CSC458

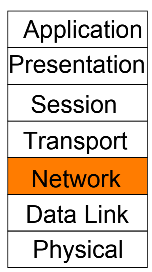
Sliding Windows, ARQ Connections

Administrivia

- Projects
  - Project #3 due on Wednesday at 2pm
  - Project #4 out today -- last project
- Homework
  - Homework #4 out last week, due in two weeks
  - This is our last homework
- Readings
  - Chapters 5.1, 5.2, 6.1, 6.3, 6.4
- No tutorial today

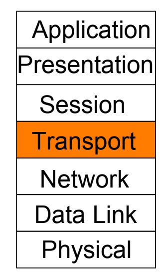
Last Time

- We finished up the Network layer
  - Internetworks (IP)
  - Routing (DV/RIP, LS/OSPF)
- It was all about routing: how to provide end-to-end delivery of packets.



This Time

- We begin on the Transport layer
- Focus
  - How do we send information reliably?
- Topics
  - The Transport layer
  - Acknowledgements and retransmissions (ARQ)
  - Sliding windows



The Transport Layer

- Builds on the services of the Network layer
- Communication between processes running on hosts
  - Naming/Addressing
- Stronger guarantees of message delivery
  - Reliability

Example – Common Properties

TCP

- Connection-oriented
- Multiple processes
- Reliable byte-stream delivery
  - In-order delivery
  - Single delivery
  - Arbitrarily long messages
- Synchronization
- Flow control
- Reliable delivery

IP

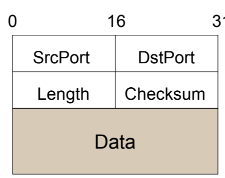
- Datagram oriented
- Lost packets
- Reordered packets
- Duplicate packets
- Limited size packets

What does it mean to be "reliable"

- How can a sender "know" the sent packet was received?
  - sender receives an acknowledgement
- How can a receiver "know" a received packet was sent?
  - sender includes sequence number, checksum
- Do sender and receiver need to come to consensus on what is sent and received?
  - When is it OK for the receiver's TCP/IP stack to deliver the data to the application?

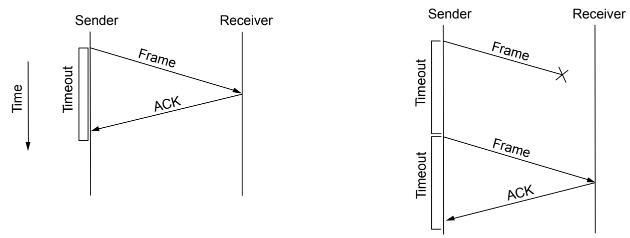
Internet Transport Protocols

- UDP
  - Datagram abstraction between processes
  - With error detection



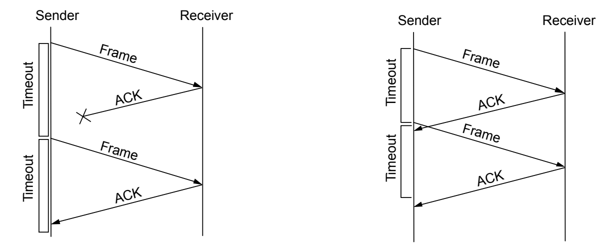
- TCP
  - Bytestream abstraction between processes
  - With reliability
  - Plus congestion control (later!)

Automatic Repeat Request (ARQ)



- Packets can be corrupted or lost. How do we add reliability?
- Acknowledgments (ACKs) and retransmissions after a timeout
- ARQ is generic name for protocols based on this strategy

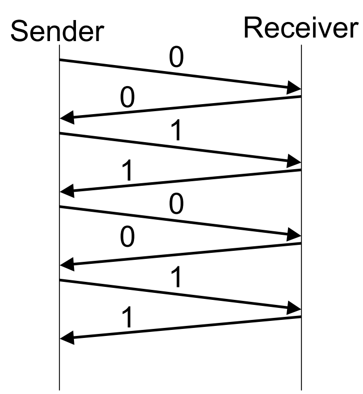
The Need for Sequence Numbers



- In the case of ACK loss (or poor choice of timeout) the receiver can't distinguish this message from the next
  - Need to understand how many packets can be outstanding and number the packets; here, a single bit will do

Stop-and-Wait

- Only one outstanding packet at a time
- Also called alternating bit protocol

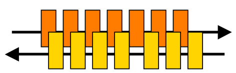


Limitation of Stop-and-Wait

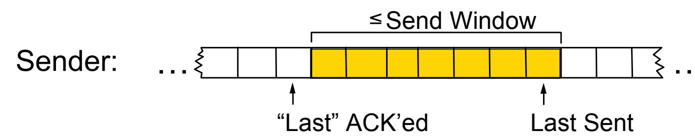
- Lousy performance if trans. delay << prop. delay
  - Max BW: B
  - Actual BW: M/2D
    - Example: B = 100Mb/s, M=1500Bytes, D=50ms
    - Actual BW = 1500Bytes/100ms --> 15000 Bytes/s --> ~100Kb/s
    - 100Mb vs 100Kb?

More BW Please

- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Remember the bandwidth-delay product?
- Leads to Sliding Window Protocol
  - “window size” says how much data can be sent without waiting for an acknowledgement

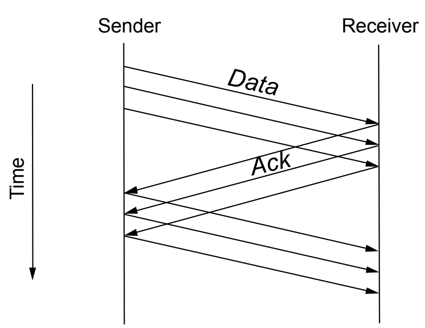


Sliding Window – Sender



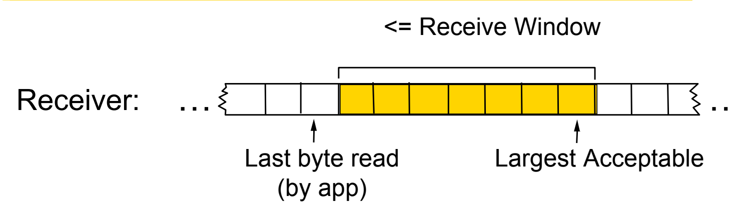
- Window bounds outstanding data
  - Implies need for buffering at sender
    - Specifically, must buffer unack’ed data
- “Last” ACK applies to in-order data
  - Need not buffer acked data
- Sender maintains timers too
  - Go-Back-N: one timer, send all unacknowledged on timeout
  - Selective Repeat: timer per packet, resend as needed

Sliding Window – Timeline



- Receiver ACK choices:
  - Individual
    - Each packet acked
  - Cumulative (TCP)
    - Ack says “got everything up to X-1…”
    - really, “my ack means that the next byte I am expecting is X”
  - Selective (newer TCP)
    - Ack says “I got X through Y”
  - Negative
    - Ack says “I did not get X”

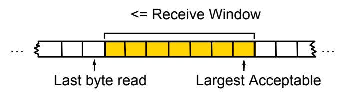
Sliding Window – Receiver



- Receiver buffers too:
  - data may arrive out-of-order
  - or faster than can be consumed by receiving process
- No sense having more data on the wire than can be buffered at the receiver.
  - In other words, receiver buffer size should limit the sender’s window size

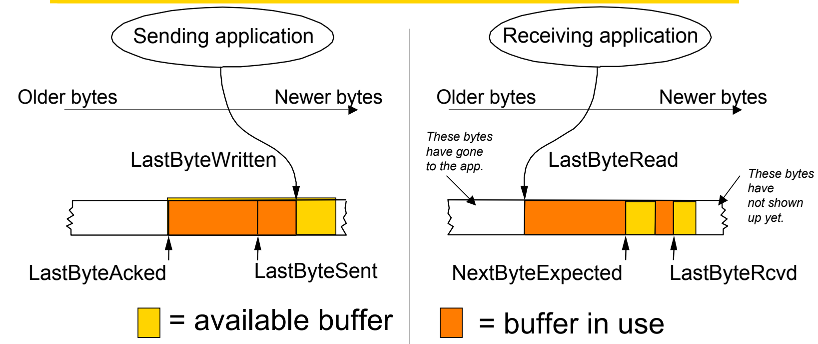
Flow Control

- Sender must transmit data no faster than it can be consumed by receiver
  - Receiver might be a slow machine
  - App might consume data slowly



- Accomplish by adjusting the size of sliding window used at the sender
  - sender adjusts based on receiver's feedback about available buffer space
  - the receiver tells the sender an "Advertised Window"

Sender and Receiver Buffering

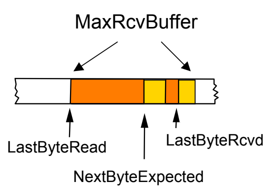


LastByteAcked <= LastByteSent
LastByteSent <= LastByteWritten

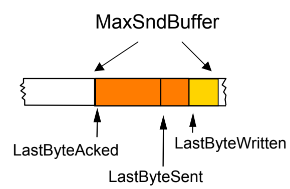
LastByteRead < NextByteExpected
NextByteExpected <= LastByteRcvd+1
== if data arrives in order
else start of first gap.

Flow Control

Receiver:



Sender:



Receiver's goal: always ensure that LastByteRcvd - LastByteRead <= MaxRcvBuffer

- in other words, ensure it never needs to buffer more than MaxRcvBuffer data

To accomplish this, receiver advertises the following window size:

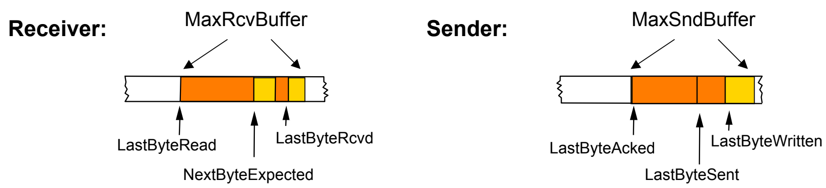
- AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)
- *"All the buffer space minus the buffer space that's in use."*

Flow control on the receiver

- As data arrives:
  - receiver acknowledges it so long as all preceding bytes have also arrived
  - ACKs also carry a piggybacked AdvertisedWindow
  - So, an ACK tells the sender:
    1. All data up to the ACK'ed seqno has been received
    2. How much more data fits in the receiver's buffer, as of receiving the ACK'ed data

- AdvertisedWindow:
  - shrinks as data is received
  - grows as receiving app. reads the data from the buffer

Flow Control On the Sender



Sender's goal: always ensure that LastByteSent - LastByteAcked <= AdvertisedWindow

- in other words, don't sent that which is unwanted

Notion of "EffectiveWindow": how much new data it is OK for sender to currently send

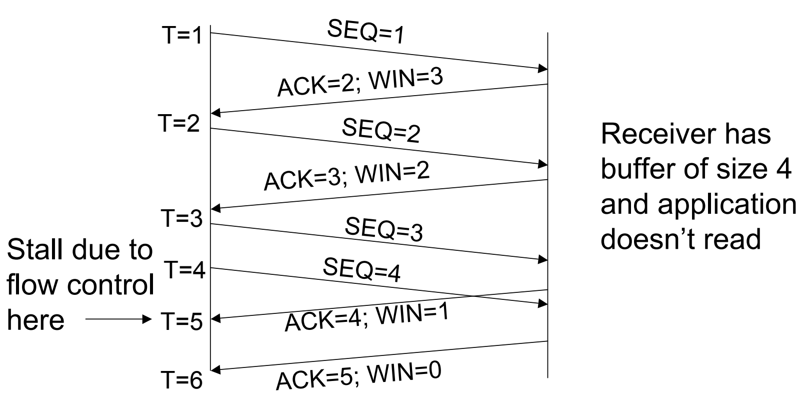
- EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)

OK to send that which there is room for, which is that which was advertised (AdvertisedWindow) minus that which I've already sent since receiving the last advertisement.

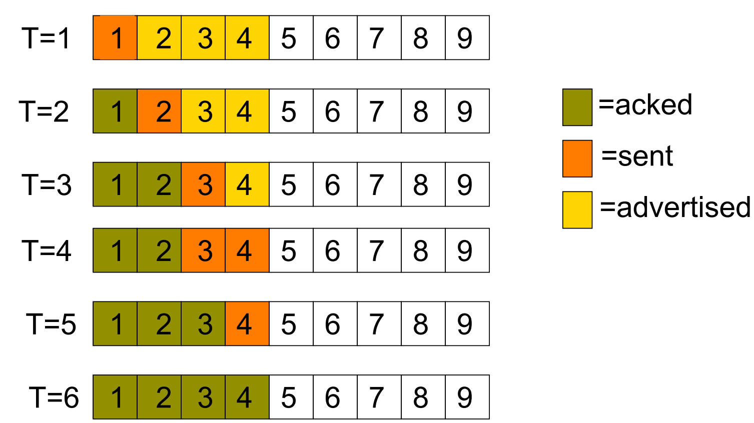
Sending Side

- As acknowledgements arrive:
  - advance LastByteAcked
  - update AdvertisedWindow
  - calculate new EffectiveWindow
    - If EffectiveWindow > 0, it is OK to send more data
- One last detail on the sender:
  - sender has finite buffer space as well
    - LastByteWritten - LastByteAcked <= MaxSendBuffer
  - OS needs to block application writes if buffer fills
    - i.e., block write(y) if
      (LastByteWritten - LastByteAcked) + y > MaxSendBuffer

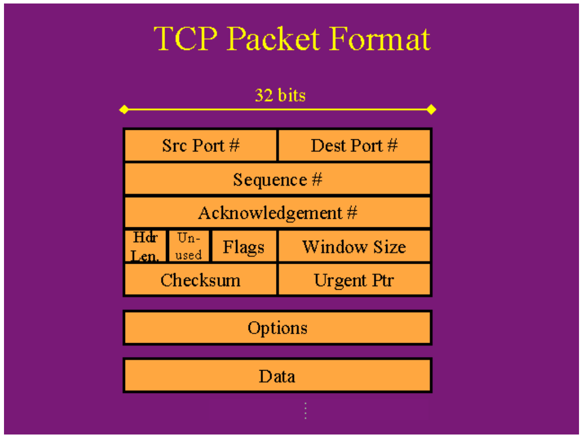
Example – Exchange of Packets



Example – Buffer at Sender



Packet Format



16 bit window size gets
Cramped with large
Bandwidth x delay

16 bits --> 64K
BD ethernet: 122KB
STS24 (1.2Gb/s): 14.8MB

32 bit sequence number
must not wrap around faster
than the maximum packet
lifetime. (120 seconds)
-- 622Mb/s link: 55 seconds

Sliding Window Functions

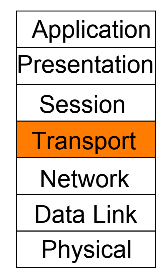
- Sliding window is a mechanism
- It supports multiple functions:
  - Reliable delivery
    - *If I hear you got it, I know you got it.*
    - ACK (Ack # is "next byte expected")
  - In-order delivery
    - *If you get it, you get it in the right order.*
    - SEQ # (Seq # is "the byte this is in the sequence")
  - Flow control
    - *If you don't have room for it, I won't send it.*
    - Advertised Receiver Window
    - AdvertisedWindow is amount of free space in buffer

Key Concepts

- Transport layer allows processes to communicate with stronger guarantees, e.g., reliability
- Basic reliability is provided by ARQ mechanisms
  - Stop-and-Wait through Sliding Window plus retransmissions

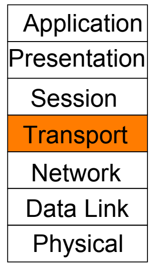
Last Time

- We began on the Transport layer
- Focus
  - How do we send information reliably?
- Topics
  - ARQ and sliding windows



This Time

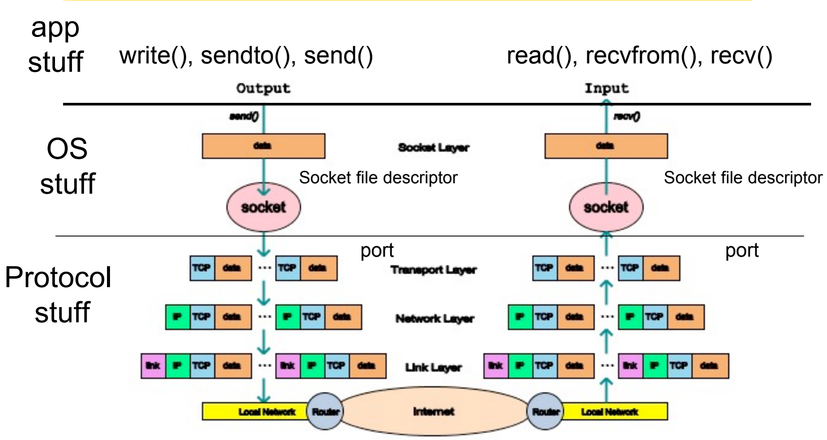
- More on the Transport Layer
- Focus
  - How do we connect processes?
- Topics
  - Naming processes
  - Connection setup / teardown
  - Flow control



Naming Processes/Services

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.
- How do we identify for remote communication?
  - Process id or memory address are OS-specific and transient
- So TCP and UDP use Ports
  - 16-bit integers representing mailboxes that processes "rent"
    - typically from OS
  - Identify process uniquely as (IP address, protocol, port)
    - OS converts into process-specific channel, like "socket"

Processes as Endpoints

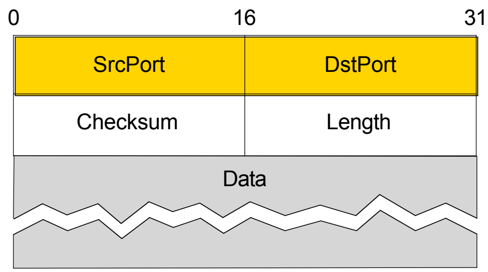


Picking Port Numbers

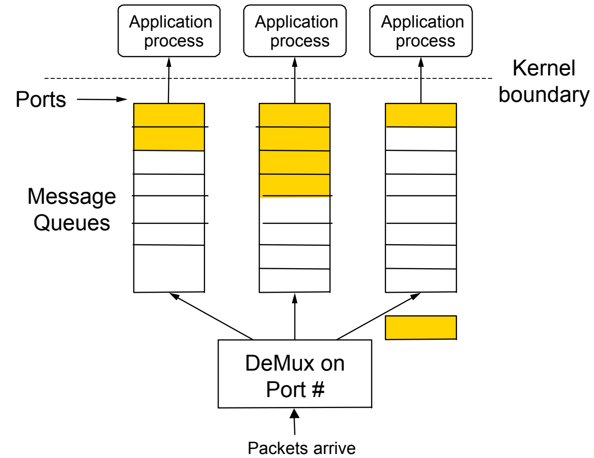
- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?
- Servers typically bind to "well-known" port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, … look in /etc/services
  - Ports below 1024 reserved for "well-known" services
- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished

User Datagram Protocol (UDP)

- Provides message delivery between processes
  - Source port filled in by OS as message is sent
  - Destination port identifies UDP delivery queue at endpoint

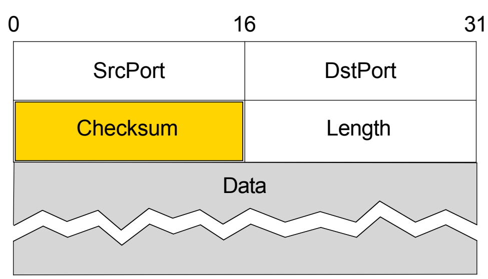


UDP Delivery



UDP Checksum

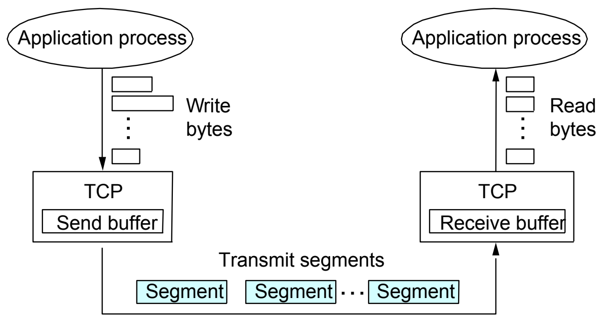
- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header, and IP pseudoheader



Transmission Control Protocol (TCP)

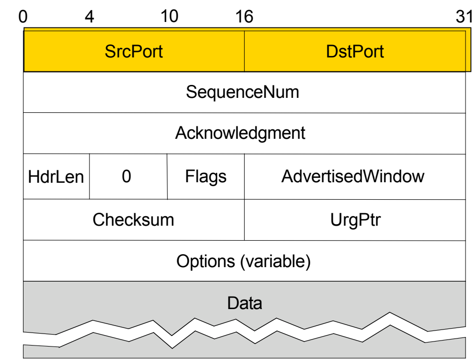
- Reliable bi-directional bytestream between processes
  - Message boundaries are not preserved
- Connections
  - Conversation between endpoints with beginning and end
- Flow control
  - Prevents sender from over-running receiver buffers
- Congestion control
  - Prevents sender from over-running network buffers

TCP Delivery



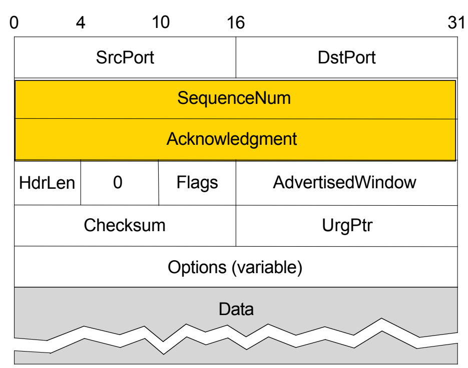
TCP Header Format

- Ports plus IP addresses identify a connection



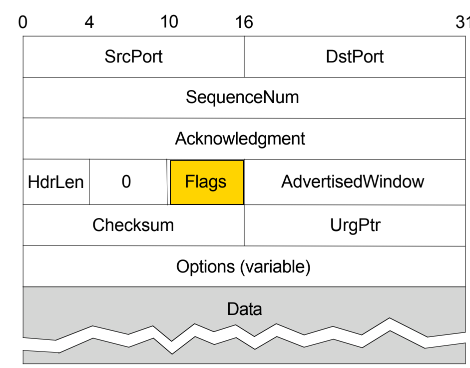
TCP Header Format

- Sequence, Ack numbers used for the sliding window



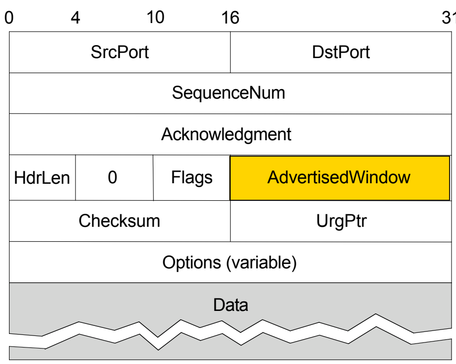
TCP Header Format

- Flags may be URG, ACK, PSH, RST, SYN, FIN



TCP Header Format

- Advertised window is used for flow control



Other TCP Header Fields

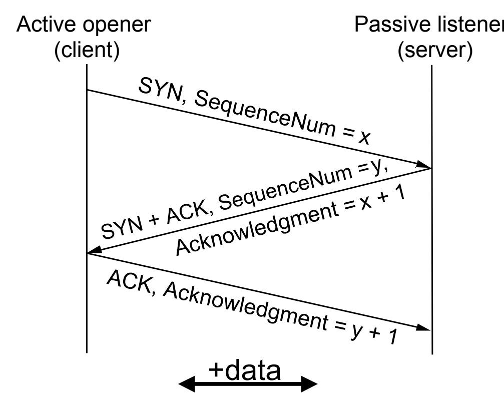
- Header length allows for variable length TCP header
  - options for extensions such as timestamps, selective acknowledgements, etc.
- Checksum is analogous to that of UDP
- Urgent pointer/data not used in practice

TCP Connection Establishment

- Both sender and receiver must be ready before we start to transfer the data
  - Sender and receiver need to agree on a set of parameters
  - e.g., the Maximum Segment Size (MSS)
- This is “signaling”
  - It sets up state at the endpoints
  - Compare to “dialing” in the telephone network
- In TCP a Three-Way Handshake is used

Three-Way Handshake

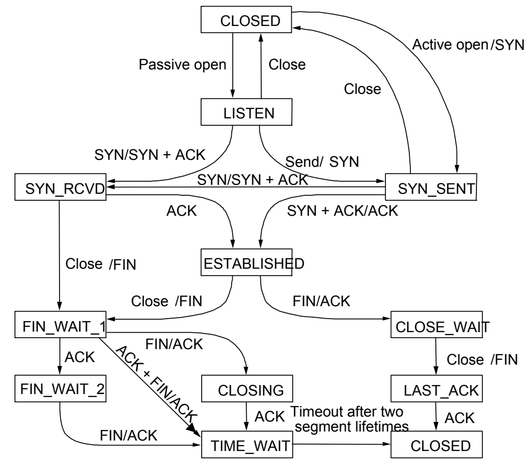
- Opens both directions for transfer



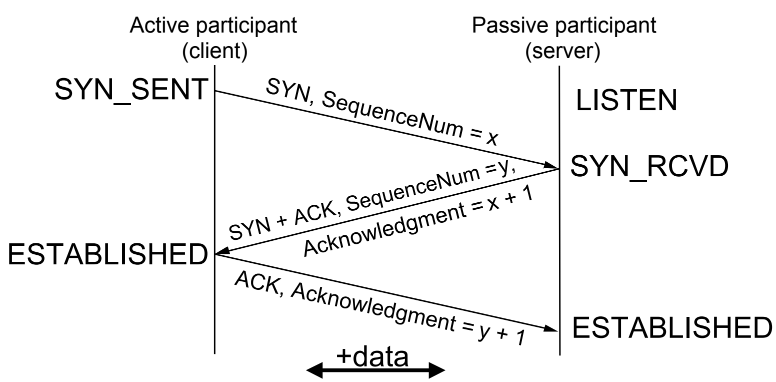
Some Comments

- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
  - Three-way handshake from Tomlinson 1975
- Choice of changing initial sequence numbers (ISNs) minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection
- But with random ISN it actually proves that two hosts can communicate
  - Weak form of authentication

TCP State Transitions



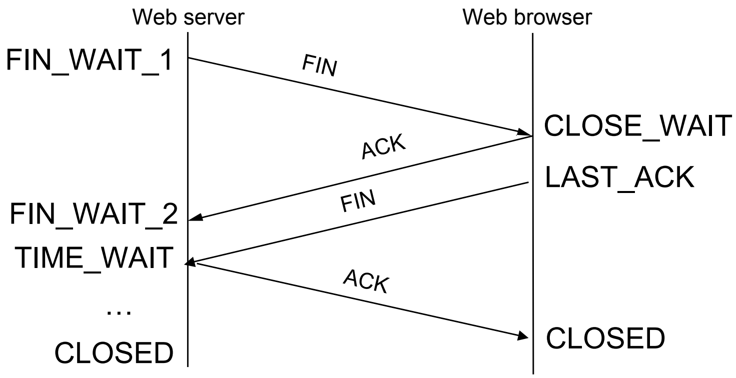
Again, with States



Connection Teardown

- Orderly release by sender and receiver when done
  - Delivers all pending data and "hangs up"
- Cleans up state in sender and receiver
- TCP provides a "symmetric" close
  - both sides shutdown independently

TCP Connection Teardown



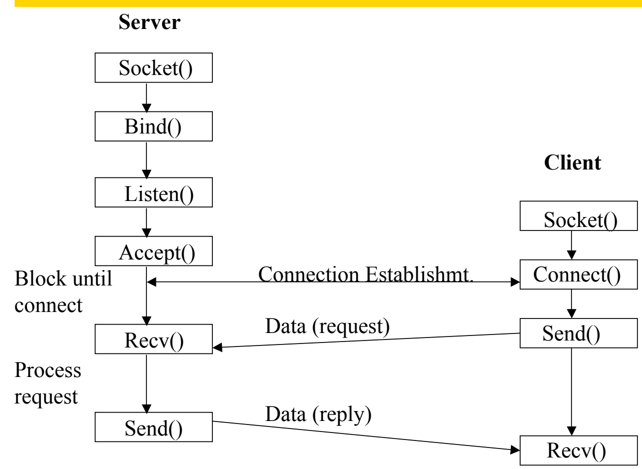
The TIME_WAIT State

- We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close
- Why?
- ACK might have been lost and so FIN will be resent
- Could interfere with a subsequent connection

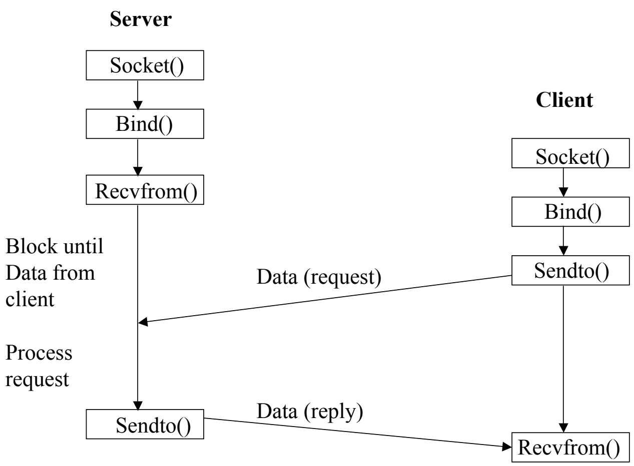
Berkeley Sockets interface

- Networking protocols implemented in OS
  - OS must expose a programming API to applications
  - most OSs use the "socket" interface
  - originally provided by BSD 4.1c in ~1982.
- Principle abstraction is a "socket"
  - a point at which an application attaches to the network
  - defines operations for creating connections, attaching to network, sending and receiving data, closing connections

TCP (connection-oriented)



UDP (connectionless)



Socket call

- Means by which an application attached to the network
  - #include <sys/socket.h>…
- int socket(int family, int type, int protocol)
- *Family*: address family (protocol family)
  - AF_UNIX, AF_INET, AF_NS, AF_IMPLINK
- *Type*: semantics of communication
  - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
  - Not all combinations of family and type are valid
- *Protocol*: Usually set to 0 but can be set to specific value.
  - Family and type usually imply the protocol
- Return value is a *handle* for new socket

Bind call

- Typically a server call
- Binds a newly created socket to the specified address
  - *int bind(int socket, struct sockaddr *address, int addr_len)*
- *Socket*: newly created socket handle
- *Address*: data structure of address of *local* system
  - IP address and port number (demux keys)
  - Same operation for both connection-oriented and connectionless servers
    - Can use well known port or unique port

Listen call

- Used by connection-oriented servers to indicate an application is willing to receive connections
- Int(int socket, int backlog)
- *Socket*: handle of newly creates socket
- *Backlog*: number of connection requests that can be queued by the system while waiting for server to execute accept call.

Accept call

- A server call
- After executing *listen*, the accept call carries out a *passive open* (server prepared to accept connects).
- int accept(int socket, struct sockaddr *address, int addr_len)
- It blocks until a remote client carries out a connection request.
- When it does return, it returns with a *new* socket that corresponds with new connection and the address contains the clients address

Connect call

- A client call
- Client executes an *active open* of a connection
  - *int connect(int socket, struct sockaddr *address, int addr_len)*
  - How does the OS know where the server is?
- Call does not return until the three-way handshake (TCP) is complete
- Address field contains remote system's address
- Client OS usually selects random, unused port

Input and Output

- After connection has been made, application uses send/recv to data
- int send(int socket, char *message, int msg_len, int flags)
  - Send specified message using specified socket
- int recv(int socket, char *buffer, int buf_len, int flags)
  - Receive message from specified socket into specified buffer
- Or can use read/write
  - int read(int socket, char* buffer, int len)
  - int write(int socket, char* buffer, int len);
- Or can sometimes use sendto/recvfrom
- Or can use sendmsg, recvmsg for "scatter/gather"

Sample Code

Key Concepts

- We use ports to name processes in TCP/UDP
  - "Well-known" ports are used for popular services
- Connection setup and teardown complicated by the effects of the network on messages
  - TCP uses a three-way handshake to set up a connection
  - TCP uses a symmetric disconnect