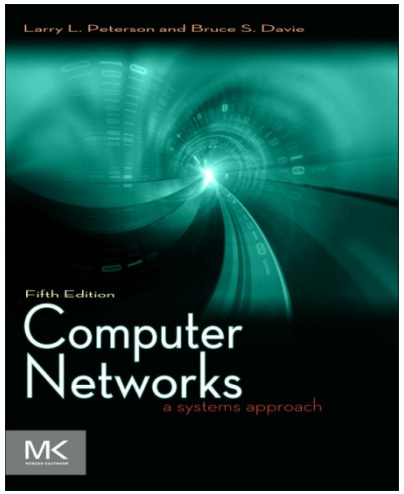


Chapter 5

End-to-End Protocols

Note: some slides from Princeton U.

Problem

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

Chapter Outline

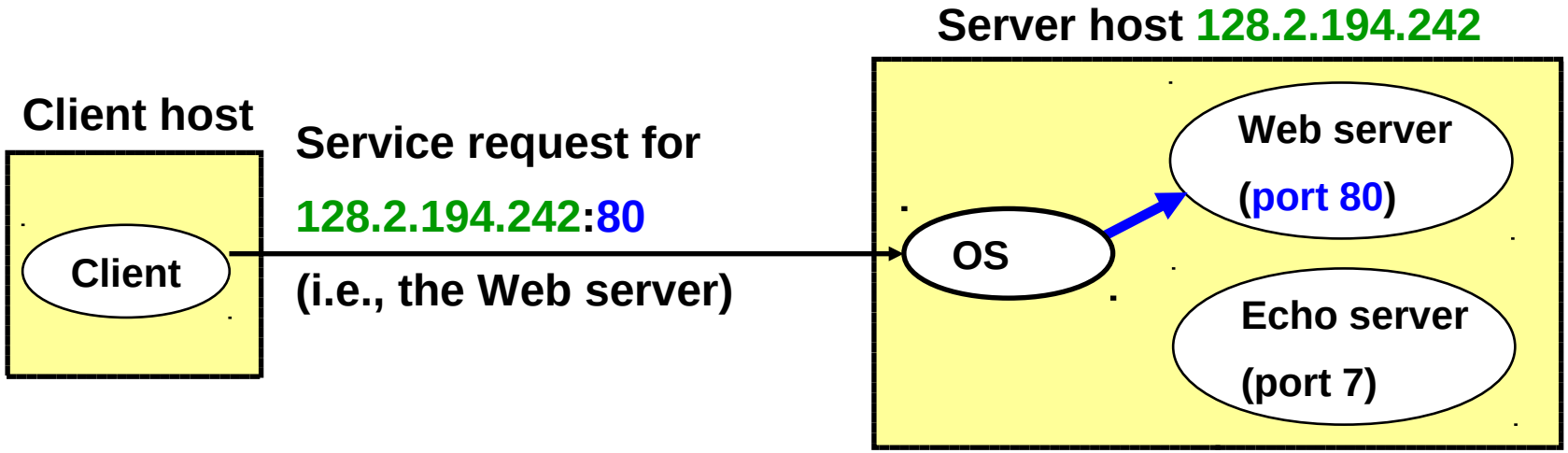
- Simple Demultiplexer (UDP)
- Reliable Byte Stream (TCP)

Chapter Objectives

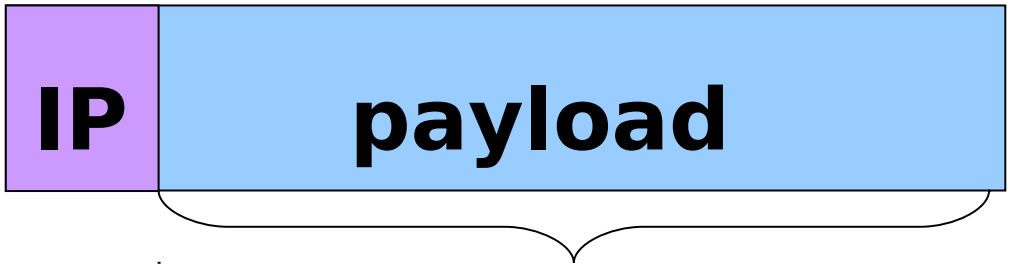
1. Analyze the UDP transport protocol. (CO: 5)
2. Explain the operation of the TCP protocol. (CO: 5)
3. Compare datagram and connection-oriented transport protocols. (CO: 5)

Transport Layer Features

- **Demultiplexing: port numbers**



- **Error detection: checksums**



End-to-end Protocols

- 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

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

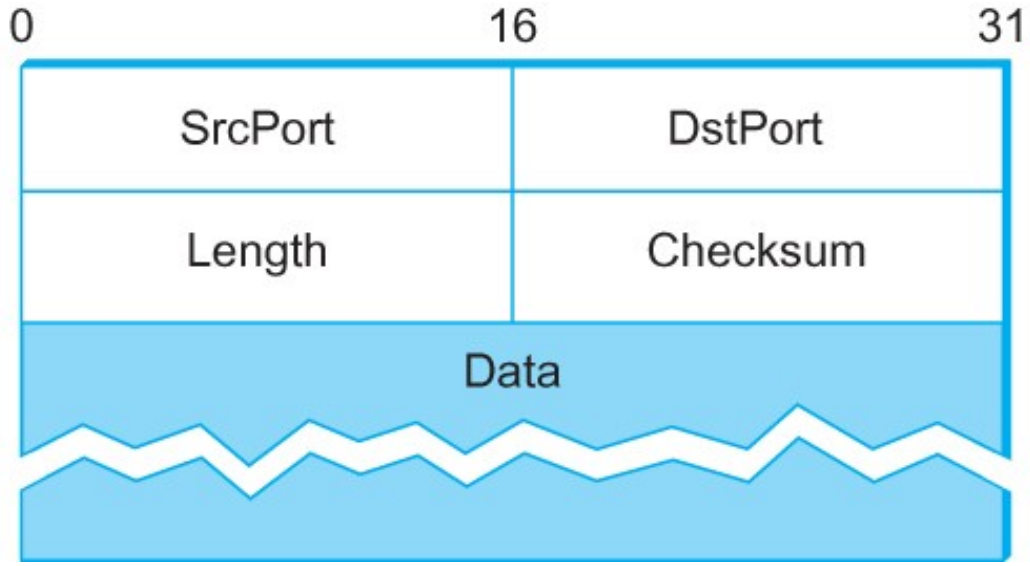
IP-based Transport Layer Protocols

- **UDP (User Datagram Protocol)**
- **TCP (Transmission Control Protocol**
- RTP (Real-time Transport Protocol), for VoIP
- SCTP (Stream Control Transmission Protocol)
- QUIC (Quick UDP Internet Connections), by Google in 2013 – Chrome browser, intended to work under HTTP, on top of UDP
  - Last two superior to TCP but not widely deployed)

UDP – User Datagram Protocol

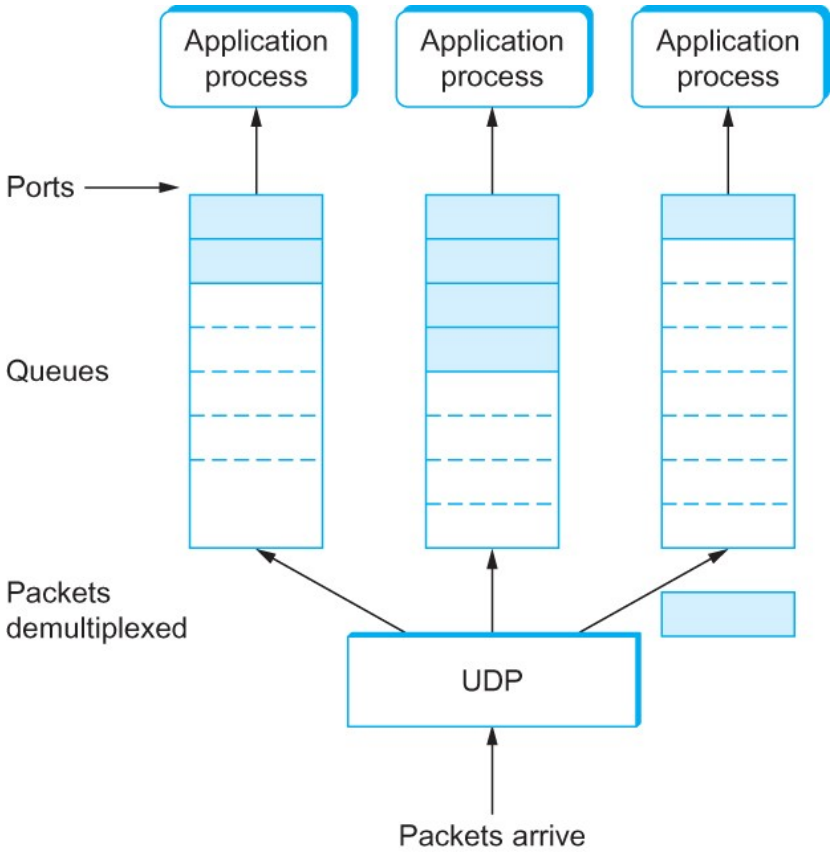
- Extends host-to-host delivery service of the underlying network into a process-to-process communication service
- Adds a level of demultiplexing which allows multiple application processes on each host to share the network
  - demultiplexing key: *port numbers* to identify source process and the destination process
- No flow control, no error control, just a *best effort* datagram service over IP. (IP is also best effort)

Simple Demultiplexer (UDP)



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

Simple Demultiplexer (UDP)



UDP Message Queue

UDP – Port Numbers

- Port number: 16 bits, i.e. 2^{16}=65536 possible ports
  - Well known port numbers sparsely between 0 and 1023: identify “standard” or widely used network services, such as SSH (22), FTP (21), SMTP (25) – for both TCP and UDP
    - Require “root/admin” permission to use as a source port# by a process (address “binding “)
  - Port numbers ≥ 1024 can be used by user programs
    - Some port numbers between 1024 and 49151 are “registered” to commonly used applications, such as 5900 (VNC remote buffer)

UDP – Port Numbers


```
tcpmux          1/tcp                       # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp                      # message send protocol
msp             18/udp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                      # SSH Remote Login Protocol
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
```

/etc/services on Unix & Linux lists well known port numbers and protocols

Alternative to Well-known Ports

- **Port Mapper** service with its own well-known port:
  - Accepts requests from clients to map a service name to a server-local port#:
    - Client request: “what is port# for the Git Version Control System ?”
    - Server reply: “It is 9418 and runs on TCP”
    - Client then connects on TCP port 9418
  - makes it easy to change the port associated with different services over time and for each host to use a different port for the same service.

Why Would Anyone Use UDP?

- Fine control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, parameters, sequence #s, etc.
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only eight-bytes long
- Build other transport protocols on top of UDP. E.g. RTP, QUIC

Popular Applications That Use UDP

- Simple query protocols like DNS
  - Overhead of connection establishment is overkill
  - Easier to have the application retransmit if needed

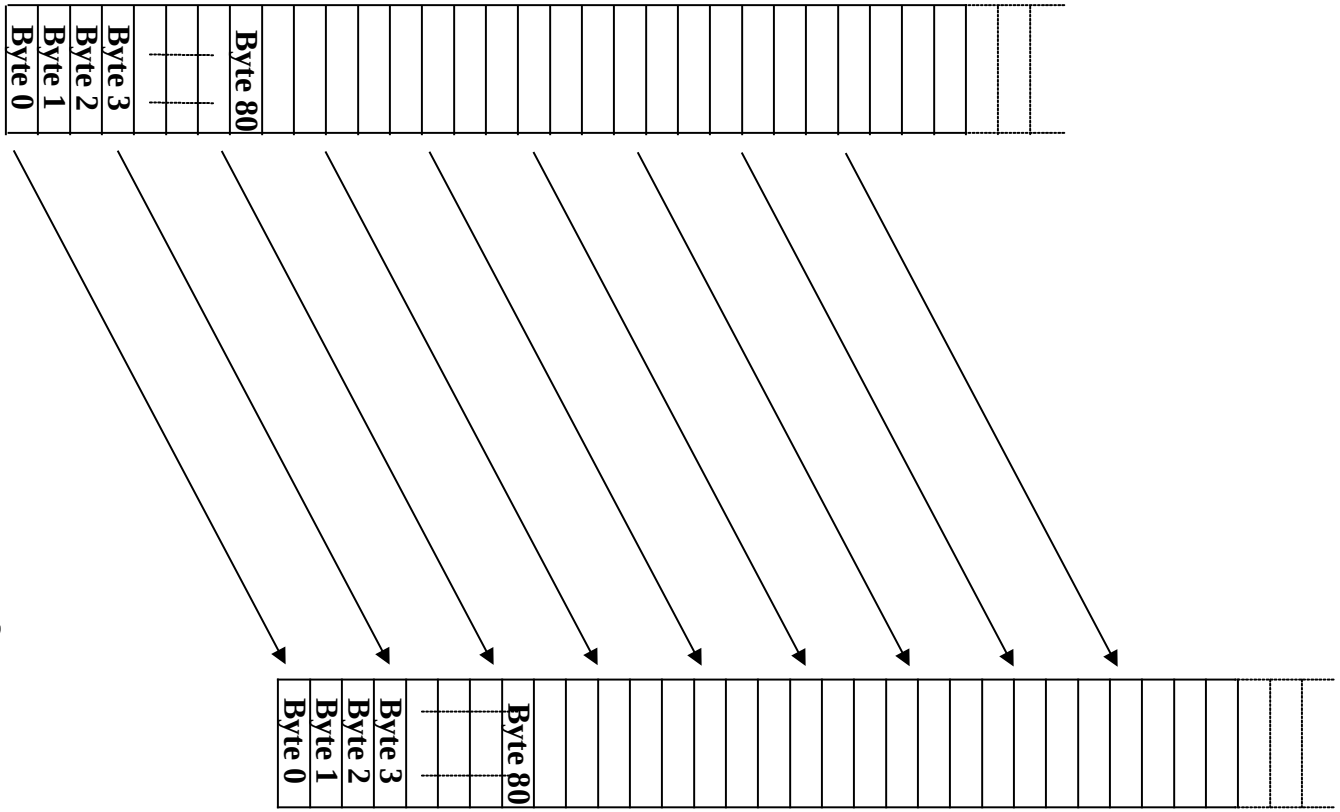
- Multimedia streaming
  - Retransmitting lost/corrupted packets is not worthwhile
  - By the time the packet is retransmitted, it’s too late
  - E.g., telephone calls, video conferencing, gaming

Reliable Byte Stream (TCP)

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

TCP “Stream of Bytes” Service

Host A



Host B

19

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
  - TCP congestion control discussed in Chapter 5

End-to-end Issues

- At the heart of TCP is the sliding window algorithm (discussed in Chapter 2)
- 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

End-to-end Issues

- 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

TCP Segment

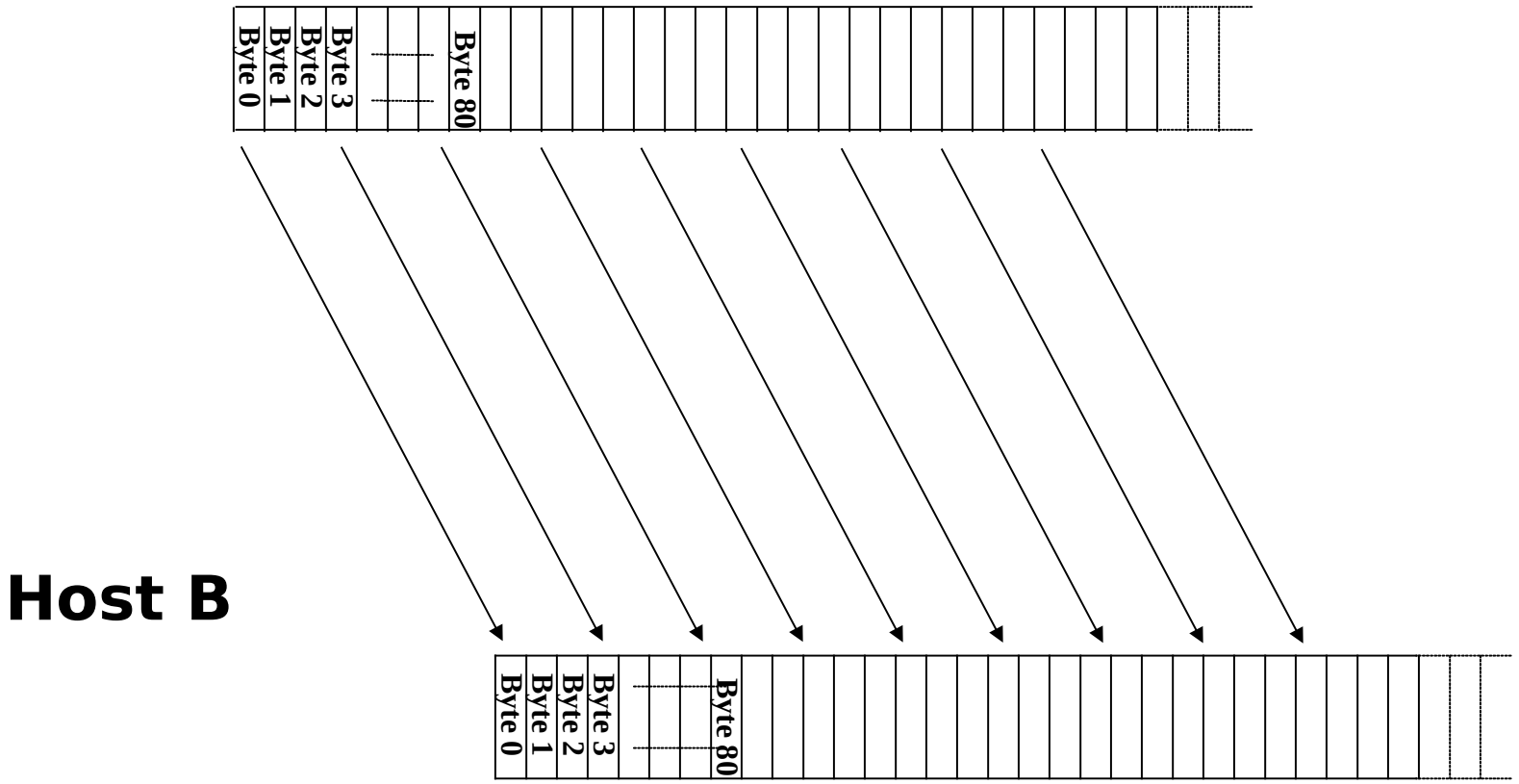
- TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.

- Although “byte stream” describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet.

TCP Segment

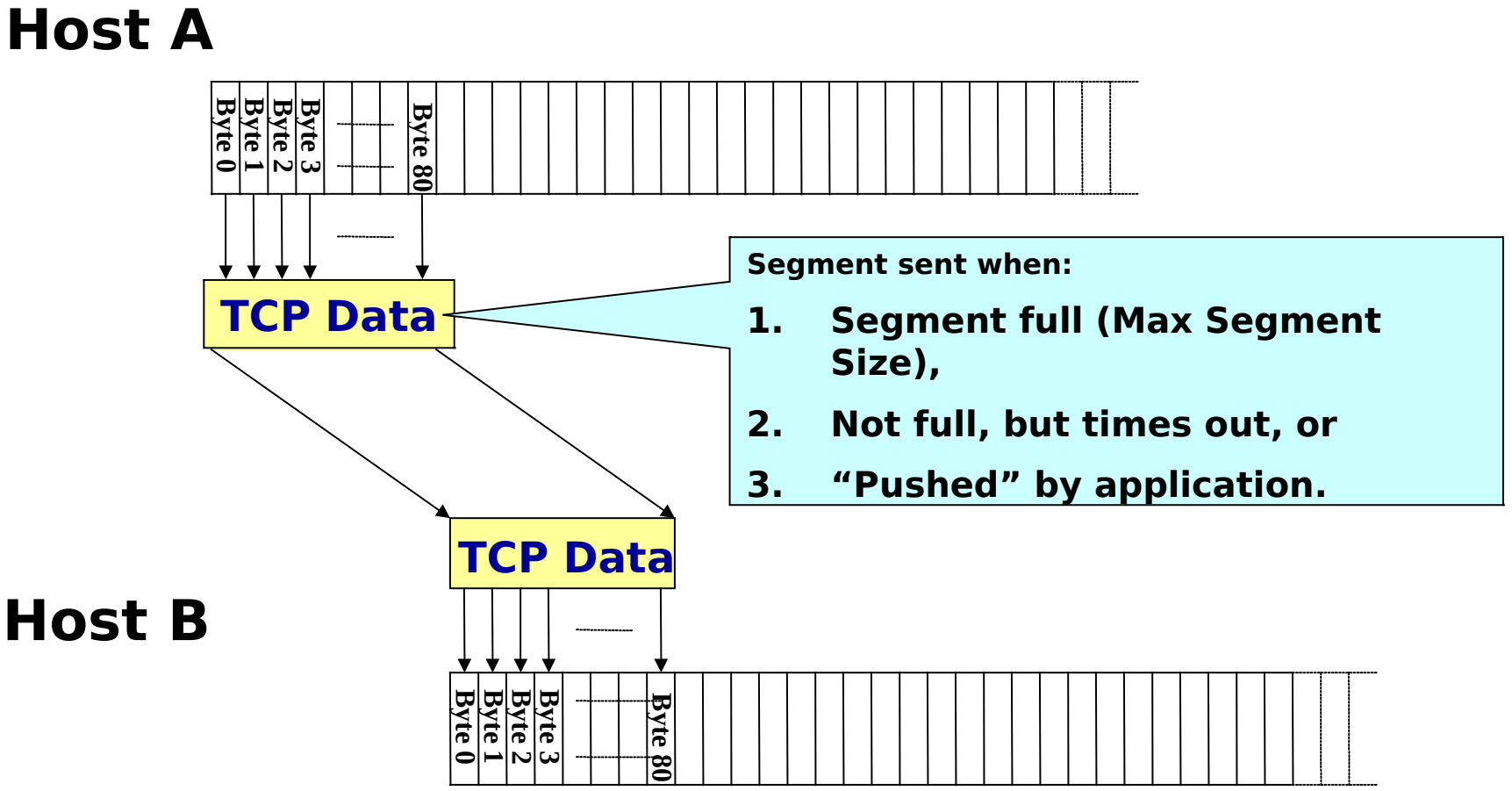
- TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host.
- TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure.
- The packets exchanged between TCP peers are called *segments*.

TCP “Stream of Bytes” Service

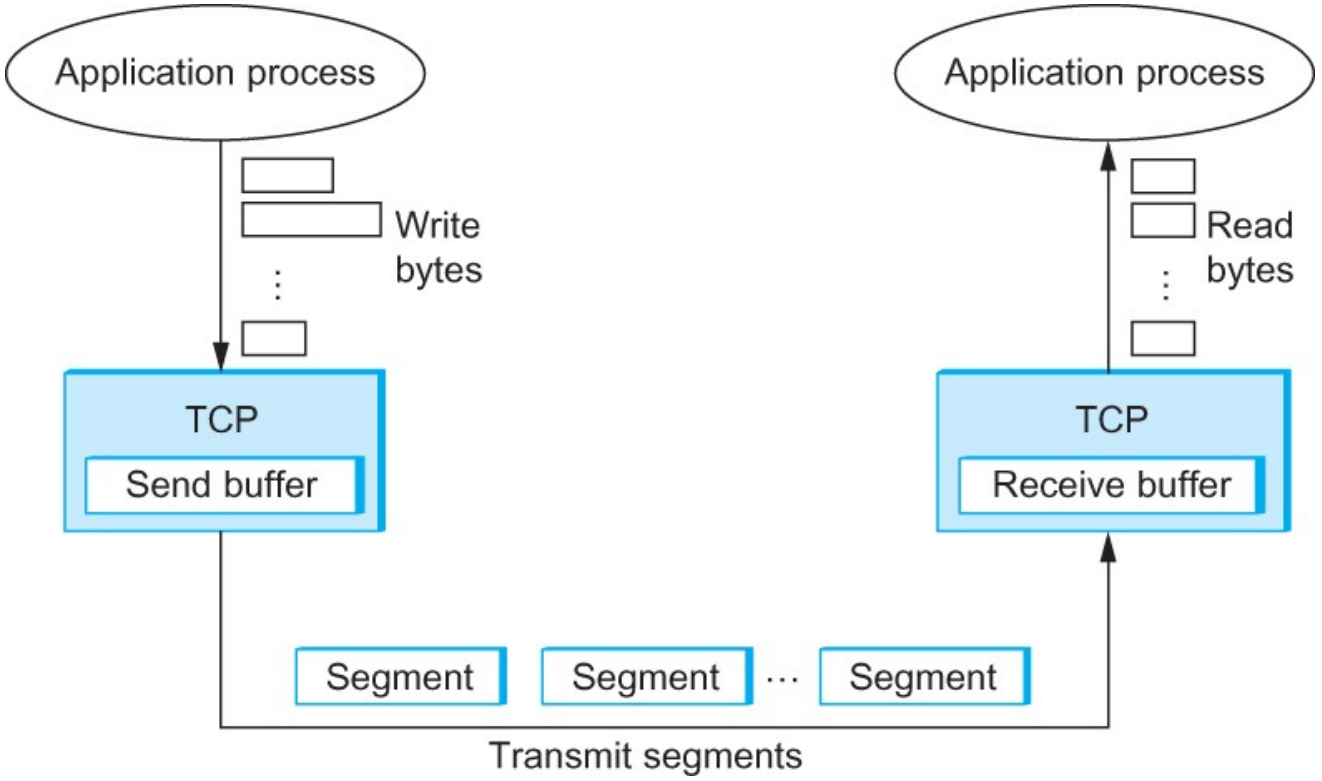


25

…Emulated Using TCP "Segments"

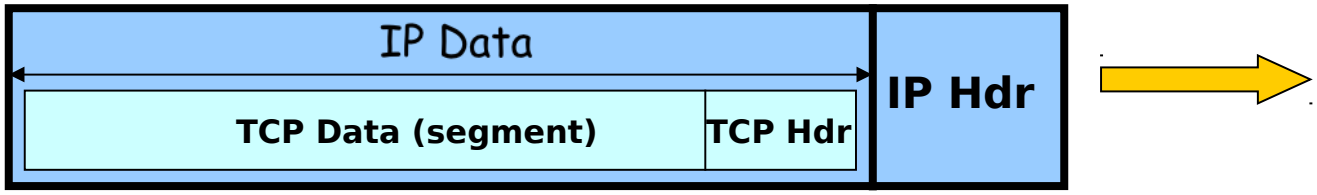


TCP Segment



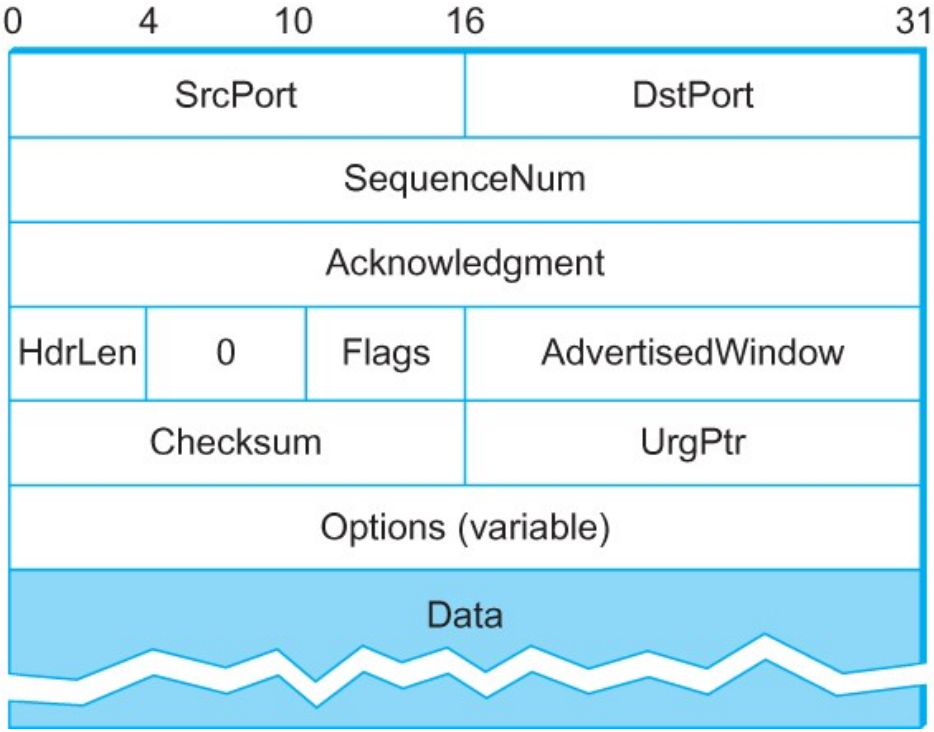
How TCP manages a byte stream.

TCP Segment



- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream

TCP Header



TCP Header Format

TCP Demux Key

The SrcPort and DstPort fields identify the source and destination ports, respectively, just as in UDP.

TCP's demux key is given by the 4-tuple:

(SrcPort, SrcIPAddr, DstPort, DstIPAddr)

TCP Header

- The SrcPort and DstPort fields identify the source and destination ports, respectively.
- The Acknowledgment, SequenceNum, and AdvertisedWindow fields are all involved in TCP's sliding window algorithm.
- Because TCP is a byte-oriented protocol, each byte of data has a sequence number; the SequenceNum field contains the sequence number for the first byte of data carried in that segment.
- The Acknowledgment and AdvertisedWindow fields carry information about the flow of data going in the other direction.

TCP Header

- The 6-bit Flags field is used to relay control information between TCP peers.
- The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK.
- The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively.
- The ACK flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it.

TCP Header

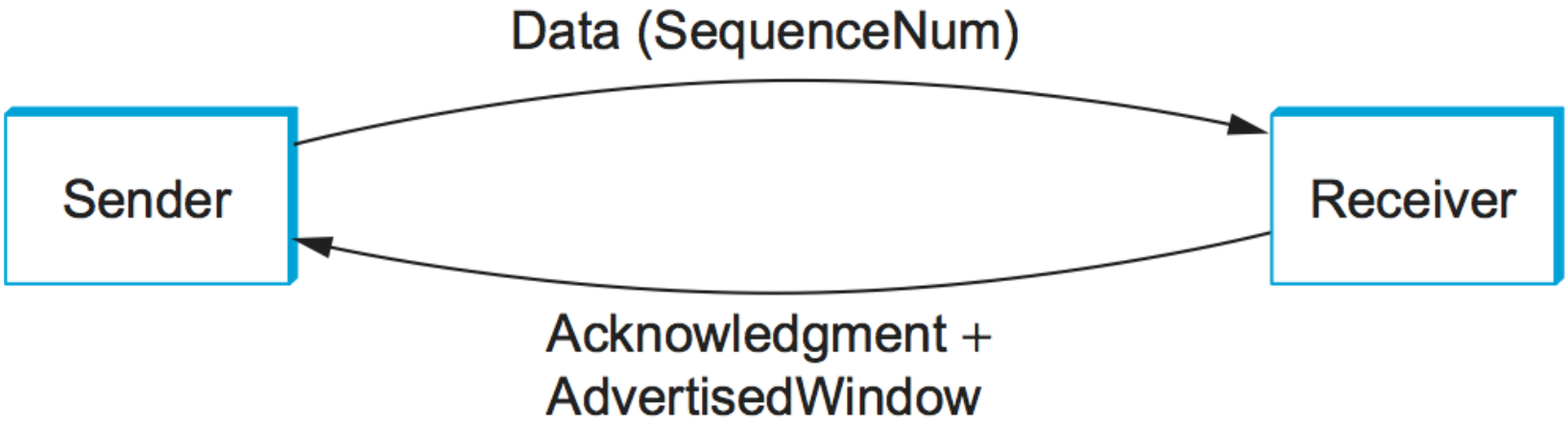
- The URG flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the nonurgent data contained in this segment begins.
- The urgent data is contained at the front of the segment body, up to and including a value of UrgPtr bytes into the segment.
- The PUSH flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact.
- Finally, the RESET flag signifies that the receiver has become confused

TCP Header

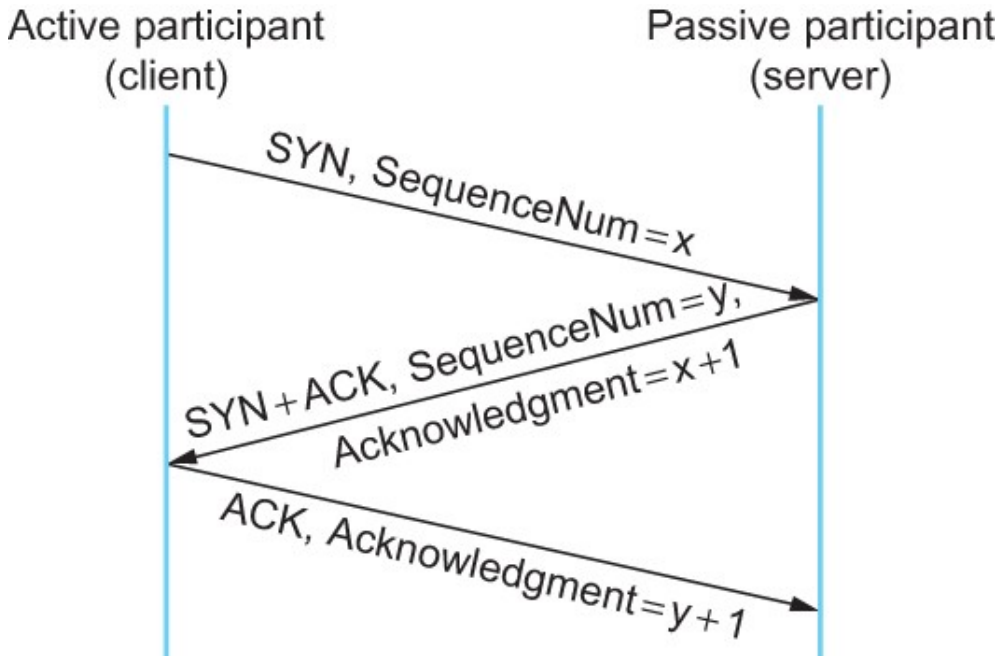
- Finally, the RESET flag signifies that the receiver has become confused, it received a segment it did not expect to receive—and so wants to abort the connection.
- Finally, the Checksum field is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the pseudoheader, which is made up of the source address, destination address, and length fields from the IP header.

TCP Data Flow

- In flow control in both directions, independent of each other:

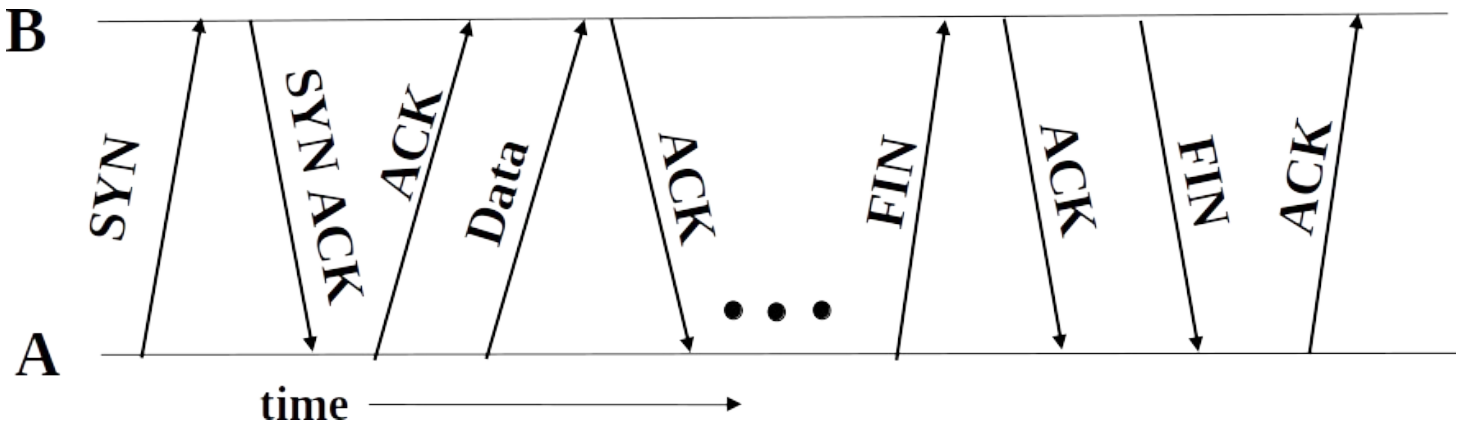


Connection Establishment/Termination in TCP



Timeline for three-way handshake algorithm

TCP Connection Tear-down

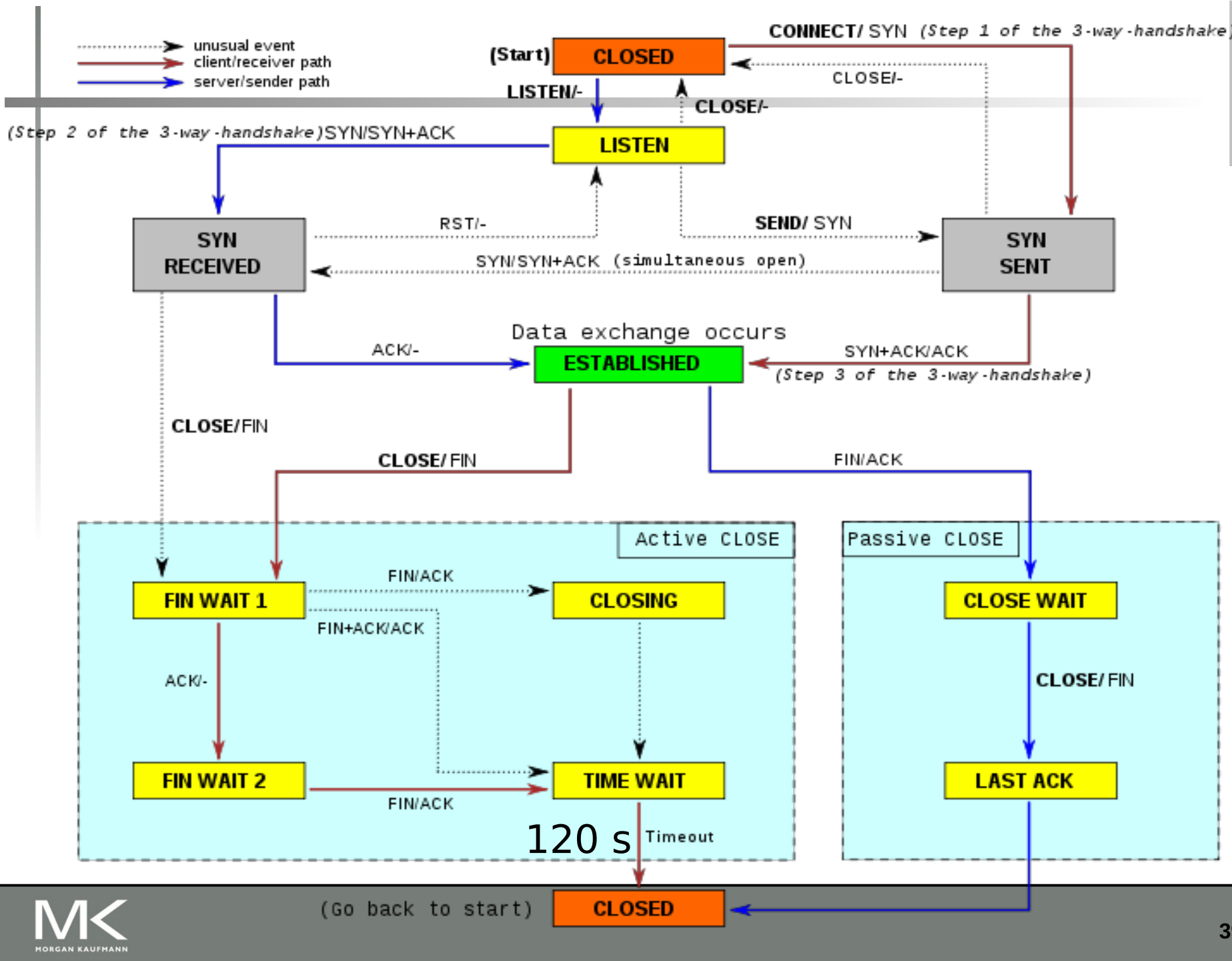


- Closing (each end of) the connection
  - Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge
  - Reset (RST) to close and not receive remaining bytes

Sending/Receiving the FIN Packet

- Sending a FIN: close()
  - Process is done sending data via the socket
  - Process invokes “close()” to close the socket
  - Once TCP has sent all of the outstanding bytes…
  - … then TCP sends a FIN
- Receiving a FIN: EOF
  - Process is reading data from the socket
  - Eventually, the attempt to read returns an EOF

Next slide: TCP protocol state machine


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
SYN RECEIVED
RST/-
SEND/ SYN
SYN/SYN+ACK (simultaneous open)
SYN SENT
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/ FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
FIN+ACK/ACK
CLOSE WAIT
ACK/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
FIN/ACK
LAST ACK
120 s
Timeout
(Go back to start)
CLOSED

Sliding Window Revisited

- TCP's variant of the sliding window algorithm, which serves several purposes:

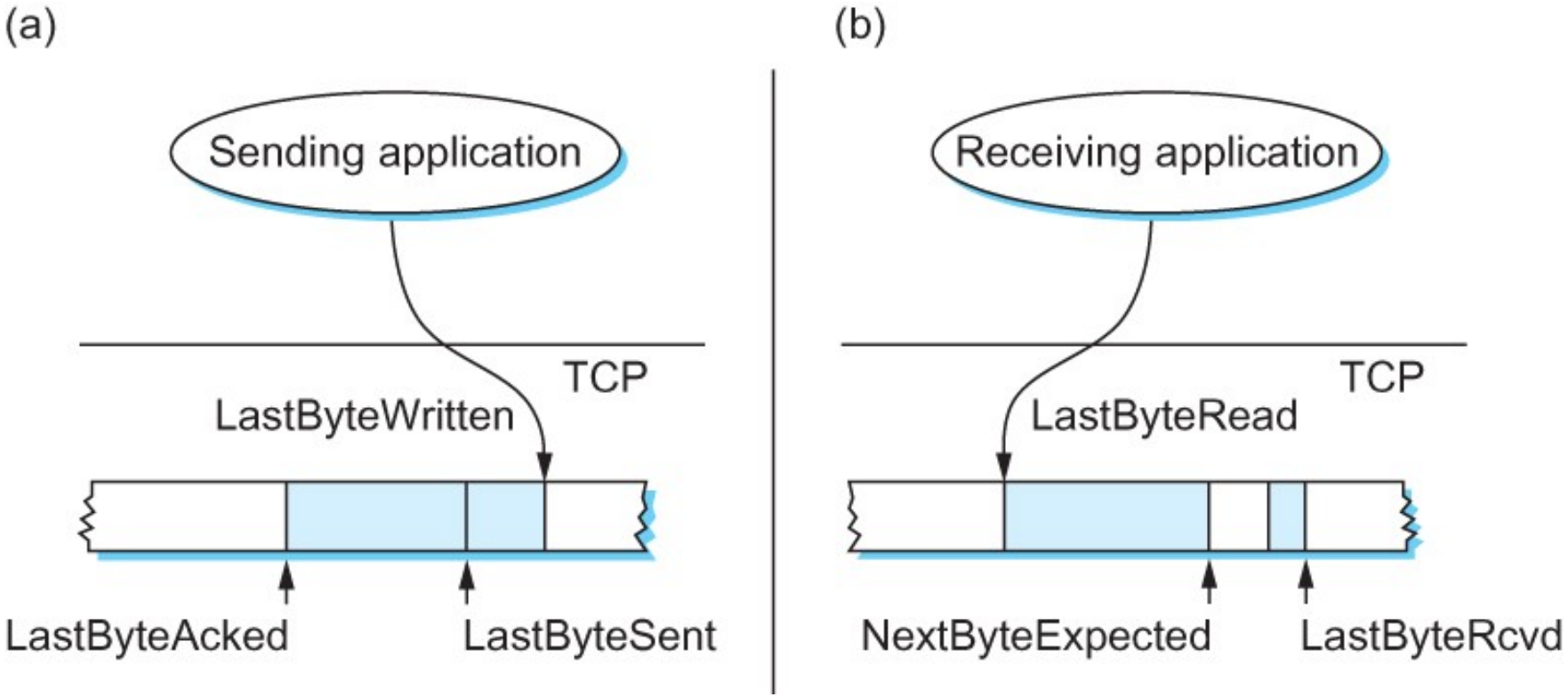
  (1) it guarantees the reliable delivery of data,

  (2) it ensures that data is delivered in order, and

  (3) it enforces flow control between the sender and the receiver.

- TCP sender and receiver have buffers.
  - Receiver buffer holds data received out of order or in-order data not consumed by process.
  - Send buffer holds data not yet sent or not yet ACKed.

Sliding Window Revisited



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: rules

- 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 **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

Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not a de facto ISN of 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … and there is a chance an old packet is still in flight
  - … and might be associated with the new connection
- So, TCP requires changing the ISN over time
  - Set from a 32-bit clock that ticks every 4 microseconds
  - … which only wraps around once every 4.55 hours
- But, this means the hosts need to exchange ISNs

Protecting against Wraparound

Table 22. Time Until 32-Bit Sequence Number Space Wraps Around.

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-48 (2.5 Gbps)	14 seconds
OC-192 (10 Gbps)	3 seconds
10GigE (10 Gbps)	3 seconds

Method: use a TCP header extension 32-bit *timestamp* field as an identifier in fast networks. It is always increasing and the receiver can distinguish between two segments with the same seq. number.

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

Table 23. Required Window Size for 100-ms RTT

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-48 (2.5 Gbps)	29.6 MB
OC-192 (10 Gbps)	118.4 MB
10GigE (10 Gbps)	118.4 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

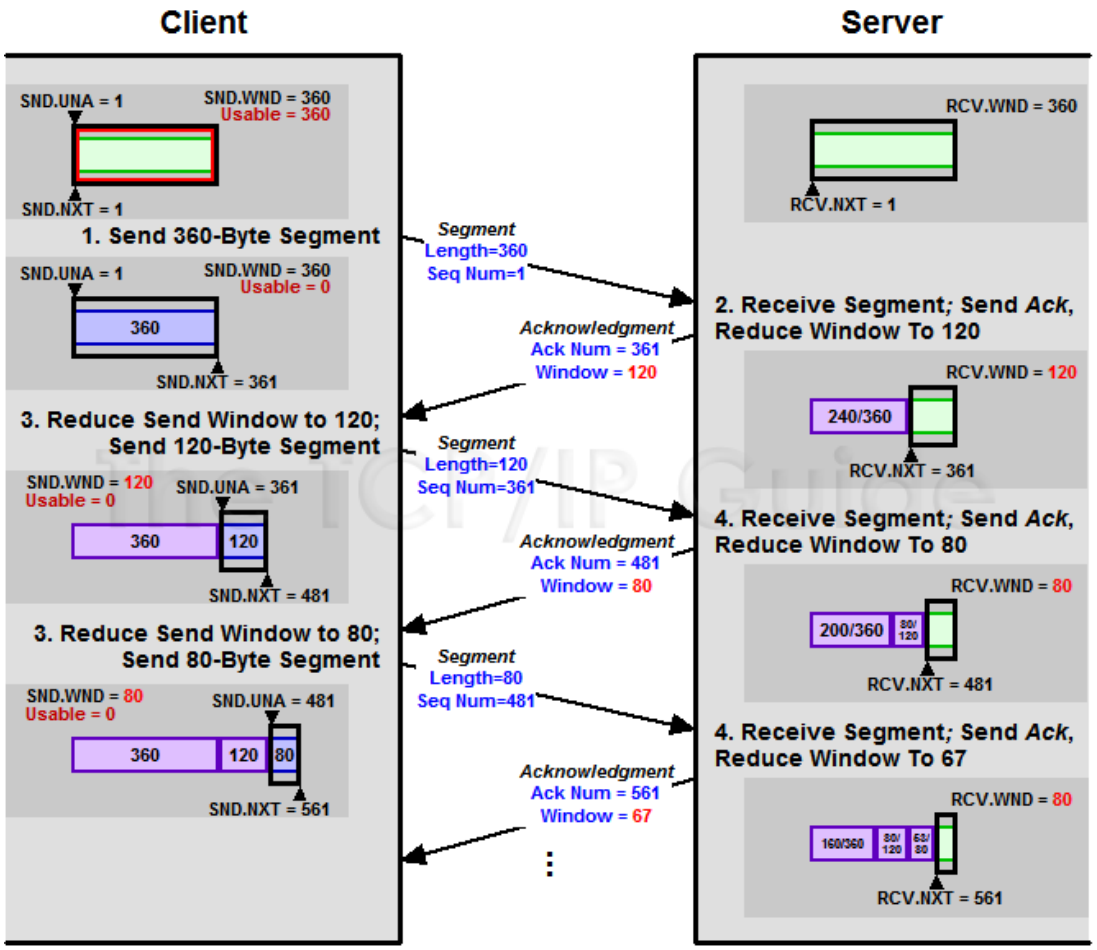
Triggering Transmission

- 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 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



From
http://www.tcpipguide.com/free/t_TCPSillyWindowSyndromeandChangesTotheSlidingWindow.htm

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
```

Estimating RTT: Adaptive Retransmission

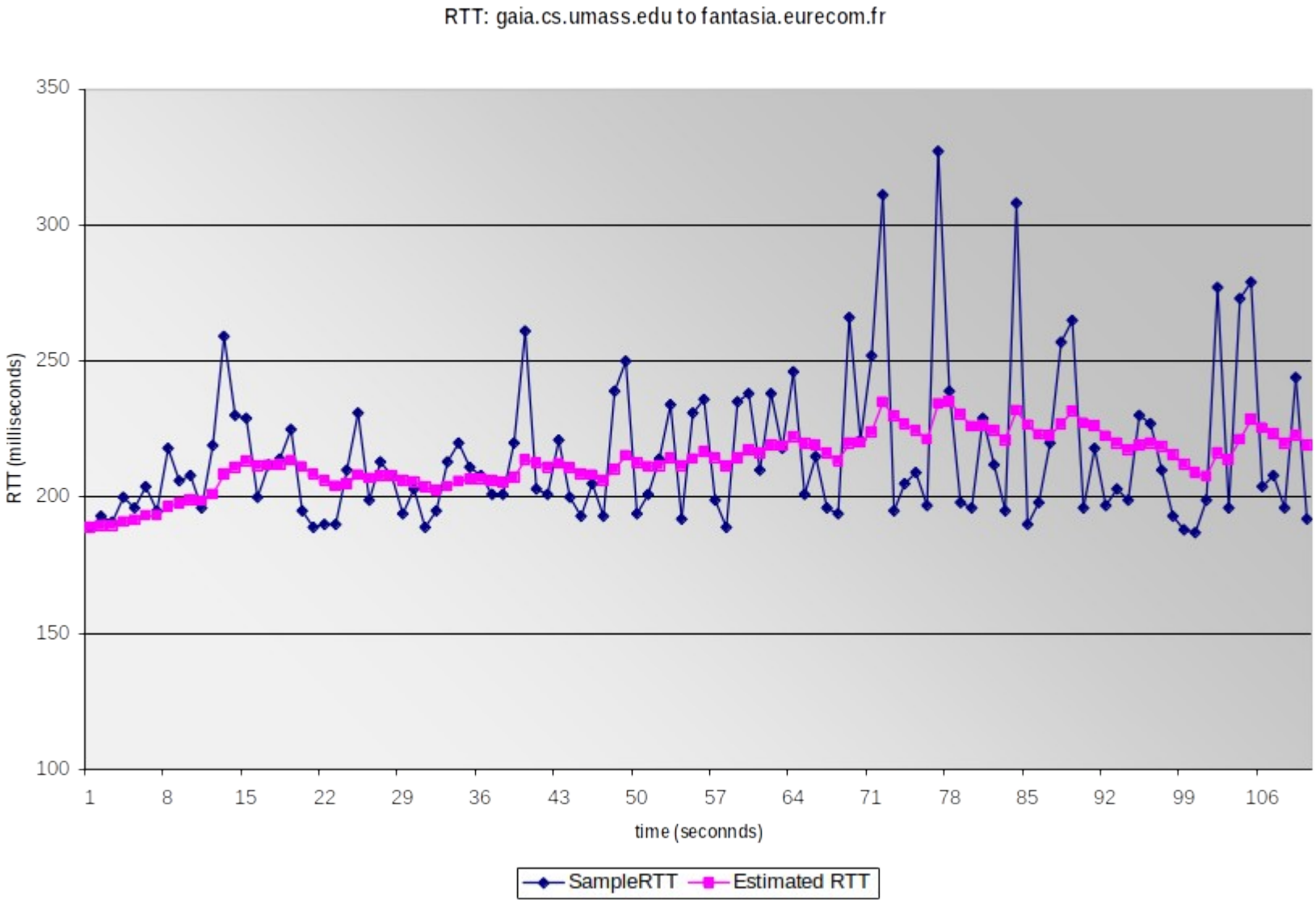
- Original Algorithm
  - Measure **`SampleRTT`** for each segment/ ACK pair
  - Compute weighted average of RTT
    - **`EstRTT`** = α x **`EstRTT`** + (1 - α)x **`SampleRTT`**
    - α between 0.8 and 0.9
  - Set timeout based on **`EstRTT`**
    - **`TimeOut = 2`** x **`EstRTT`**

Original RTT 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

•Retransmissions are caused by congestion (for wired nets)

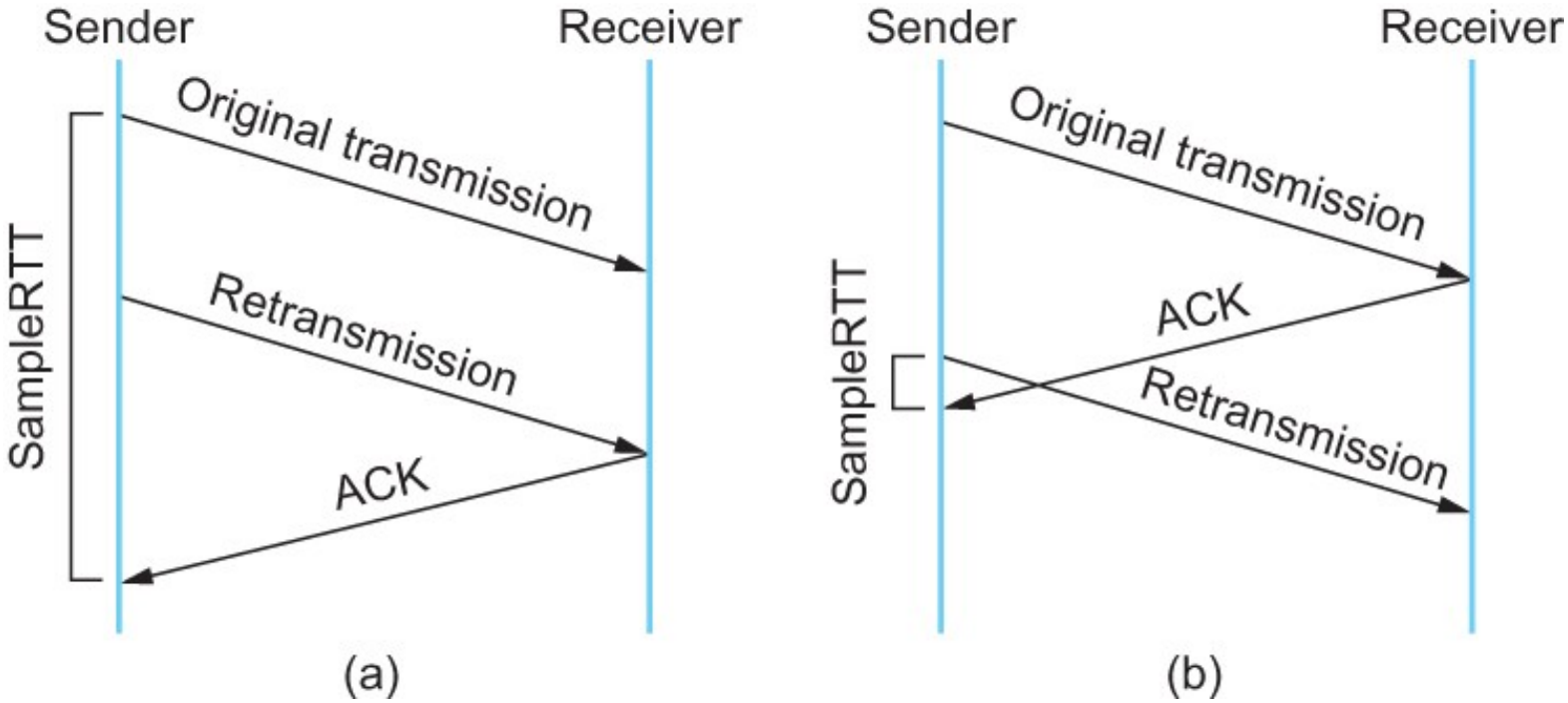
Example RTT Estimation



Karn/Partridge Algorithm

Big problem causing congestion in ‘86-87: wrong RTT causing retransmissions too soon.

ISSUE: RTT measurement error since ACK may come for received segment and for a retransmitted one.



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

Karn/Partridge Algorithm (1987)

- Do not sample RTT when retransmitting
- Double timeout after each retransmission
  - A form of *binary exponential backoff*
  - Conservative approach, assuming lost segment due to congestion

Karn/Partridge Algorithm

- 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

Issues with RTT Estimation

- 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

Issues with RTT Estimation

- 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 + (δ × Difference)
- Deviation = Deviation + δ(|Difference| − Deviation)
- TimeOut = μ × EstimatedRTT + φ × Deviation
  - where based on experience, μ is typically set to 1, φ is set to 4, and δ is between 0 and 1
  - Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the deviation term to dominate the calculation.

TCP Record Boundaries

- TCP is byte-stream oriented, not message-oriented: it has not message boundaries.
- Still, it allows sender to mark message boundaries
- Urgent Data: *Urg* flag and *UrgPtr* header fields
  - Originally used for “out-of-band” data, separate from the normal flow: delivered before the normal data to dest. process
  - Over time Urg has become a record bodady marker
- The Push operation: originally used to flush send buffer to peer (e.g. not wait for a full MSS to fill). Receiver can be notified when Pushed data arrives: marks record boundary.

TCP Extensions

- Options added to TCP header.
  - May not be supported by both peers.
  - Negotiated at connection setup.
- Timestamp extension: sender inserts fine-grained 32 bit system clock timestamp in header. Receiver sends it back with ACK segment. Sender measures elapsed time.
- Sequence number wrapping avoidance: use the timestamp option as the top 32 bits + low 32 bit sequence number.
- Window scaling: allow window size > 64 KB, up to 1GB
- Use Selective Acknowledgments (SACK): optional field ACKs additional data blocks
- Others, for congestion control (Chapter 5)

Summary

- We have discussed how to convert host-to-host packet delivery service to process-to-process communication channel.
- UDP: thin multiplexing on top of IP, best effort
- TCP: reliable byte-stream oriented connection
  - Flow and error control
  - Congestion control is complex
  - Huge impact on overall internet performance.