

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>

Application
Presentation
Session
Transport
Network
Data Link
Physical

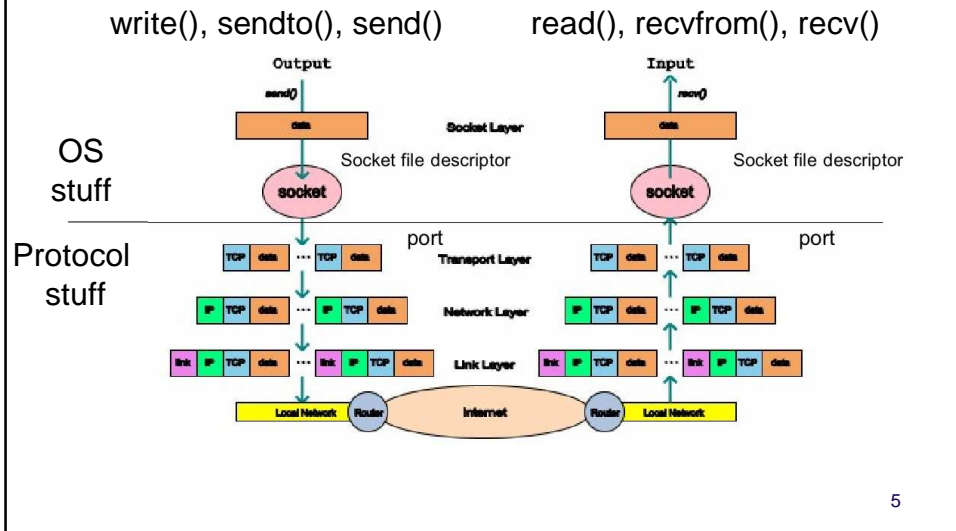
3

Naming Processes/Services

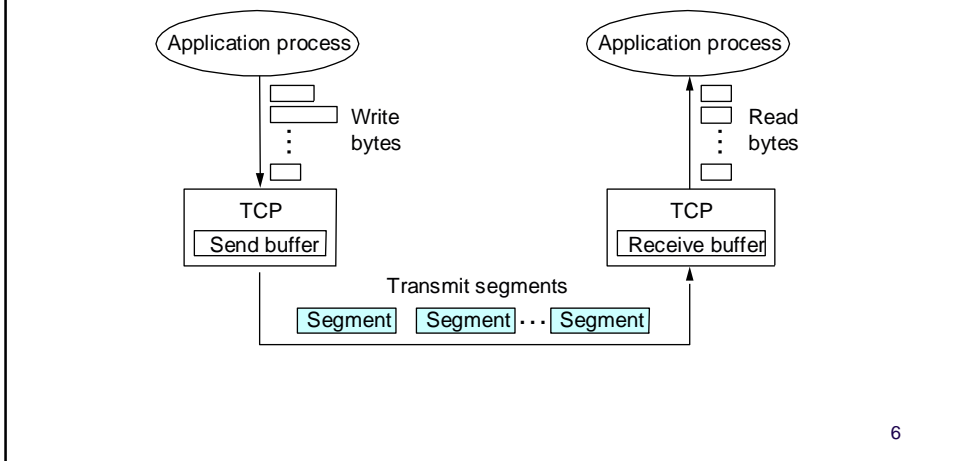
- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.
- How do we identify for remote communication?
 - Process id or memory address are OS-specific and transient
- So TCP and UDP use Ports
 - 16-bit integers representing 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

4

Processes as Endpoints



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

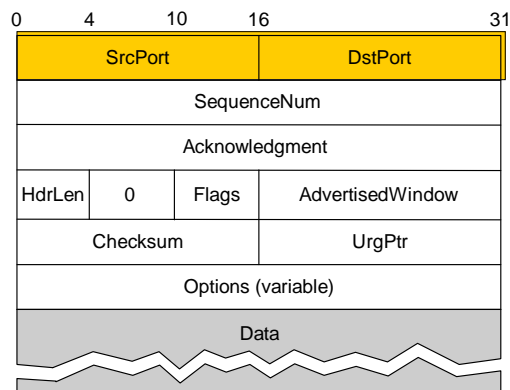
Berkeley Sockets

- 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.

9

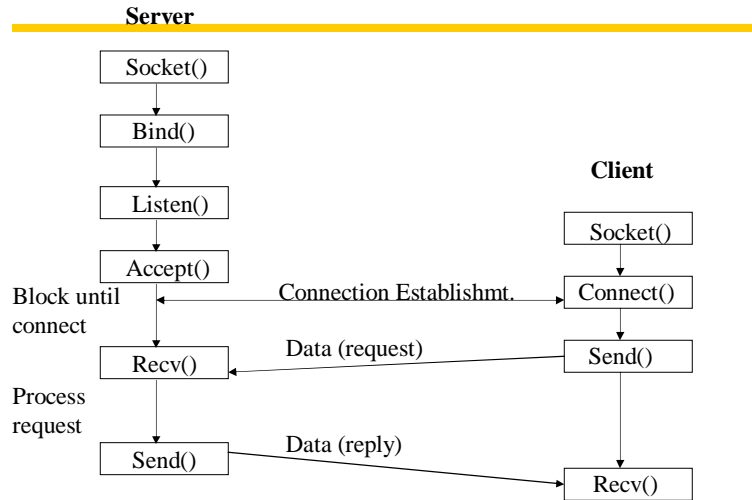
TCP Header Format

- Ports plus IP addresses identify a connection



10

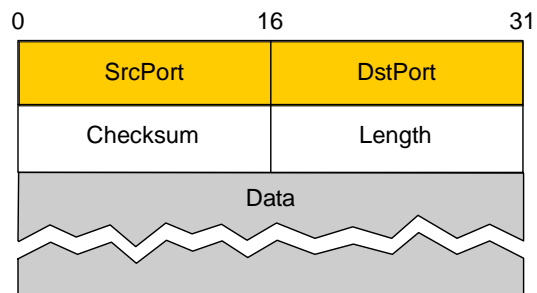
Connection-oriented example (TCP)



11

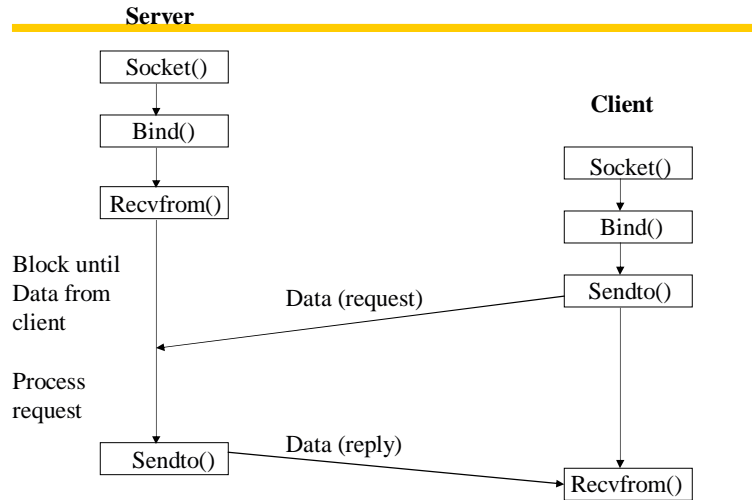
User Datagram Protocol (UDP)

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



12

Connectionless example (UDP)



13

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

14

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

15

Listen call

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

16

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

17

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

18

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”

19

Sample Code

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