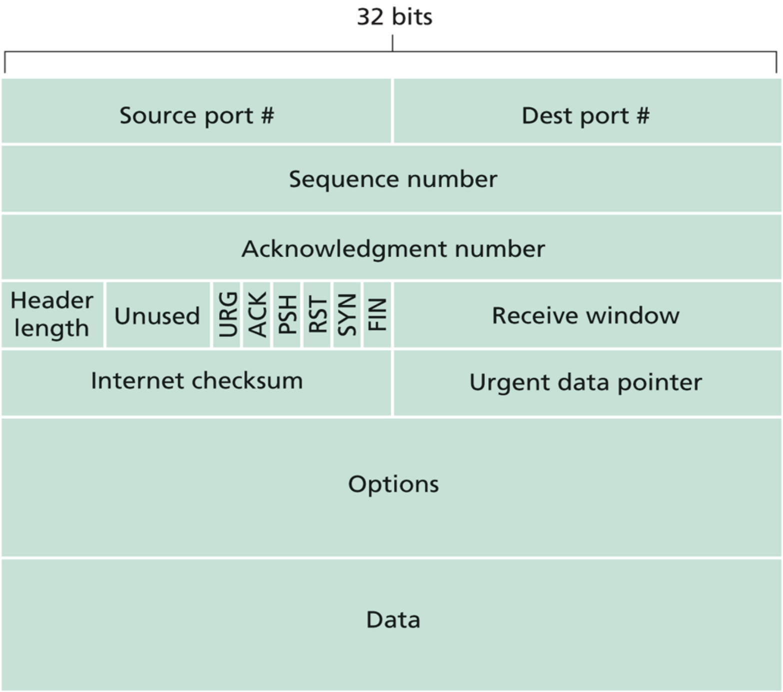


Figure 3.29 ♦ TCP segment structure

Port Numbers

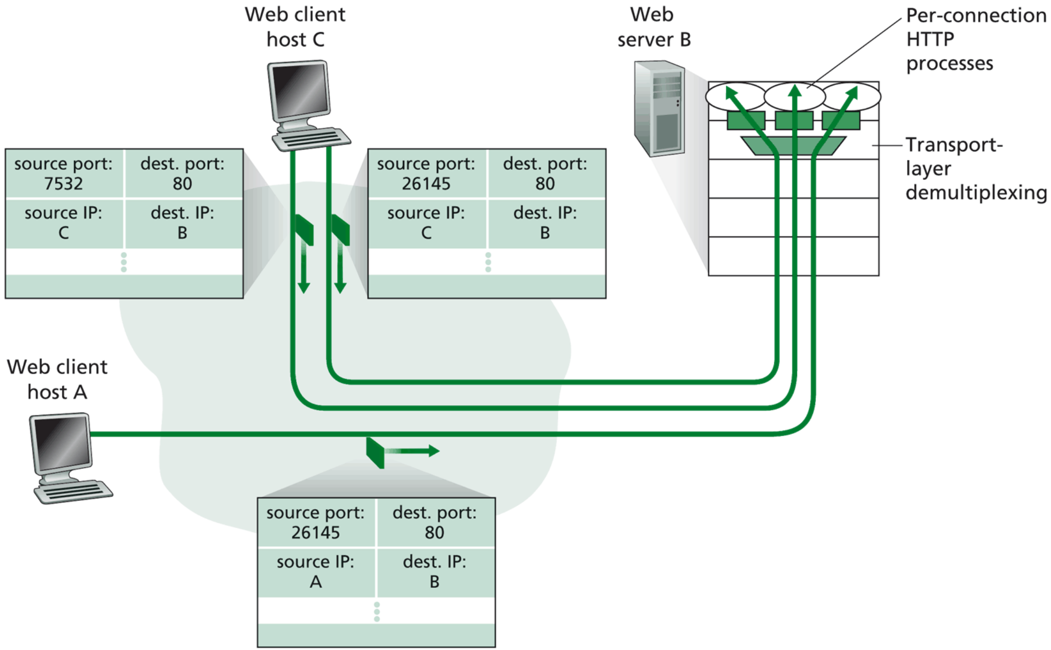


Figure 3.5 ♦ Two clients, using the same destination port number (80) to communicate with the same Web server application

- Port numbers are used to identify process/programs in both servers and clients.

UDP

Intro to UDP

- No “connection”, just send
  - No SYN, No FIN
- No reliability
  - No sequence number, no acknowledgment
- Transport layer still need to provide port number.
- Need the speed but “best effort” service, hence UDP segment may be:
  - Lost
  - Delivered out of order to application (e.g. 1,2,3,4 are sent but 2,3,1,4 are received)

Reliable Connection with UDP?

- By definition, UDP is unreliable (segment can be lost)
- Reliable connection service is a SOFTWARE implementation.
  - Just need to keep track of the bytes sent and received. (Sent and acknowledged)
  - Just need to keep the bytes in order.
  - Retransmit if the data is lost
- In real world, tftp is a “reliable” application but sending data through UDP
- The “reliable” mechanism is perform in APPLICATION layer, instead of transport layer (as in FTP with TCP).

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

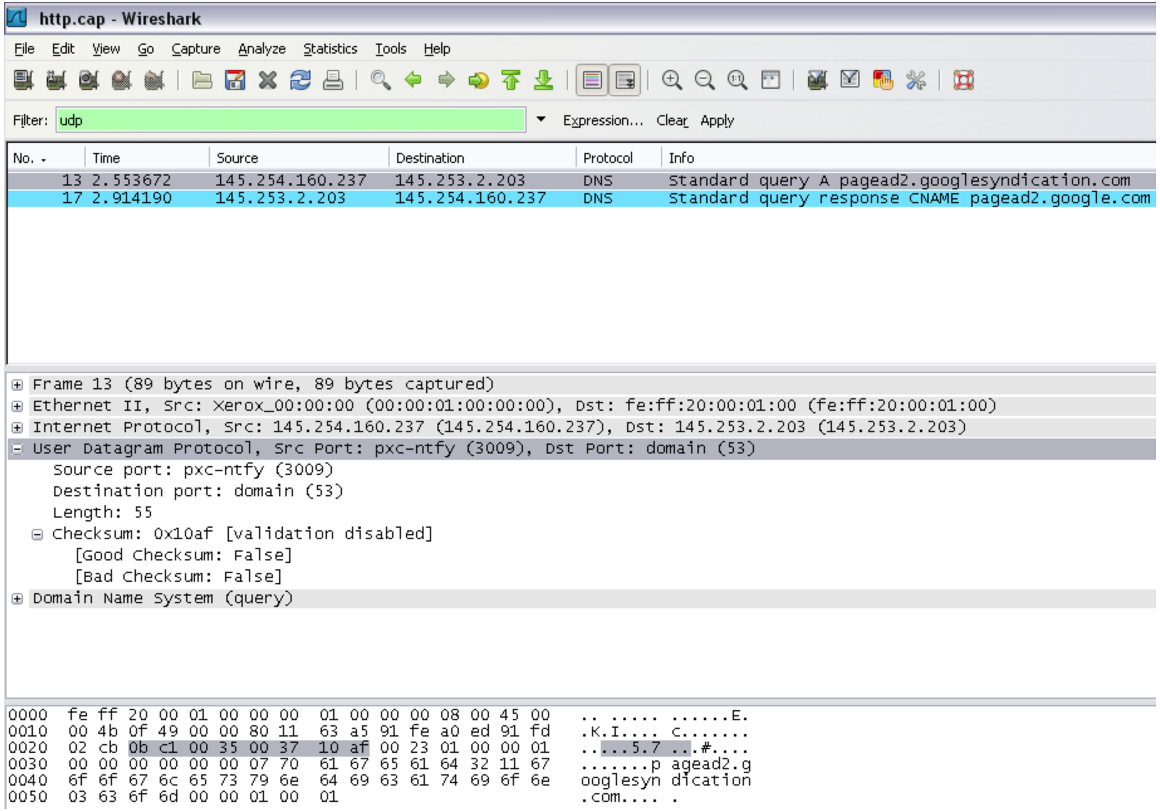
Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

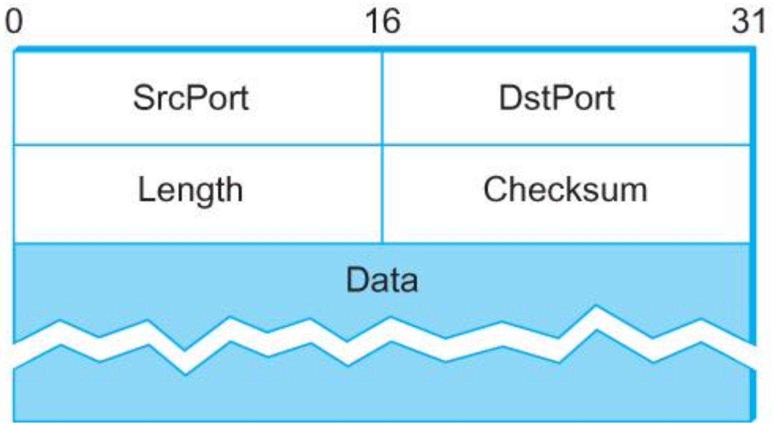
UDP - Wireshark Example



Simple Demultiplexer (UDP)

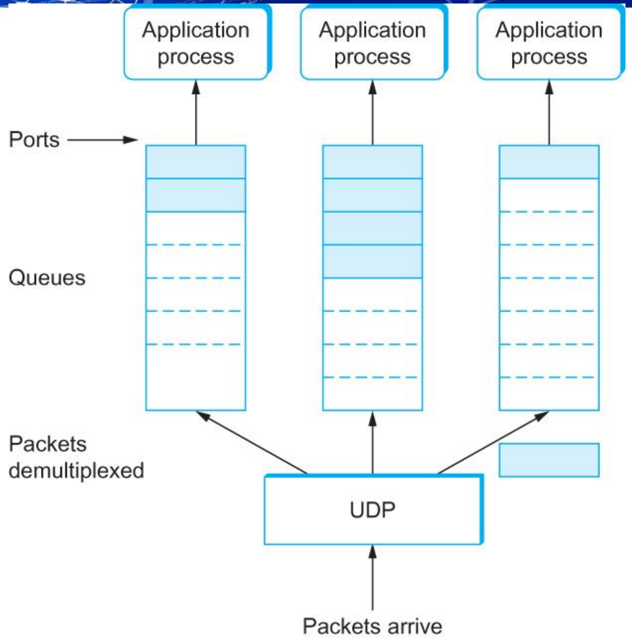
- Extends host-to-host delivery service of the underlying network into a process-to-process communication service
- Adds a level of demultiplexing which allows multiple application processes on each host to share the network

Simple Demultiplexer (UDP)



Format for UDP header (Note: length and checksum fields should be switched)

Simple Demultiplexer (UDP)



UDP Message Queue

TCP

Reliable Byte Stream (TCP)

- In contrast to UDP, Transmission Control Protocol (TCP) offers the following services
  - Reliable
  - Connection oriented
  - Byte-stream service

End-to-end Issues

- At the heart of TCP is the sliding window algorithm
- As TCP runs over the Internet rather than a point-to-point link, the following issues need to be addressed by the sliding window algorithm
  - TCP supports logical connections between processes that are running on two different computers in the Internet
  - TCP connections are likely to have widely different RTT times
  - Packets may get reordered in the Internet
- TCP needs a mechanism using which each side of a connection will learn what resources the other side is able to apply to the connection
- TCP needs a mechanism using which the sending side will learn the capacity of the network

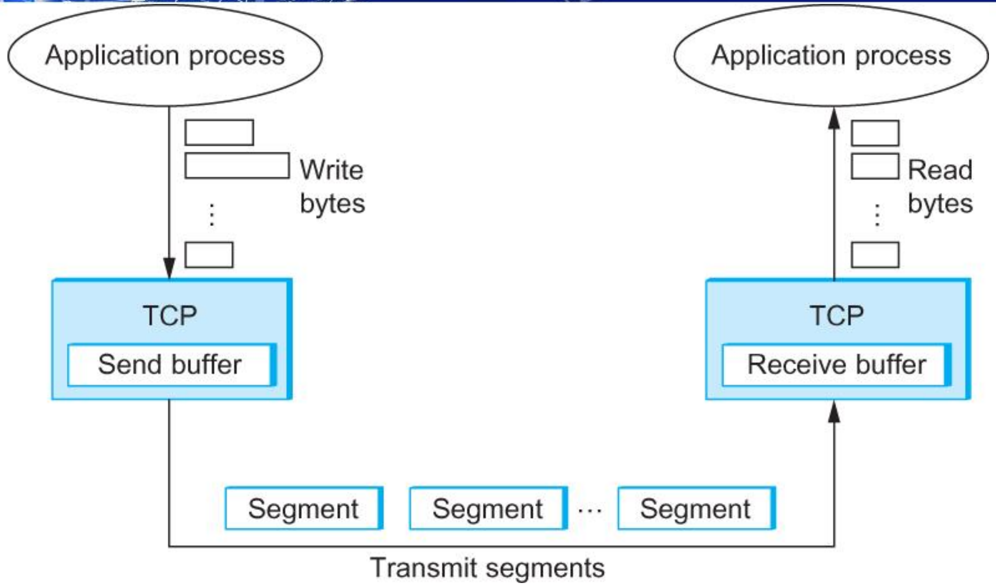
Flow control VS Congestion control

- Flow control involves preventing senders from overrunning the capacity of the receivers
- Congestion control involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded (next lecture)

TCP Segment

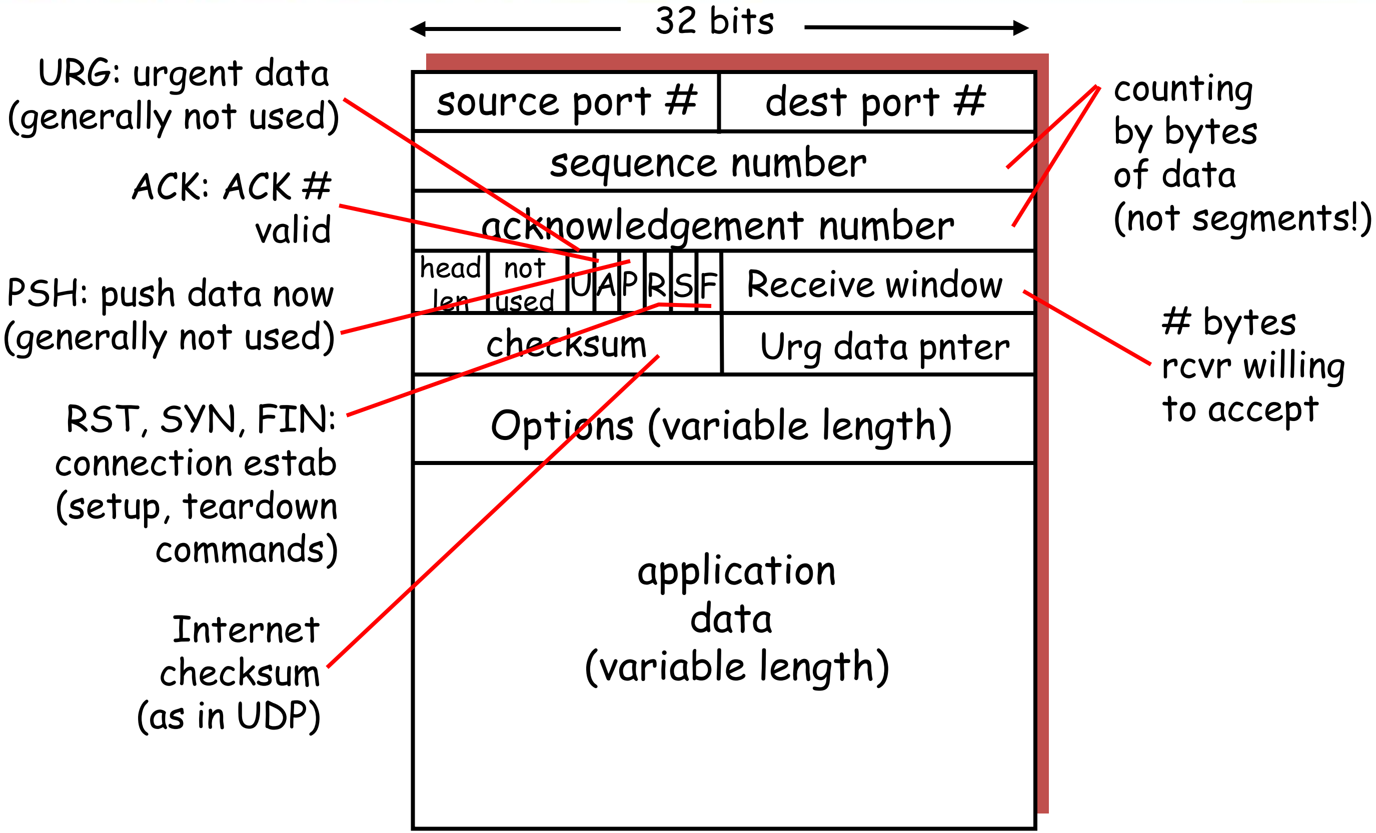
- TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.
- Although “byte stream” describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet.
- TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host.
- TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure.
- The packets exchanged between TCP peers are called *segments*.

TCP Segment



How TCP manages a byte stream.

TCP Header in Detail



TCP Header Fields

- Source port
  - This field identifies the sending port.
- Destination port
  - This field identifies the receiving port.
- Sequence number
  - The sequence number has a dual role. If the SYN flag is present then this is the initial sequence number and the first data byte is the sequence number plus 1. Otherwise if the SYN flag is not present then the first data byte is the sequence number.
- Acknowledgement number
  - If the ACK flag is set then the value of this field is the sequence number the sender expects next.

TCP Header Fields

- Data offset (Header Length)
  - This 4-bit field specifies the size of the TCP header in 32-bit words. The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes. This field gets its name from the fact that it is also the offset from the start of the TCP packet to the data.
- Window
  - The number of bytes the sender is willing to receive starting from the acknowledgement field value
- Reserved
  - 4-bit reserved field for future use and should be set to zero.

TCP Header Field

- Checksum
  - The 16-bit checksum field is used for error-checking of the header *and data*.
  - When TCP runs over IPv4, the method used to compute the checksum is defined in RFC 793:
  - *The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.*
  - In other words, all 16-bit words are summed together using one's complement (with the checksum field set to zero). The sum is then one's complemented. This final value is then inserted as the checksum field. Algorithmically speaking, this is the same as for IPv4.
  - The difference is in the data used to make the checksum. Included is a pseudo-header that mimics the IPv4 header:

TCP Header Flags

- URG: Urgent pointer is valid
  - If the bit is set, the following bytes contain an urgent message in the range:
  - SeqNo <= urgent message <= SeqNo+urgent pointer
- ACK: Acknowledgement Number is valid
- PSH: PUSH Flag
  - Notification from sender to the receiver that the receiver should pass all data that it has to the application.
  - Used to pass application layer header to application
- RST: Reset the connection
  - The flag causes the receiver to reset the connection
  - Receiver of a RST terminates the connection and indicates higher layer application about the reset
  - Normally means the closing of the network application
- SYN: Synchronize sequence numbers
  - Sent in the first packet when initiating a connection
- FIN: Sender is finished with sending
  - Used for closing a connection
  - Both sides of a connection may send a FIN

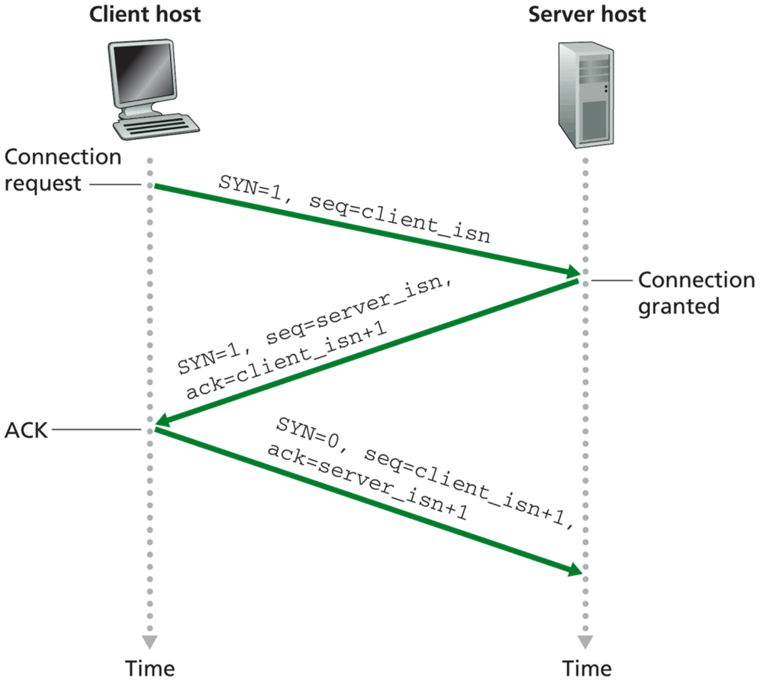
TCP: Setting UP A Connection

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- SYN flag is set
- no data

Step 2: server host receives SYN, replies with SYN-ACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYN-ACK, replies with ACK segment, which may contain data

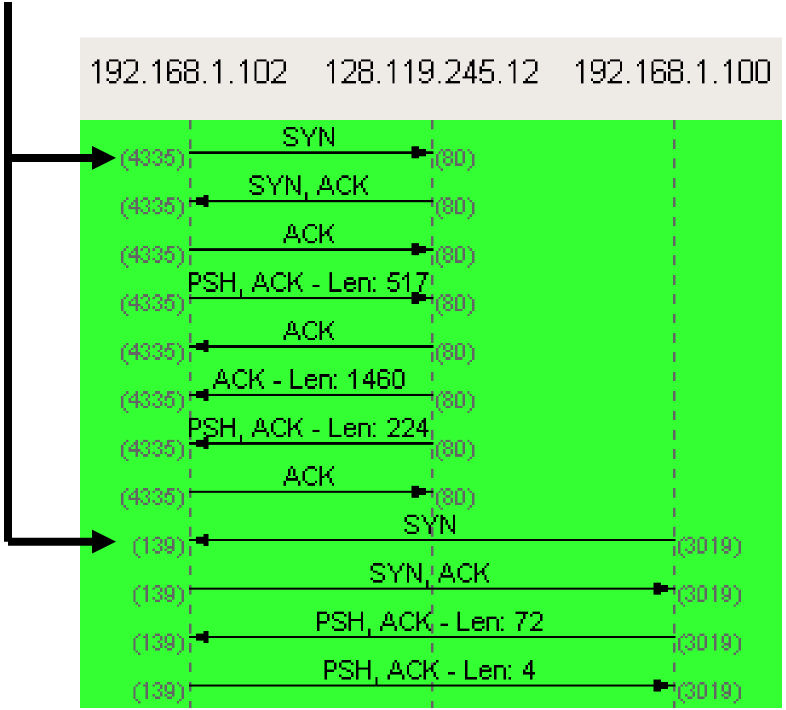


ge

Wireshark Example – TCP Flow Graph

Open Connection = A “new pair” of socket address + A “SYN”

IP Addresses

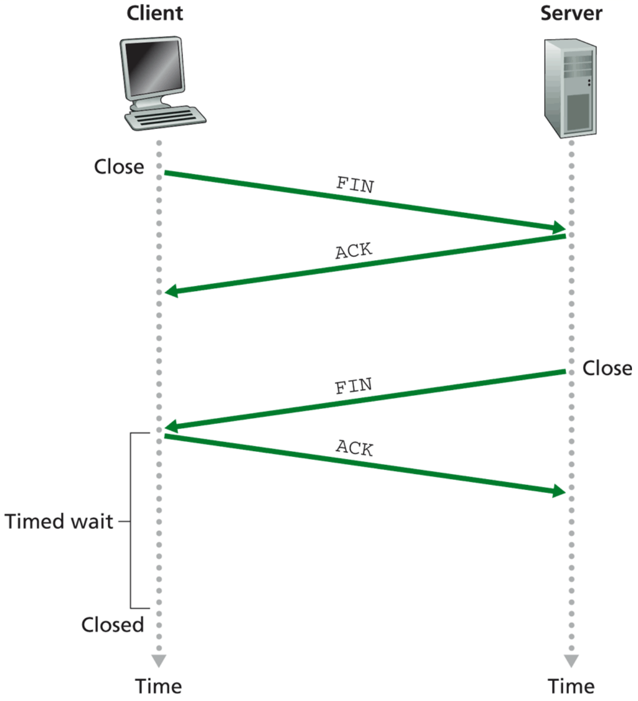


TCP: Closing a Connection

Closing a connection:

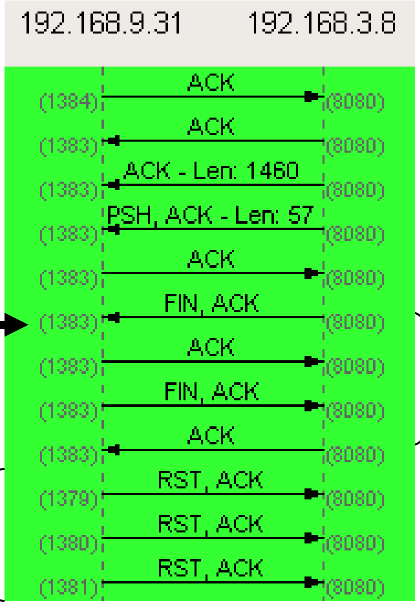
Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



Wireshark Example – TCP Flow Graph

Close Connection = “Destruction” of socket address pair + A “FIN”



Close the connection both way (beware of the direction of the arrow)

Reset Flag Closing of the program

After that, the 1529 – 8080 socket pair will not be in the flow graph (until a new SYN….)

Reliable and Connection Oriented

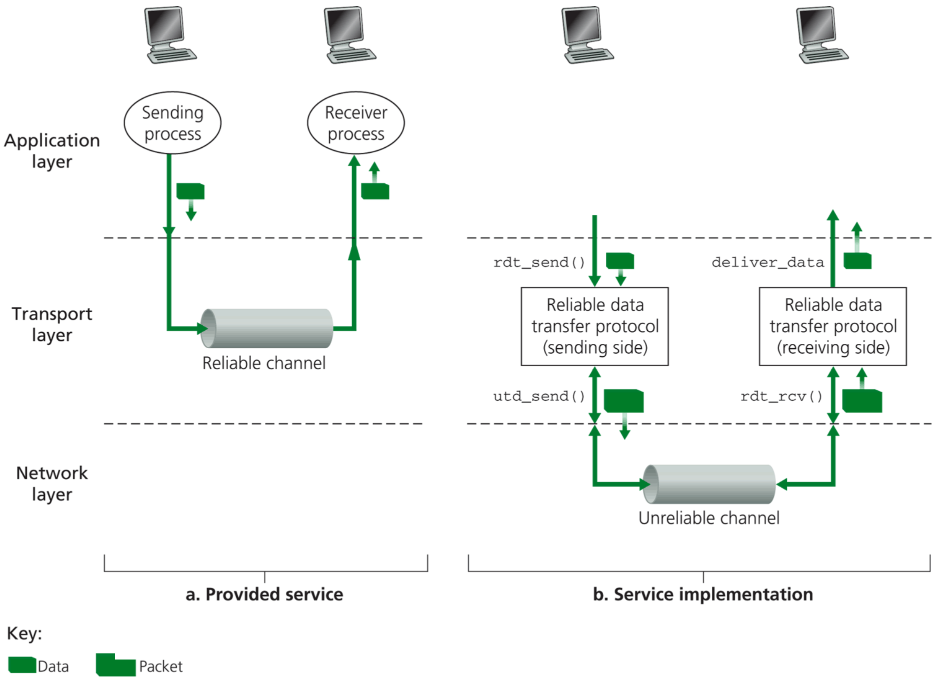


Figure 3.8 ♦ Reliable data transfer: Service model and service implementation

- Physical Layer are inherently unreliable
- It is the job of transport layer to provide "reliable" connection via software methods.
- Connection-oriented means bytes are received "in order".
  - 1,2,3,4 should be received as 1,2,3,4 and not 3,2,1,4
- TCP

How to Implement "Reliable" connection

- Through tracking the number of bytes sent
  - With sequence number
- Through tracking the number of bytes received
  - With acknowledgment number
- And various "error-correcting" mechanism
  - What if the segment is lost?
  - What if the segment is late?

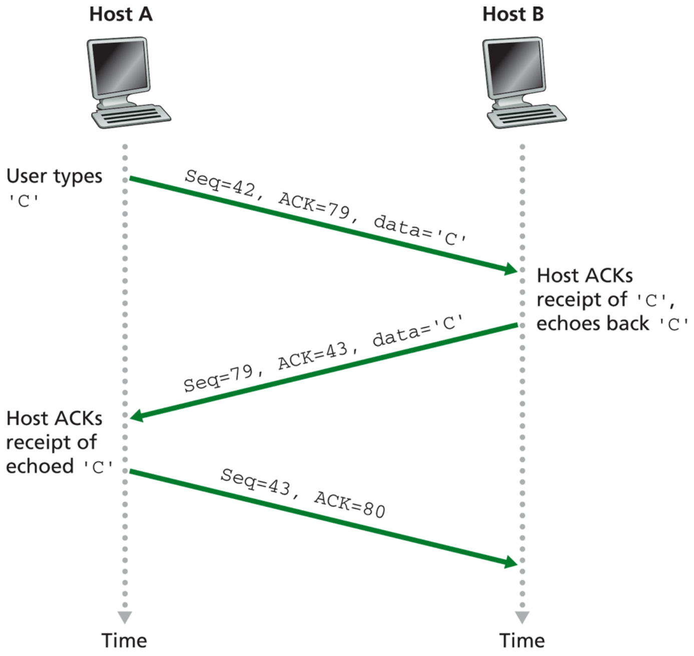
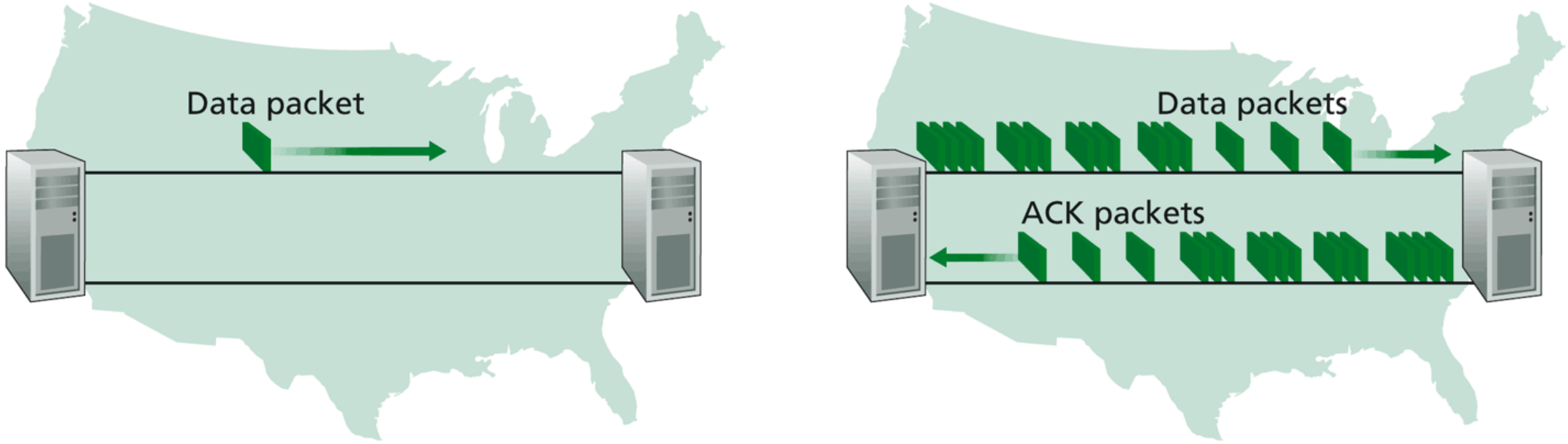


Figure 3.31 ♦ Sequence and acknowledgement numbers for a simple Telnet application over TCP

Acknowledging



a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

Figure 3.17 ♦ Stop-and-wait versus pipelined protocol

Some “transmission error” cases

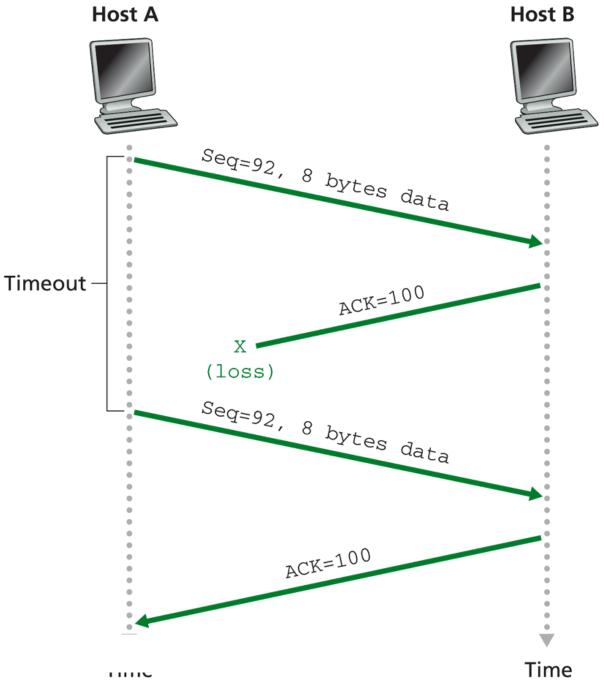


Figure 3.34 ♦ Retransmission due to a lost acknowledgment

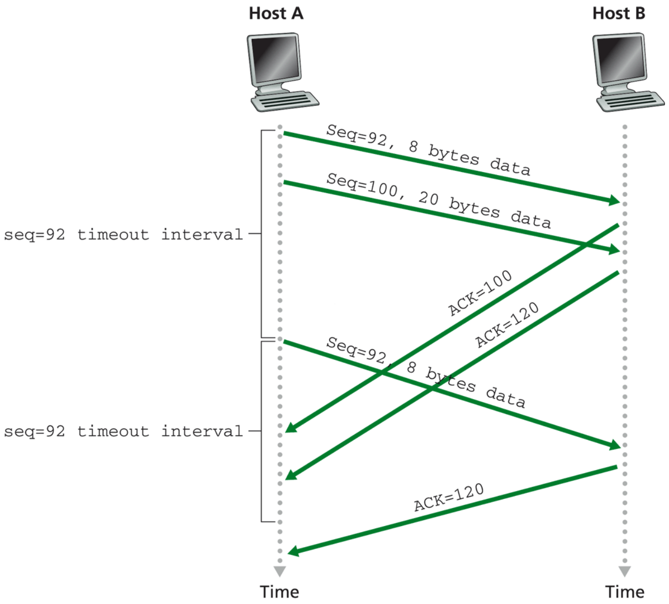


Figure 3.35 ♦ Segment 100 not retransmitted

Some "acknowledgment error" case

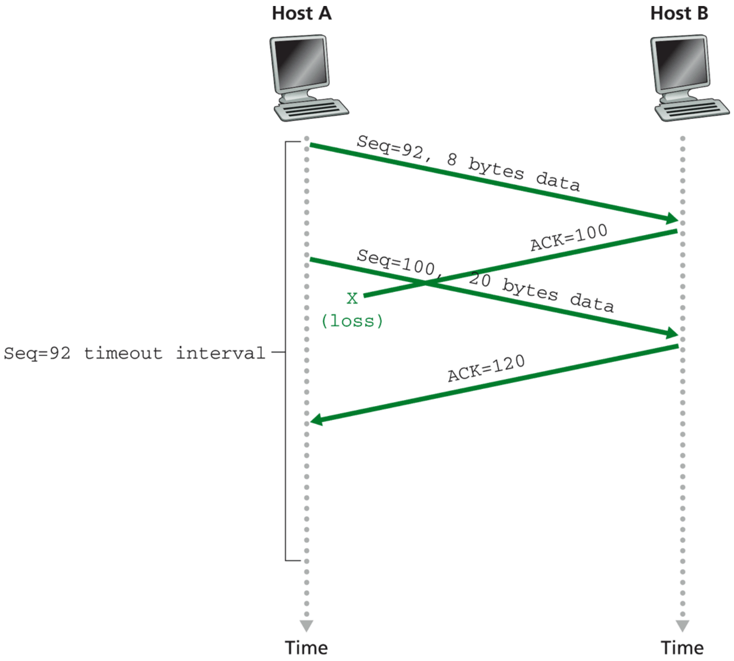
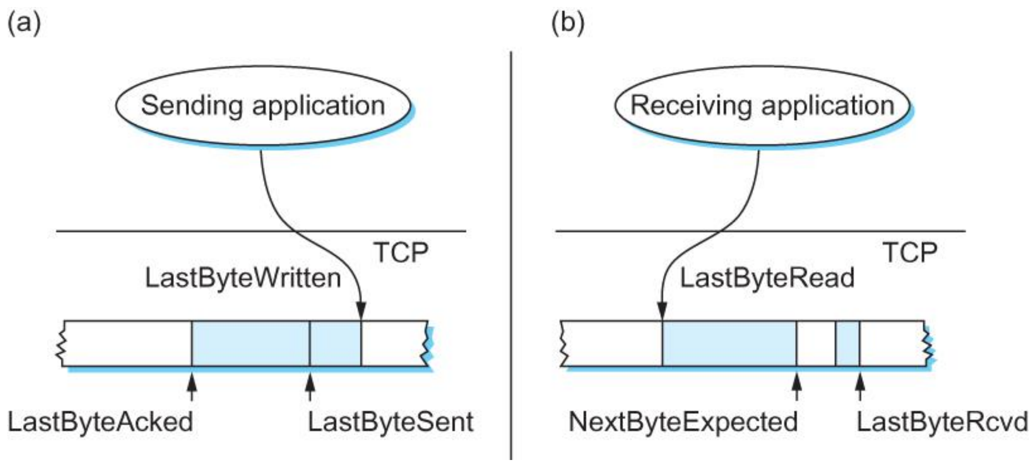


Figure 3.36 ♦ A cumulative acknowledgment avoids retransmission of the first segment.

Sliding Window

- TCP’s variant of the sliding window algorithm, which serves several purposes:
  - it guarantees the reliable delivery of data,
  - it ensures that data is delivered in order, and
  - it enforces flow control between the sender and the receiver.

Sliding Window



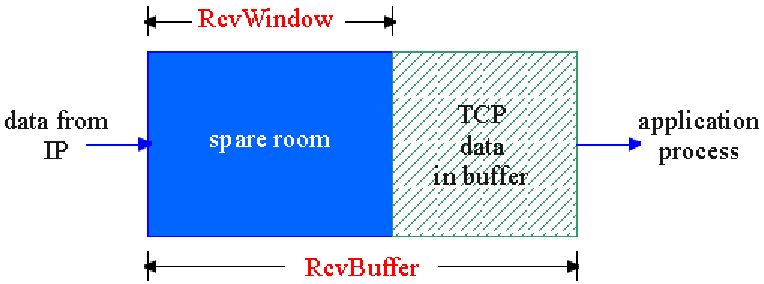
Relationship between TCP send buffer (a) and receive buffer (b).

TCP Sliding Window

- Sending Side
  - LastByteAcked ≤ LastByteSent
  - LastByteSent ≤ LastByteWritten
- Receiving Side
  - LastByteRead < NextByteExpected
  - NextByteExpected ≤ LastByteRcvd + 1

TCP Flow Control

- receive side of TCP connection has a receive buffer:



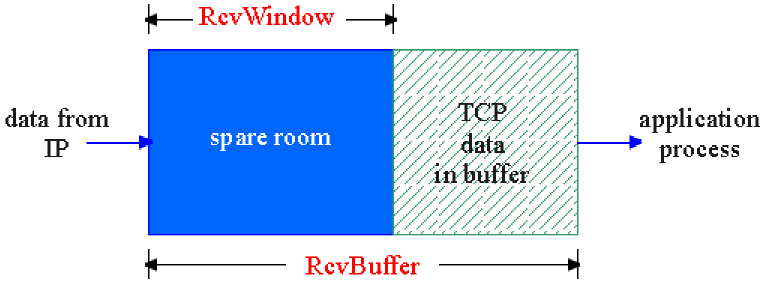
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= **`RcvWindow`**

= **`RcvBuffer-[LastByteRcvd - LastByteRead]`**

- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

TCP Flow Control

- LastByteRcvd − LastByteRead ≤ MaxRcvBuffer
- AdvertisedWindow = MaxRcvBuffer − ((NextByteExpected − 1) − LastByteRead)
- LastByteSent − LastByteAcked ≤ AdvertisedWindow
- EffectiveWindow = AdvertisedWindow − (LastByteSent − LastByteAcked)
- LastByteWritten − LastByteAcked ≤ MaxSendBuffer
- If the sending process tries to write y bytes to TCP, but

  (LastByteWritten − LastByteAcked) + y > MaxSendBuffer

  then TCP blocks the sending process and does not allow it to generate more data.

Protecting against Wraparound

- SequenceNum: 32 bits longs
- AdvertisedWindow: 16 bits long
  - TCP has satisfied the requirement of the sliding
  - window algorithm that is the sequence number
  - space be twice as big as the window size
  - $2^{32} >> 2 \times 2^{16}$

Protecting against Wraparound

- Relevance of the 32-bit sequence number space
  - The sequence number used on a given connection might wraparound
  - A byte with sequence number *x* could be sent at one time, and then at a later time a second byte with the same sequence number *x* could be sent
  - Packets cannot survive in the Internet for longer than the Maximum Segment Lifetime (**MSL)**
  - **MSL** is set to 120 sec
  - We need to make sure that the sequence number does not wrap around within a 120-second period of time
  - Depends on how fast data can be transmitted over the Internet

Protecting against Wraparound

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-12 (622 Mbps)	55 seconds
OC-48 (2.5 Gbps)	14 seconds

Time until 32-bit sequence number space wraps around.

Wrap Around Time
= (Total sequence number) / (Bandwidth)
= (2^32) / (Bandwidth)

Keeping the Pipe Full

- 16-bit AdvertisedWindow field must be big enough to allow the sender to keep the pipe full
- Clearly the receiver is free not to open the window as large as the AdvertisedWindow field allows
- If the receiver has enough buffer space
  - The window needs to be opened far enough to allow a full
  - delay × bandwidth product’s worth of data
  - Assuming an RTT of 100 ms

Keeping the Pipe Full

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-12 (622 Mbps)	7.4 MB
OC-48 (2.5 Gbps)	29.6 MB

Required window size for 100-ms RTT.

Triggering Transmission

- How does TCP decide to transmit a segment?
  - TCP supports a byte stream abstraction
  - Application programs write bytes into streams
  - It is up to TCP to decide that it has enough bytes to send a segment

Triggering Transmission

- What factors governs this decision
  - Ignore flow control: window is wide open, as would be the case when the connection starts
  - TCP has three mechanism to trigger the transmission of a segment
    - 1) TCP maintains a variable maximum segment size (MSS) and sends a segment as soon as it has collected MSS bytes from the sending process
      - MSS is usually set to the size of the largest segment TCP can send without causing local IP to fragment.
      - MSS: MTU of directly connected network – (TCP header + and IP header)
    - 2) Sending process has explicitly asked TCP to send it
      - TCP supports push operation
    - 3) When a timer fires
      - Resulting segment contains as many bytes as are currently buffered for transmission

Silly Window Syndrome

- If you think of a TCP stream as a conveyer belt with “full” containers (data segments) going in one direction and empty containers (ACKs) going in the reverse direction, then MSS-sized segments correspond to large containers and 1-byte segments correspond to very small containers.
- If the sender aggressively fills an empty container as soon as it arrives, then any small container introduced into the system remains in the system indefinitely.
- That is, it is immediately filled and emptied at each end, and never coalesced with adjacent containers to create larger containers.

Silly Window Syndrome

- **Silly Window Syndrome** is a problem that arises due to poor implementation of TCP. It degrades the TCP performance and makes the data transmission extremely inefficient. The problem is called so because:
  - It causes the sender window size to shrink to a silly value.
  - The window size shrinks to such an extent that the data being transmitted is smaller than TCP Header.

Silly Window Syndrome

- **What are the causes?**
- Sender window transmitting one byte of data repeatedly.

  Suppose only one byte of data is generated by an application . The poor implementation of TCP leads to transmit this small segment of data.Every time the application generates a byte of data, the window transmits it.
- Receiver window accepting one byte of data repeatedly.

  Suppose consider the case when the receiver is unable to process all the incoming data. In such a case, the receiver will advertise a small window size. The process continues and the window size becomes smaller and smaller. A stage arrives when it repeatedly advertises window size of 1

Silly Window Syndrome

Silly Window Syndrome

Nagle's Algorithm

- If there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data
- But how long?
- If we wait too long, then we hurt interactive applications like Telnet
- If we don't wait long enough, then we risk sending a bunch of tiny packets and falling into the *silly window* syndrome
  - The solution is to introduce a timer and to transmit when the timer expires

Nagle's Algorithm

- We could use a clock-based timer, for example one that fires every 100 ms
- Nagle introduced an elegant self-clocking solution
- Key Idea
  - As long as TCP has any data in flight, the sender will eventually receive an ACK
  - This ACK can be treated like a timer firing, triggering the transmission of more data

Nagle's Algorithm

When the application produces data to send

    if both the available data and the window ≥ MSS

        send a full segment

    else

        if there is unACKed data in flight

            buffer the new data until an ACK arrives

        else

            send all the new data now

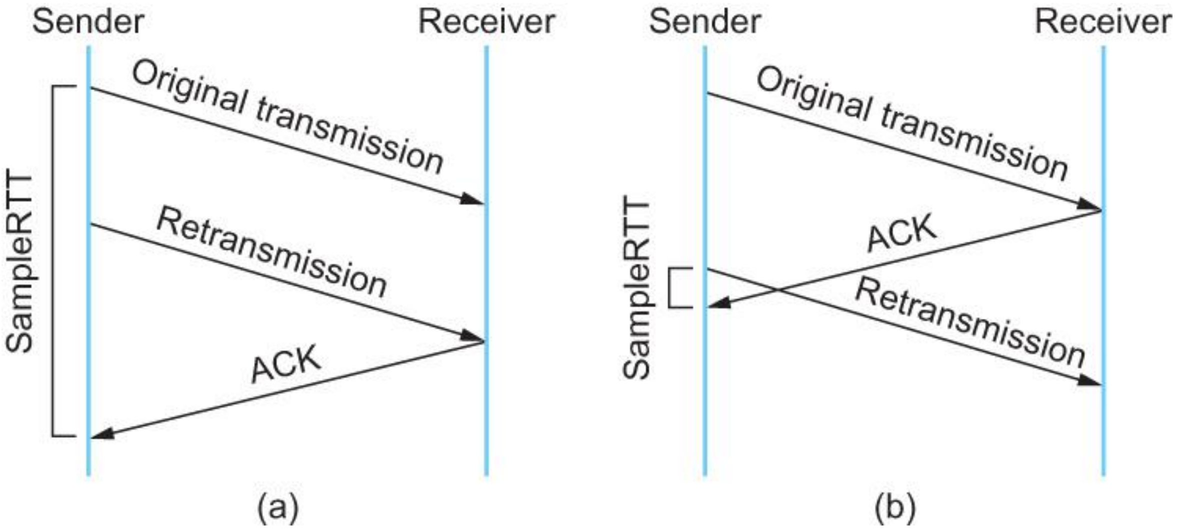
Adaptive Retransmission

- Original Algorithm
  - Measure **SampleRTT** for each segment/ ACK pair
  - Compute weighted average of RTT
    - **EstRTT** = α x **EstRTT** + (1 - α)x **SampleRTT**
  - α between 0.8 and 0.9
  - Set timeout based on **EstRTT**
    - **TimeOut** = 2 x **EstRTT**

Original Algorithm

- Problem
  - ACK does not really acknowledge a transmission
    - It actually acknowledges the receipt of data
  - When a segment is retransmitted and then an ACK arrives at the sender
    - It is impossible to decide if this ACK should be associated with the first or the second transmission for calculating RTTs

Karn/Partridge Algorithm



Associating the ACK with (a) original transmission versus (b) retransmission

Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission
- Karn-Partridge algorithm was an improvement over the original approach, but it does not eliminate congestion
- We need to understand how timeout is related to congestion
  - If you timeout too soon, you may unnecessarily retransmit a segment which adds load to the network
- Main problem with the original computation is that it does not take variance of Sample RTTs into consideration.
- If the variance among Sample RTTs is small
  - Then the Estimated RTT can be better trusted
  - There is no need to multiply this by 2 to compute the timeout
- On the other hand, a large variance in the samples suggest that timeout value should not be tightly coupled to the Estimated RTT
- Jacobson/Karels proposed a new scheme for TCP retransmission

Jacobson/Karels Algorithm

- Difference = SampleRTT − EstimatedRTT
- EstimatedRTT = EstimatedRTT + (× Difference)
- Deviation = Deviation + (|Difference| − Deviation)
- TimeOut = μ × EstimatedRTT + × Deviation
  - where based on experience, μ is typically set to 1 and is set to 4. Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the deviation term to dominate the calculation.

Summary

- We have discussed how to convert host-to-host packet delivery service to process-to-process communication channel.
- We have discussed UDP
- We have discussed TCP