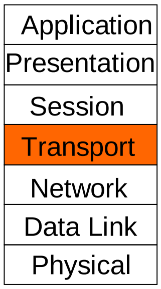
CSE/EE 461
Connections

Last Time

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

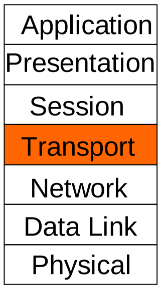

Application
Presentation
Session
Transport
Network
Data Link
Physical

This Time

- More on the Transport Layer

- Focus
  - How do we connect processes?

- Topics
  - Naming processes
  - The Socket interface
  - Connection setup / teardown
  - TCP State Diagram
    - http://www.faqs.org/rfcs/rfc793.html



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” from some per-IP-endpoint broker
    - Typically the OS
  - Identify process uniquely as (IP address, protocol, port)
    - OS converts into [process-specific communications channel]
      - Such as a *socket* file descriptor or a message queue
      - Totally OS dependent. Outside the protocol

Processes as Endpoints

write(), sendto(), send()

read(), recvfrom(), recv()

OS
stuff

Socket file descriptor

Socket file descriptor

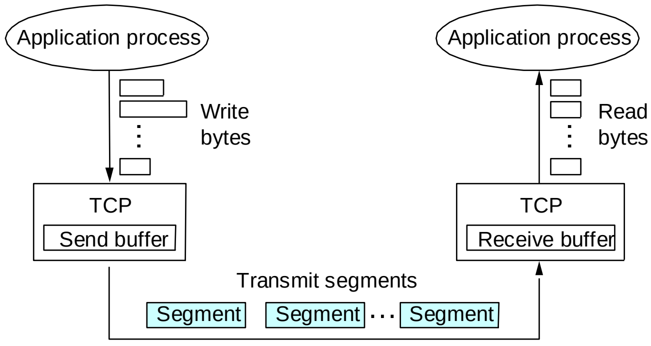
Protocol
stuff

port

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

port

TCP Delivery



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
  - Originally considered ‘secure’ but that didn’t last long
- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished
- It’s all actually quite messy.

7

UNIX Sockets

The Ugly Truth

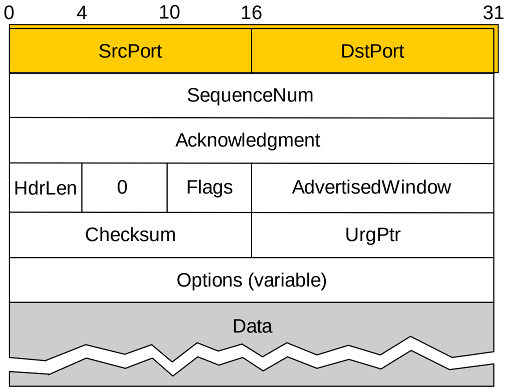
8

Berkeley Sockets

- Networking protocols are implemented as part of the OS
  - The networking API exported by most OS's is the *socket interface*
  - Originally provided by BSD 4.1c ~1982.
- The principal abstraction is a socket
  - Point at which an application attaches to the network
  - Defines operations for creating connections, attaching to network, sending/receiving data, closing.

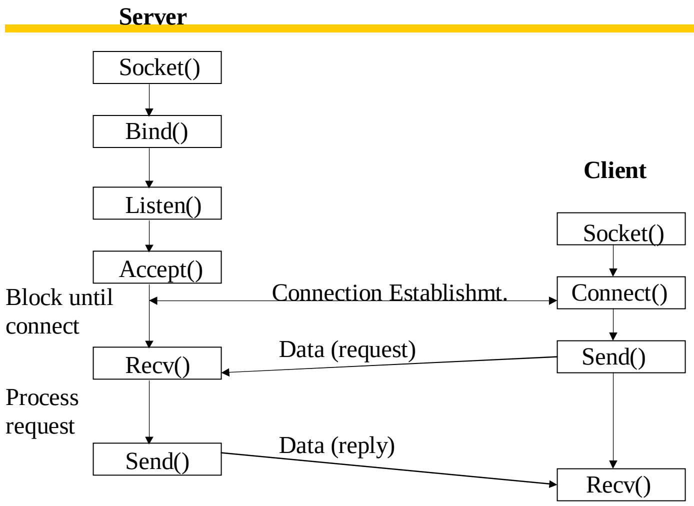
TCP Header Format

- Ports plus IP addresses identify a connection



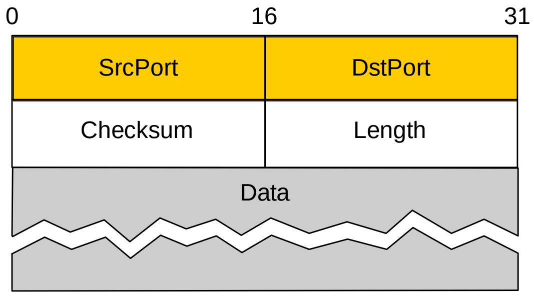
0	4	10	16	31
SrcPort			DstPort	
SequenceNum				
Acknowledgment				
HdrLen	0	Flags	AdvertisedWindow	
Checksum			UrgPtr	
Options (variable)				
Data				

Connection-oriented example (TCP)

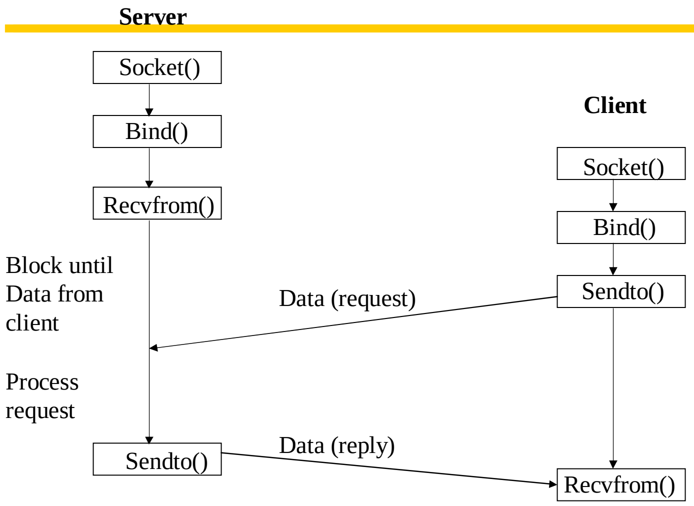


User Datagram Protocol (UDP)

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



Connectionless example (UDP)



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

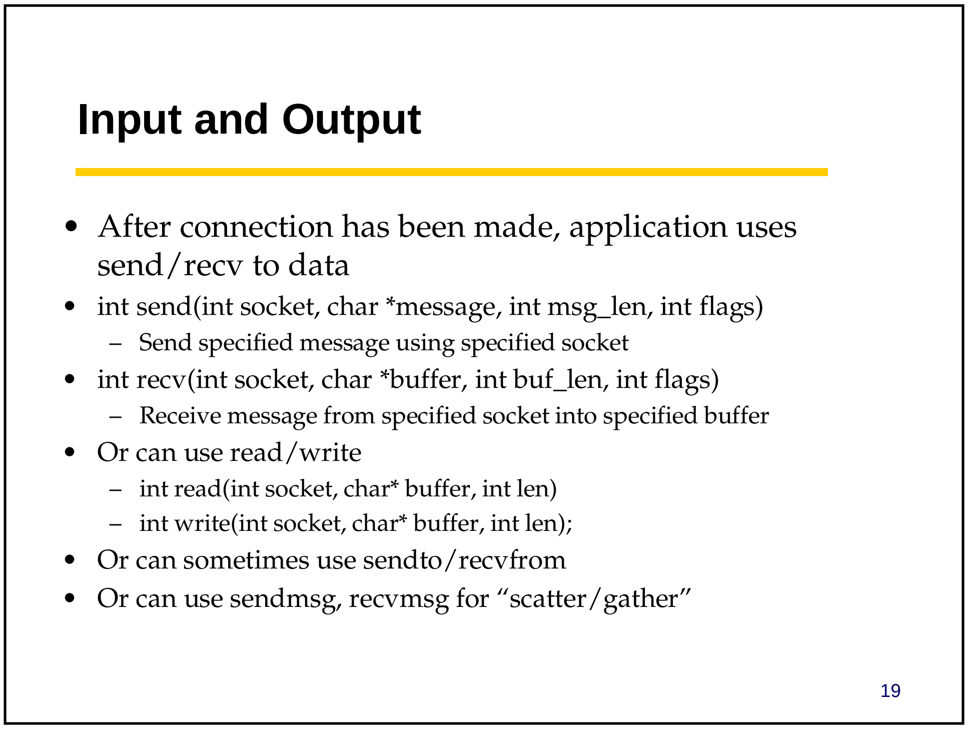
- Used by connection-oriented servers to indicate an application is willing to receive connections
- Int(int socket, int backlog)
- *Socket*: handle of newly creates socket
- *Backlog*: number of connection requests that can be queued by the system while waiting for server to execute accept call.

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

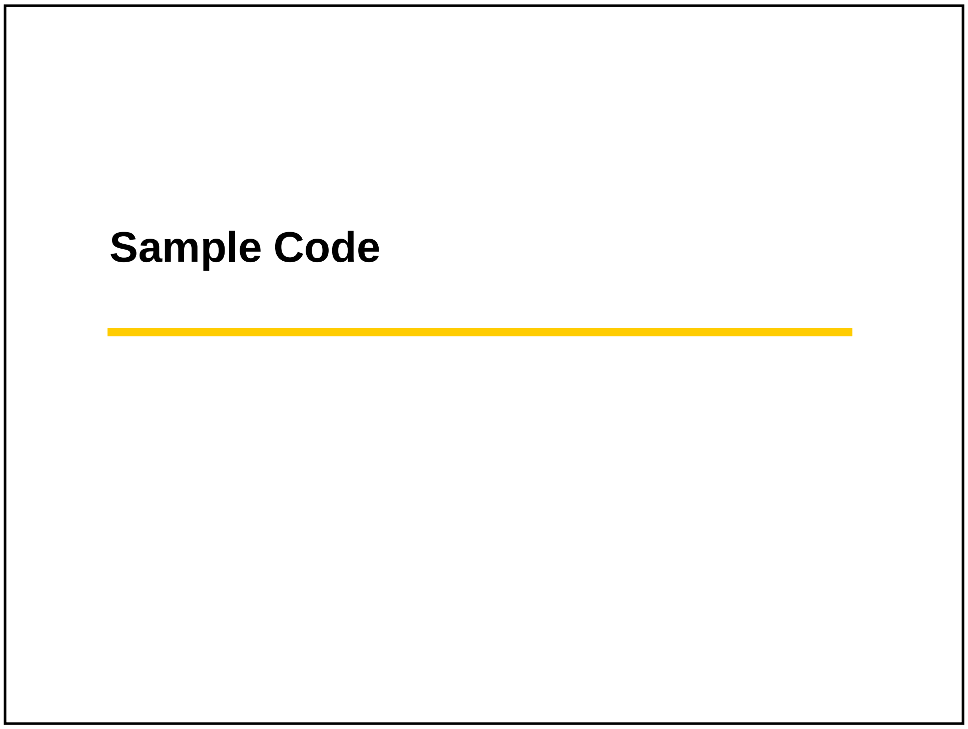
Connect call

- 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”

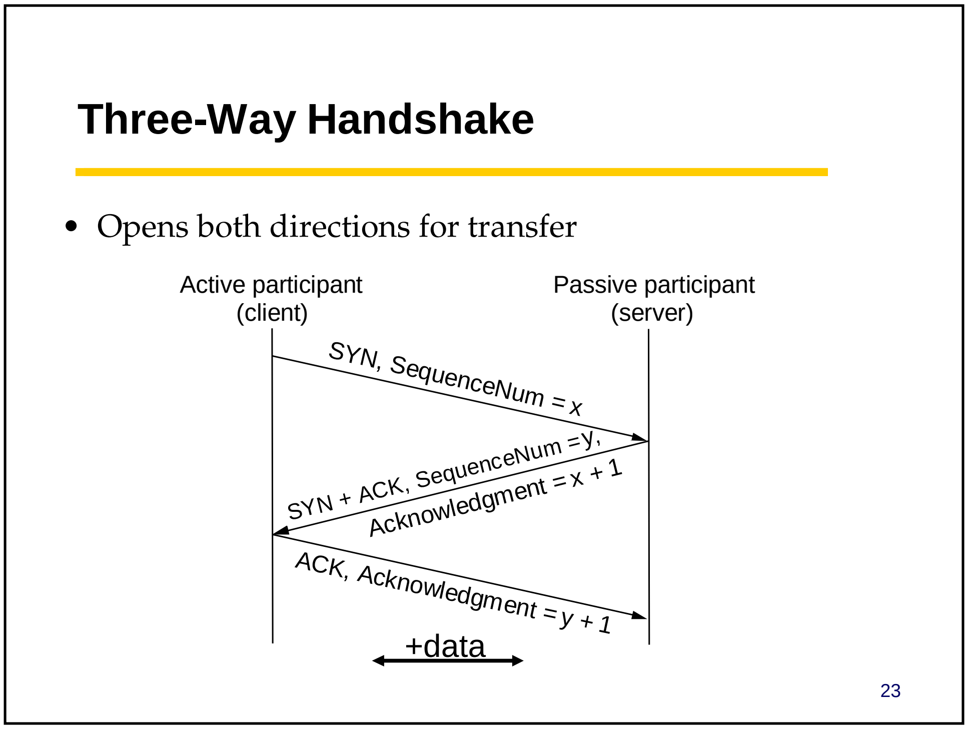


Sample Code

Returning to the Land of Abstraction…

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

Active participant
(client)

Passive participant
(server)

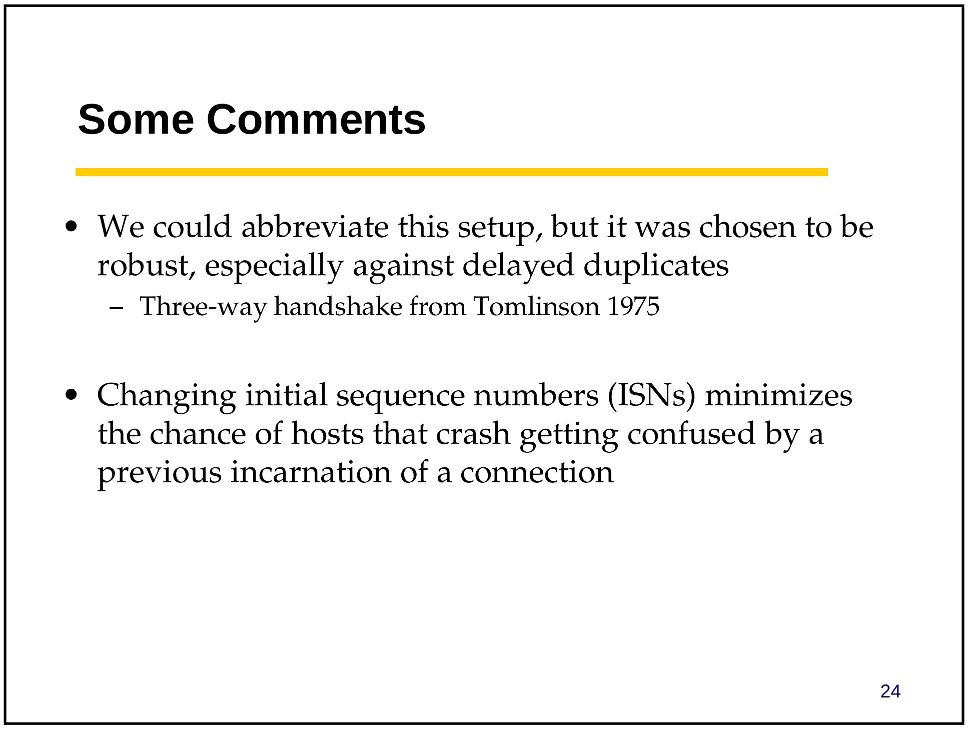
SYN, SequenceNum = *x*

SYN + ACK, SequenceNum = *y*,
Acknowledgment = *x* + 1

ACK, Acknowledgment = *y* + 1

+data

23

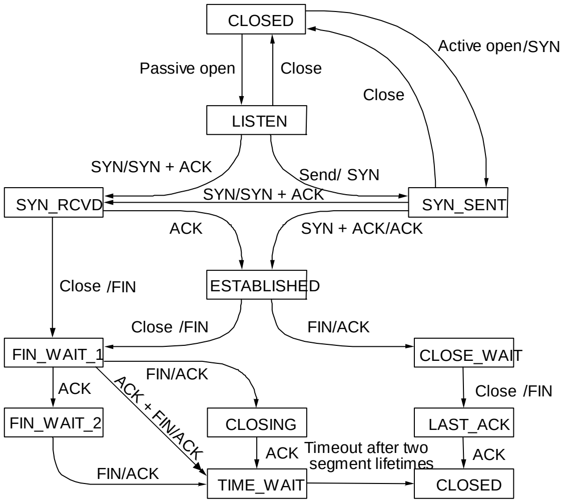


Some Comments

- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
  - Three-way handshake from Tomlinson 1975
- Changing initial sequence numbers (ISNs) minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection

24

TCP State Transitions



CLOSED

Passive open

Close

Active open/SYN

Close

LISTEN

SYN/SYN + ACK

Send/ SYN

SYN_RCVD

SYN/SYN + ACK

SYN_SENT

ACK

SYN + ACK/ACK

ESTABLISHED

Close /FIN

Close /FIN

FIN/ACK

FIN_WAIT_1

CLOSE_WAIT

FIN/ACK

ACK

ACK + FIN/ACK

Close /FIN

FIN_WAIT_2

CLOSING

LAST_ACK

ACK

Timeout after two segment lifetimes

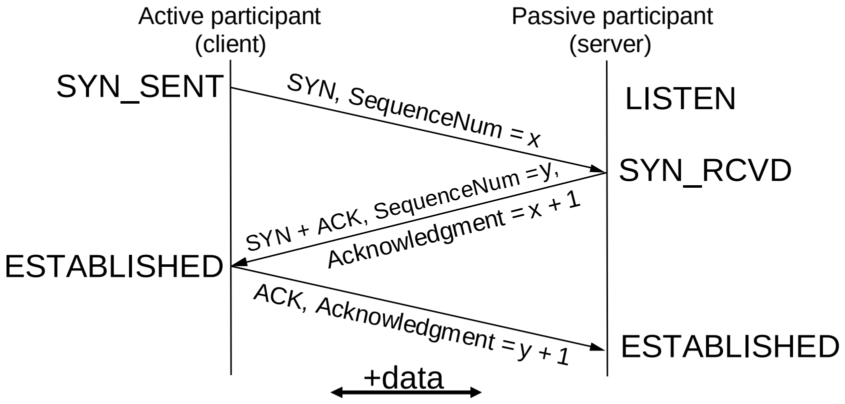
ACK

FIN/ACK

TIME_WAIT

CLOSED

Again, with States



Active participant (client)

Passive participant (server)

SYN_SENT

LISTEN

SYN, SequenceNum = x

SYN_RCVD

SYN + ACK, SequenceNum = y, Acknowledgment = x + 1

ESTABLISHED

ACK, Acknowledgment = y + 1

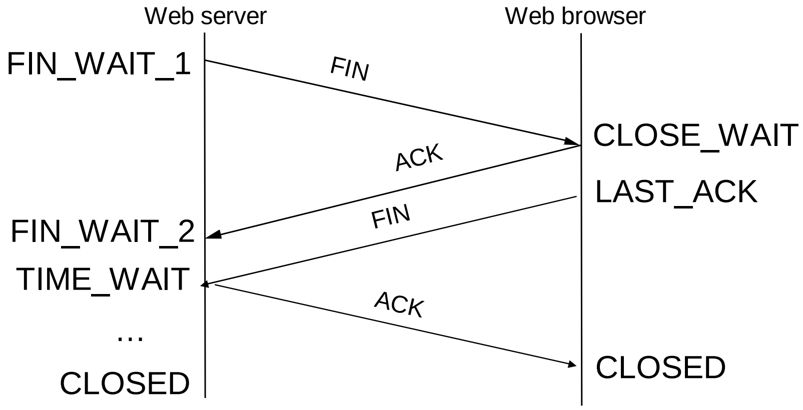
ESTABLISHED

+data

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



Web server

Web browser

FIN_WAIT_1

FIN

CLOSE_WAIT

ACK

LAST_ACK

FIN

FIN_WAIT_2

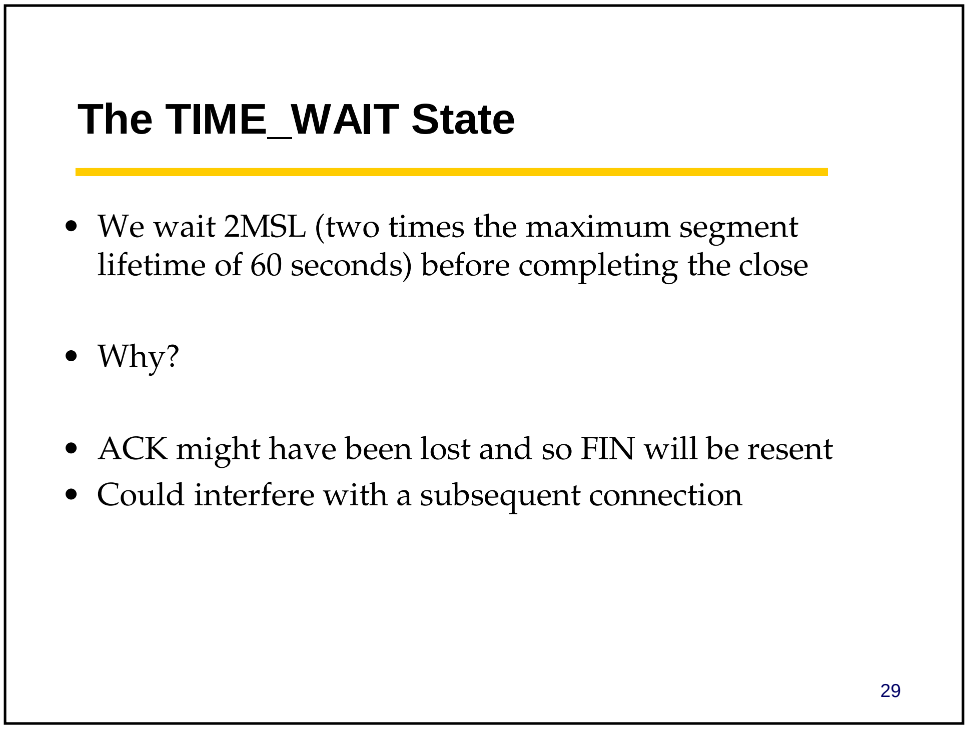
TIME_WAIT

ACK

…

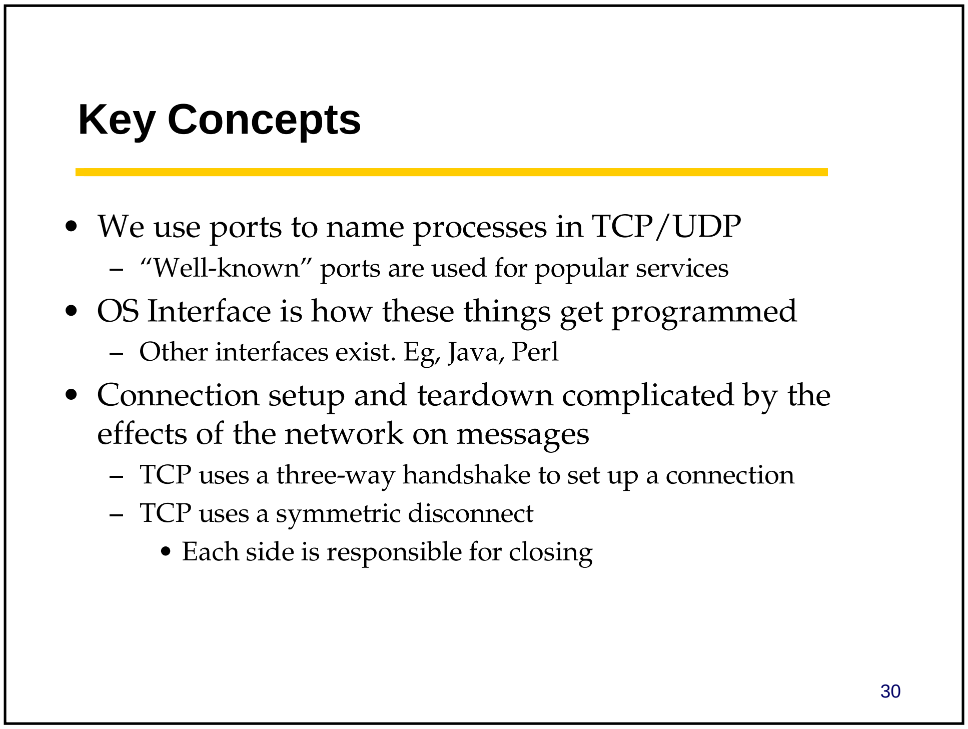
CLOSED

CLOSED



The TIME_WAIT State

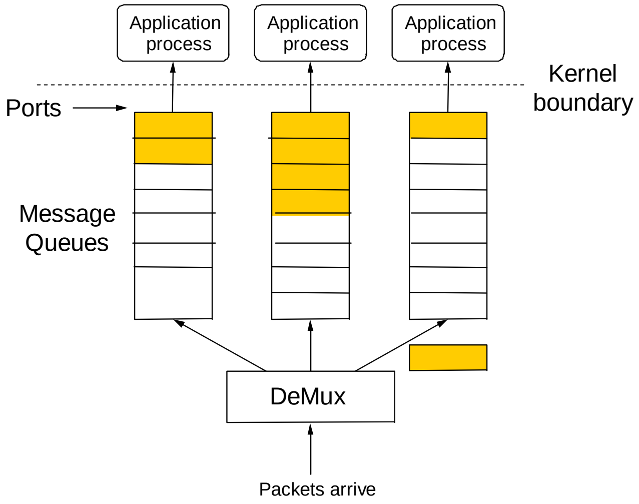
- We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close
- Why?
- ACK might have been lost and so FIN will be resent
- Could interfere with a subsequent connection



Key Concepts

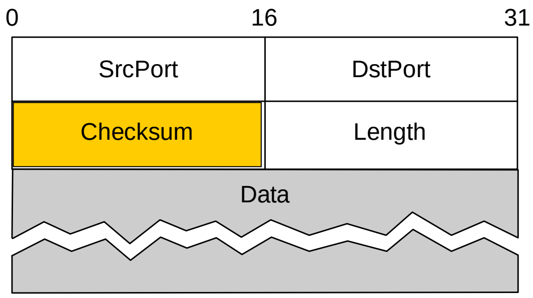
- We use ports to name processes in TCP/UDP
  - "Well-known" ports are used for popular services
- OS Interface is how these things get programmed
  - Other interfaces exist. Eg, Java, Perl
- Connection setup and teardown complicated by the effects of the network on messages
  - TCP uses a three-way handshake to set up a connection
  - TCP uses a symmetric disconnect
    - Each side is responsible for closing

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



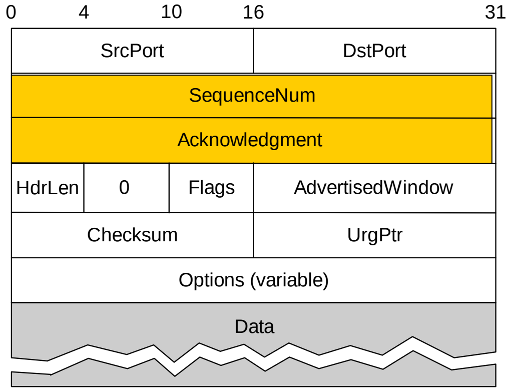
0	16	31
SrcPort	DstPort	
Checksum	Length	
Data		

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 (later)
  - Prevents sender from over-running network buffers

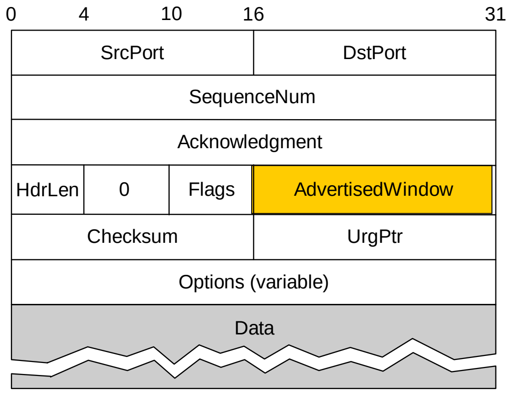
TCP Header Format

- Sequence, Ack numbers used for the sliding window
  - Congestion control works by controlling the window size



TCP Header Format

- Advertised window is used for flow control



0 4 10 16 31

SrcPort | DstPort

SequenceNum

Acknowledgment

HdrLen | 0 | Flags | AdvertisedWindow

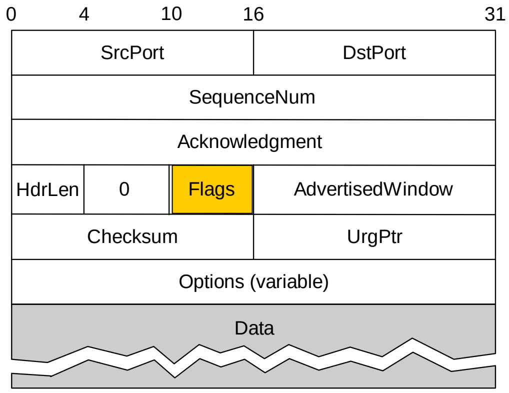
Checksum | UrgPtr

Options (variable)

Data

TCP Header Format

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



0 4 10 16 31

SrcPort | DstPort

SequenceNum

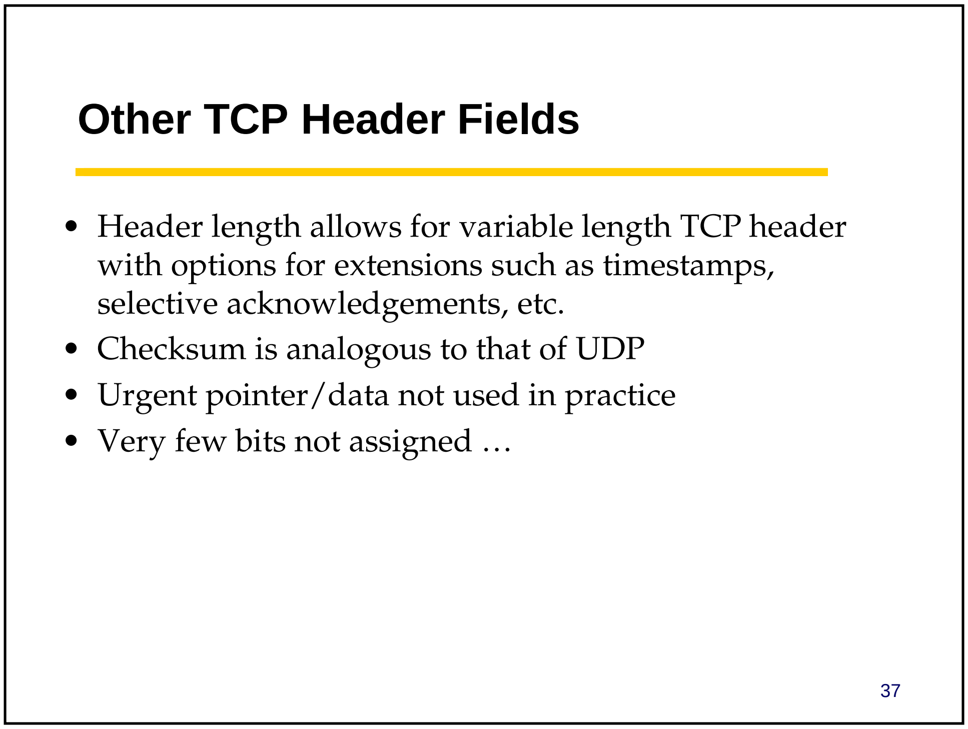
Acknowledgment

HdrLen | 0 | Flags | AdvertisedWindow

Checksum | UrgPtr

Options (variable)

Data



Other TCP Header Fields

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