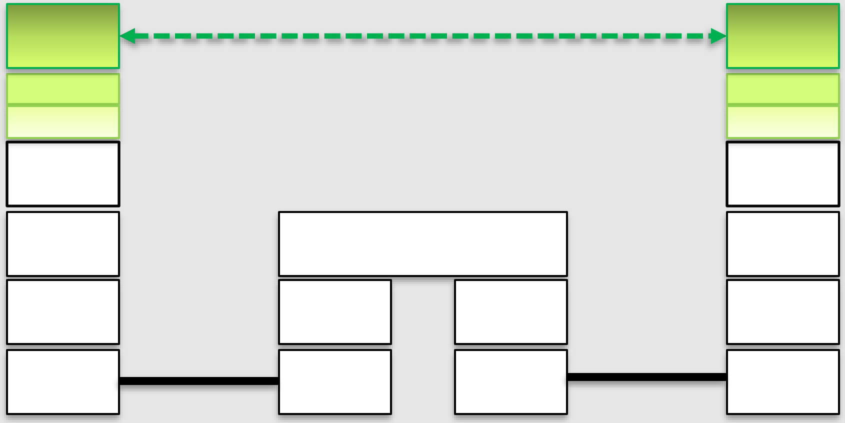
Applications – Socket Communication

Applications – Socket Communication

Application Layer

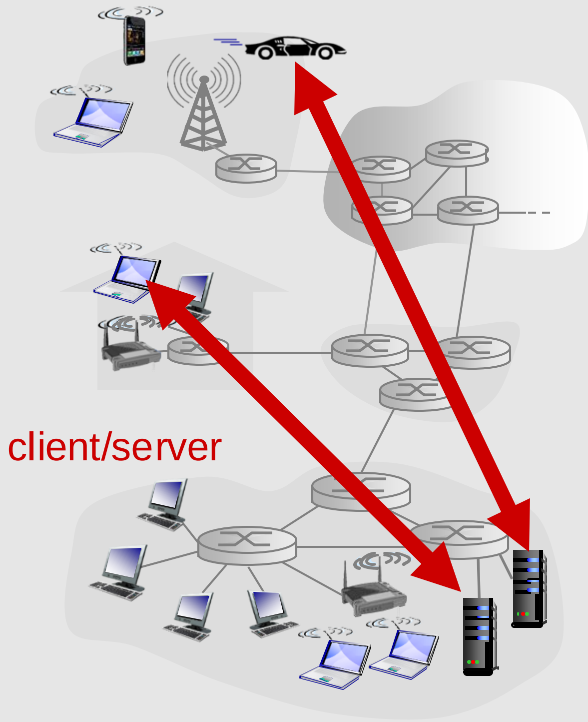
- What it's all about
- Articulated significantly after early networking developments
  - Early networks were more oriented to distributed computing
  - Later articulated as reusable across multiple applications
- Some functionality repeated by every application (Internet model)
- Enabled by UDP and TCP transport layers



Application Program

- An application program embodies algorithms, for some particular data processing needs
  - Distributed application – executed jointly at multiple location
  - Must intercommunicate
- Application program must solve rendezvous problem
  - How to discover "opposite number" i.e. communication target
- Must also create and realize application protocol
  - APDU
- Must coordinate computation
  - Checkpointing, synchronization (if has states)
- Basic joint computing model: Client/Server
  - Other models (P2P, PubSub) can be seen as generalization

Client-Server Architecture



server:

- always-**ON** host
- permanent IP address
- data centers for scaling

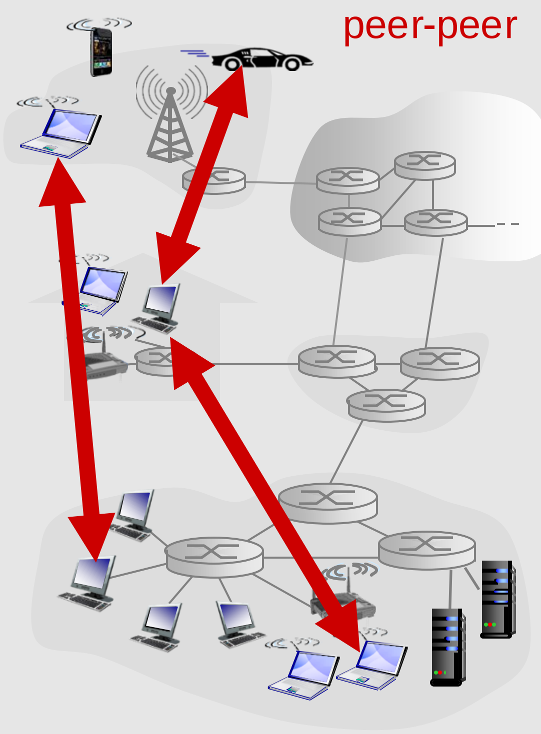
clients:

- Initiates connection to the server
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Classic Example: Web server and Web browsers

Peer-to-Peer Architecture

- *no* always-ON server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *Self-scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management, security



- Another view: each peer is both a client and a server (program)!

Examples: BitTorrent, Skype (hybrid), Internet Telephony

Clients vs. Servers

- Who takes the first step?
  - clients initiate peer-to-peer communications → *active* open
  - servers wait for incoming communication requests → *passive* open
- Lifetimes
  - servers start execution before interaction begins, and continue to accept and process requests “forever”
  - clients are short-lived
- Ports
  - servers wait for requests at a well-known port reserved for the service offered
  - clients use arbitrary, non-reserved ports for communication

Clients vs. Servers (cont’d)

- Complexity of clients
  - no special privileges required (usually)
  - overall, relatively easy to build
- Complexity of servers
  - require special system privileges (usually)
  - may need to handle multiple requests concurrently
  - must protect themselves against all possible errors
  - generally more difficult to build
- Both are usually implemented as application programs

Privileges and Complexity

- Because of their privileged status, servers must contain code that handles...
  1. Authentication (who you're talking to)
  2. Authorization (what can the user do)
  3. Data security (sensitive information)
  4. Privacy (unknown to other users)
  5. Protection (protect its own resources, DoS)

Connectionless vs. Connection-Oriented Servers

- Generally
  - "connectionless" = UDP
  - "connection-oriented" = TCP
  - critical difference = level of reliability provided
- Use of UDP recommended only when...
  - application is non-critical (occasional failure is OK), or
  - application handles reliability, or
  - application uses hardware broadcast/multicast, or
  - application cannot tolerate the overhead of TCP (IoT, tiny sensors)

Iterative ("one-at-a-time") Servers

- Request handling
  1. wait for a client request to arrive
  2. process the client request
  3. send the response back to the client
  4. go back to step 1
- OK for services such as Echo
  - short, single response
  - easy to program!
- But if Step 2 takes a while, other incoming requests may be lost
  - OS may front (as transport service) for some
  - Still, clients have to wait (in queue)

Concurrent Servers

- "Master" process responsible for *accepting* requests, "Slave" processes responsible for *handling* requests
  1. Master waits for a client request to arrive
  2. Master starts a new slave process to handle this request
  3. Master returns to Step 1
- Slave executes concurrently, terminates independently when finished
- *Efficient*, *no waiting*, but harder to program
- TCP is geared for this – makes it easier

TCP Connection States

CLOSED

LISTEN

SYN_RCVD

SYN_SENT

ESTABLISHED

CLOSE_WAIT

FIN_WAIT_1

CLOSING

LAST_ACK

FIN_WAIT_2

TIME_WAIT

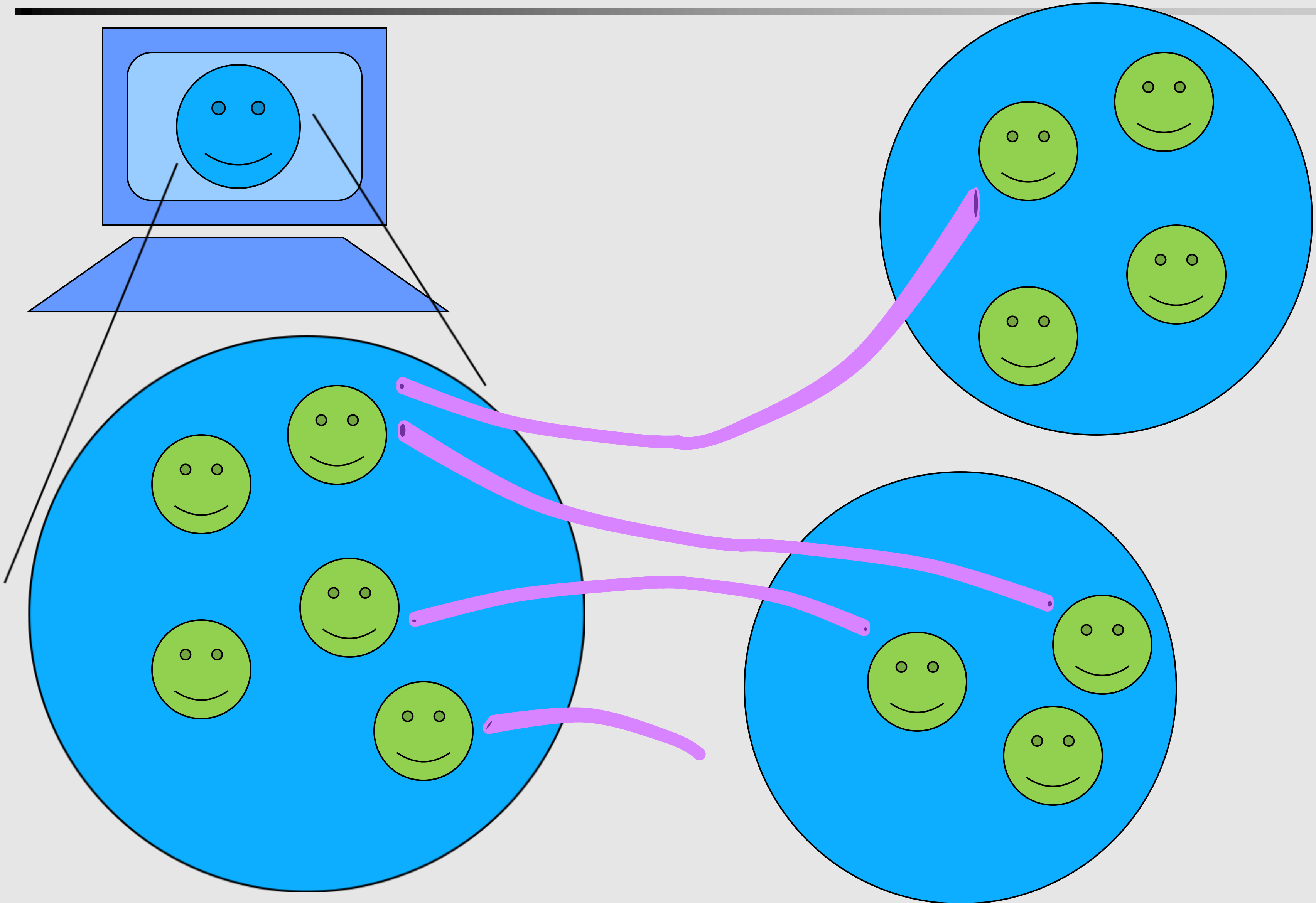
Server Deadlock Problem

- Server can be subject to deadlock if...
  - client never sends a request, server may block in a call to read() forever
  - or, client sends a sequence of requests but never reads responses
- For concurrent servers, only the slave handling the request from misbehaving client will block
- For single-process implementation, the central server process will block
  - will stop handling other connections

Types of Servers

	Connectionless	**Connection-Oriented**
Iterative (one-at-a-time)	Iterative Connectionless (*normal* UDP)	Iterative Connection-Oriented (*less common*)
Con-current	Concurrent Connectionless (*uncommon*)	Concurrent Connection-Oriented (*normal* TCP)

Transport Endpoints

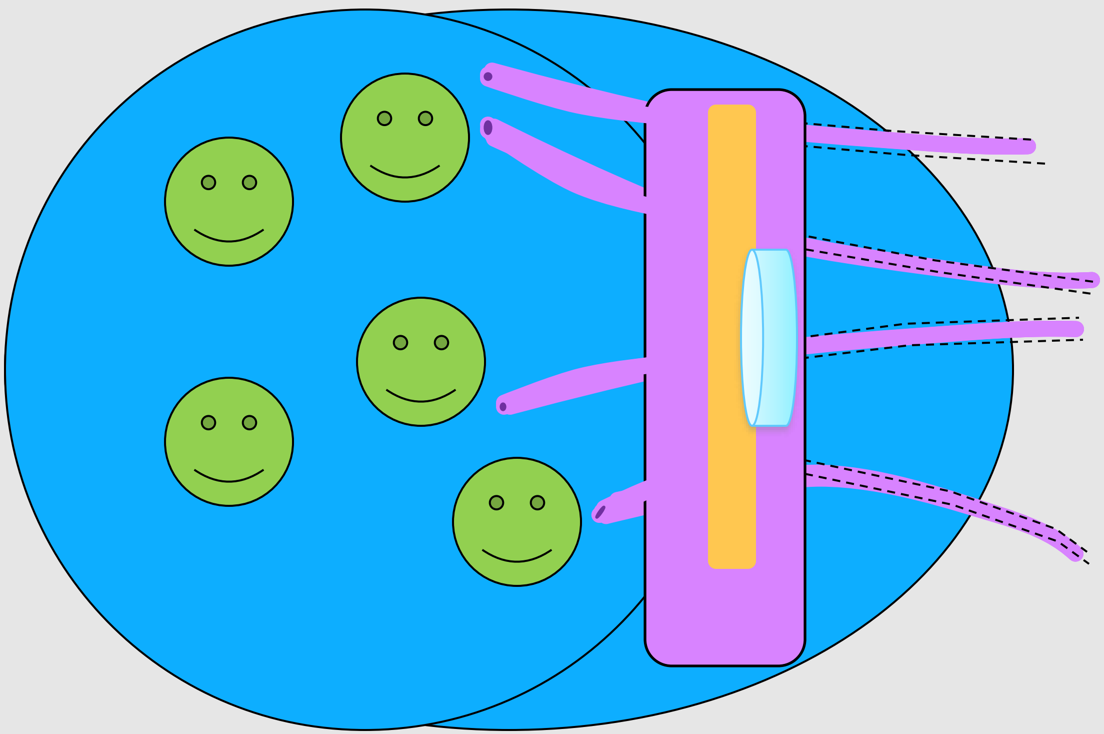


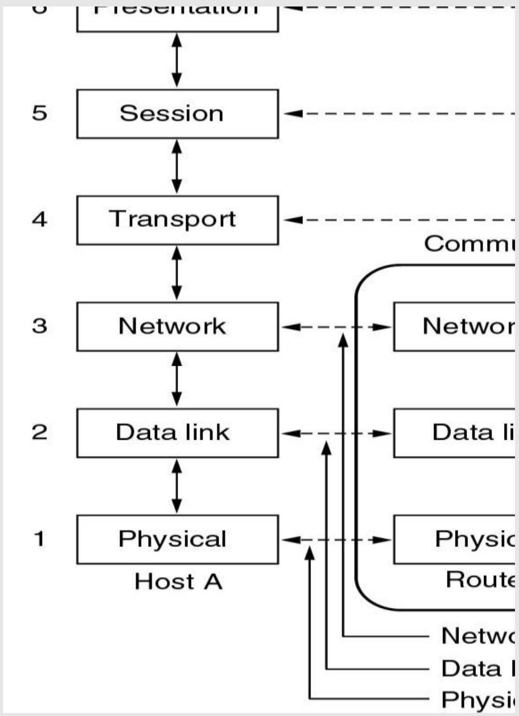
Sockets

- Interface between TCP/IP and applications only loosely specified
- The standards suggest the functionality (not the details)
- Need to programmatically
  - Allocate resources for communication
  - Specify communication endpoints
  - Initiate a connection (client) or wait for incoming connection (server)
  - Send or receive data
  - Terminate a connection gracefully
  - Handle error conditions, etc.

Endpoint Access

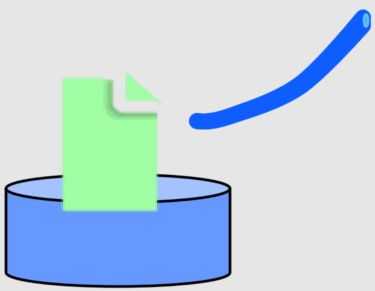
- Transport software built in two parts
  - Host specific part - multiplexes network layer, global context
  - Application specific part - maintains flow state and provides network
    - Also provides access point for higher layers (sockets)

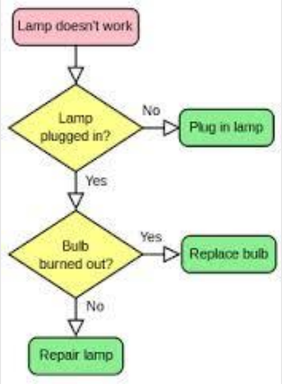




Model: UNIX File i/o

- Program calls `open()` to initiate input or output
  - returns a file descriptor (small, positive integer)
- Calls to `read()` or `write()` to transfer data
  - with the file descriptor as an argument
- Once I/O operations are completed, the program calls `close()`
- Other relevant system calls: `lseek()`, `ioctl()`
  - (these are not used in the Sockets API)

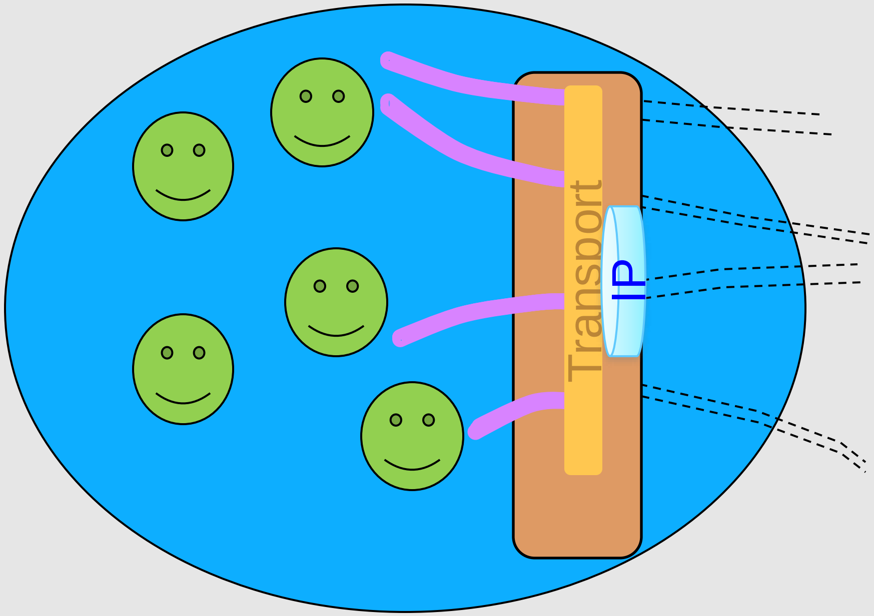




The Socket Abstraction

- Provides an endpoint for communication
  - Also provides buffering
  - Sockets are not bound to specific addresses at the time of creation
- Identified by small integer, the socket descriptor
  - Handle : program refers to it by this number, but programmer need not care
- Flexibility
  - Basic functions can be provided many different transport protocols, others specific to transport
  - Programmer specifies by type of service
  - A generic address structure

Popular Transports



- UDP
  - Create communication endpoints, multiplex
  - Datagrams (contextless, connectionless)
- TCP
  - Create communication endpoints, multiplex
  - Reliability (retransmission)
  - Congestion-sensitive (rate control)
  - Requires context (connection), buffering: “segments”
  - Stream orientation

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - UDP (connectionless)
  - TCP (connection-oriented)
- Each socket has five components: protocol, local/dest IP, local/dest port

"Create socket", "close socket"

"listen from any", "send to specific"

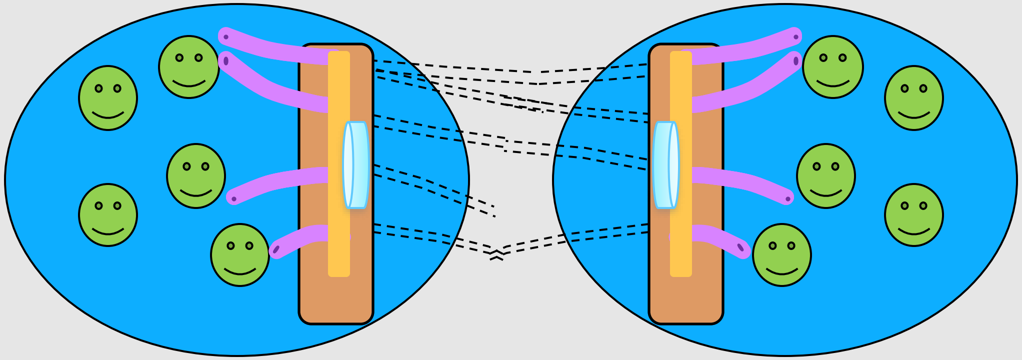
"blocking read", "non-blocking check"

"send to destination", "send to connected socket" (UDP/TCP)

"remember (bind) local address", "remember other-side address", "connect to other-side address"

Elements of a Socket

- Each socket represents a data structure
- Transport protocol provides corresponding programming library (nowadays in kernel already)
  - When a socket is (requested to be) created by an application process, corresponding transport module called
    - Therefore socket must have protocol field
  - Must distinguish between multiple sockets of multiple processes
    - Therefore a “port” number
  - Must be able to stamp identity of host into packet
    - Therefore the host IP address (or the NIC IP address)



Endpoint Addresses

- Each socket association has five components
  - protocol
  - local address
  - local port
  - remote address
  - remote port
- "Create socket" API calls
- "Bind to port" API calls
- "Send to IP/port" API calls (to socket or datagram)
- Caution: example code: need to familiarize yourself with *your* programming environment, map back to concept
  - `man` pages
  - `--help` options
  - `howto` documents

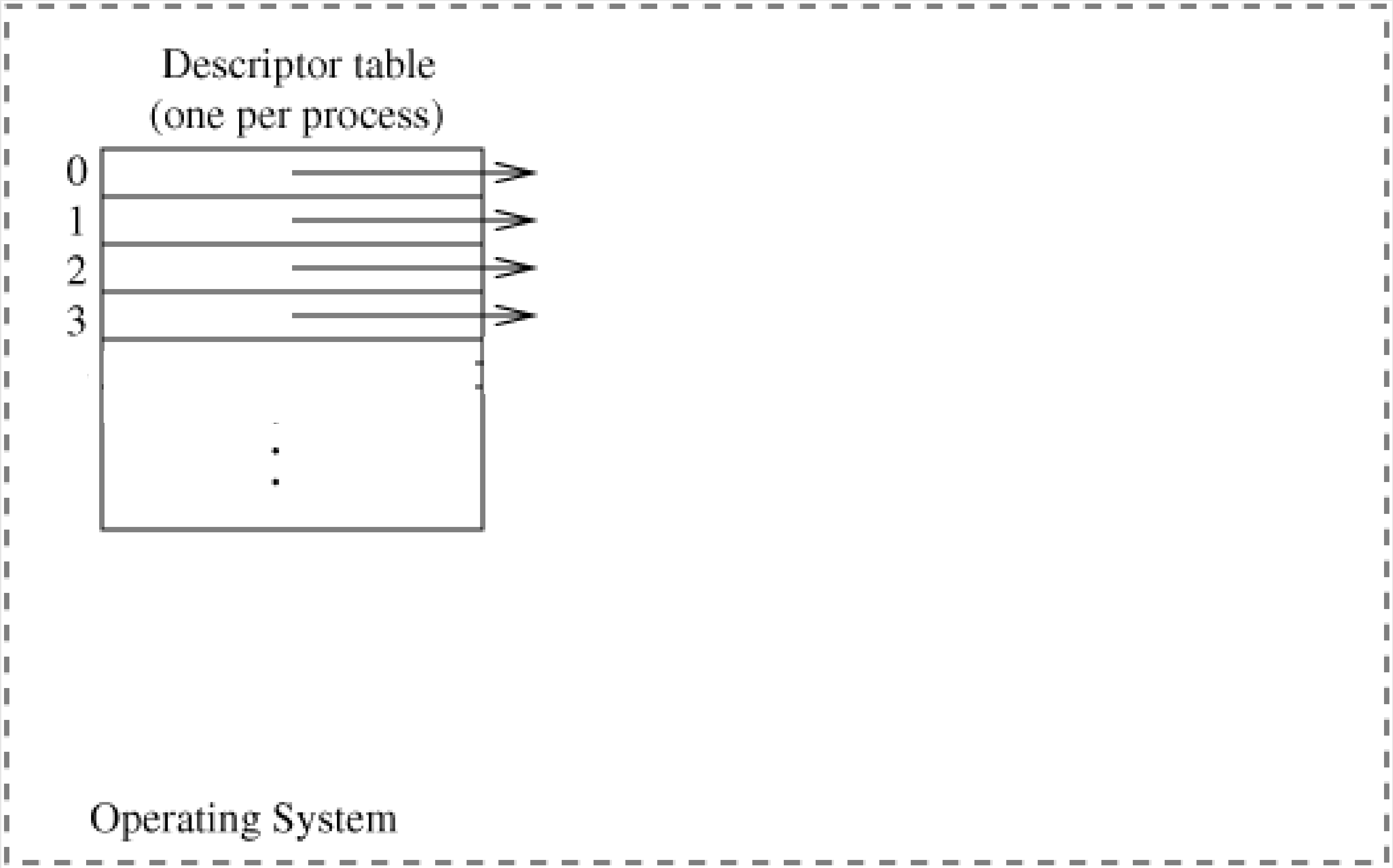
Creating A Socket

```
int s =  socket(domain, type, protocol);
```

- Parameters
  - domain: PF_INET
  - type: SOCK_DGRAM, SOCK_STREAM, SOCK_RAW
  - protocol: usually = 0 (i.e., default for type)
- Example

```
#include <sys/types.h>
#include <sys/socket.h>
…
if ((s = socket(PF_INET, SOCK_STREAM, 0) < 0)
      perror("socket");
```

After Creating A Socket



Generic Address Structure

- Each protocol family defines its own address representation
- For each protocol family there is a corresponding address family
  - (e.g., PF_INET → AF_INET, PF_UNIX → AF_UNIX)
- Generalized address format:
  - <address family, endpoint address in family>

```
struct  sockaddr {
    u_char  sa_len;         /* total length */
    u_short sa_family;      /* type of address  */
    char    sa_data[14];    /* value of address */
};
```

Socket Addresses, Internet Style

```
#include <netinet/in.h>
struct in_addr {
      u_long s_addr;          /* 32-bit host address */
};
struct sockaddr_in {
      u_char         sin_len;      /* total length */
      short          sin_family;   /* AF_INET */
      u_short        sin_port;     /* network byte order */
      struct in_addr sin_addr;     /* network address */
      char           sin_zero[8];  /* unused */
};
```

```
struct sockaddr {
      u_char  sa_len;      /* total length */
      u_short sa_family;   /* type of address */
      char    sa_data[14]; /* value of address */
};
```

Endpoint Addresses

- Each socket association has five components
  - protocol
  - local address }
  - local port
  - remote address }
  - remote port
- "Create socket" API calls
- "Bind to port" API calls
- "Send to IP/port" API calls (to socket or datagram)
- Caution: example code: need to familiarize yourself with *your* programming environment, map back to concept
  - `man` pages
  - `--help` options
  - `howto` documents

Binding the Local Address

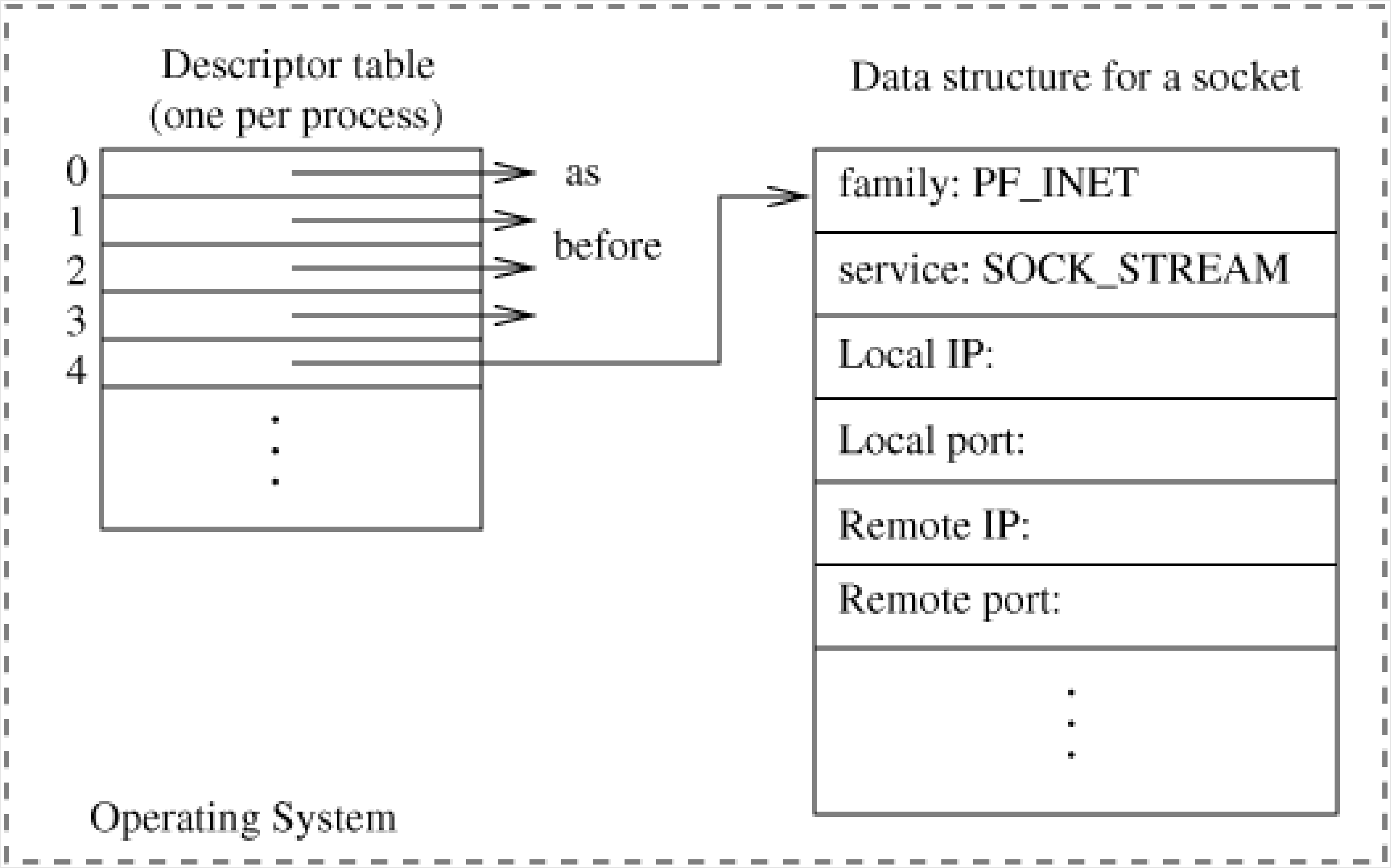
```
int  bind(int s, struct sockaddr *addr, int addrlen);
```

- Used primarily by servers to specify their well-known port
- Optional for clients
  - normally, system chooses a “random” local port
- Use INADDR_ANY to allow the socket to receive datagrams sent to *any* of the machine's IP addresses

Binding the Local Address (cont'd)

```
…
sin.sin_family        = AF_INET;
sin.sin_port          = htons(6000); /* if 0:system
   chooses */
sin.sin_addr.s_addr = INADDR_ANY;  /* allow any
   interface */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
```

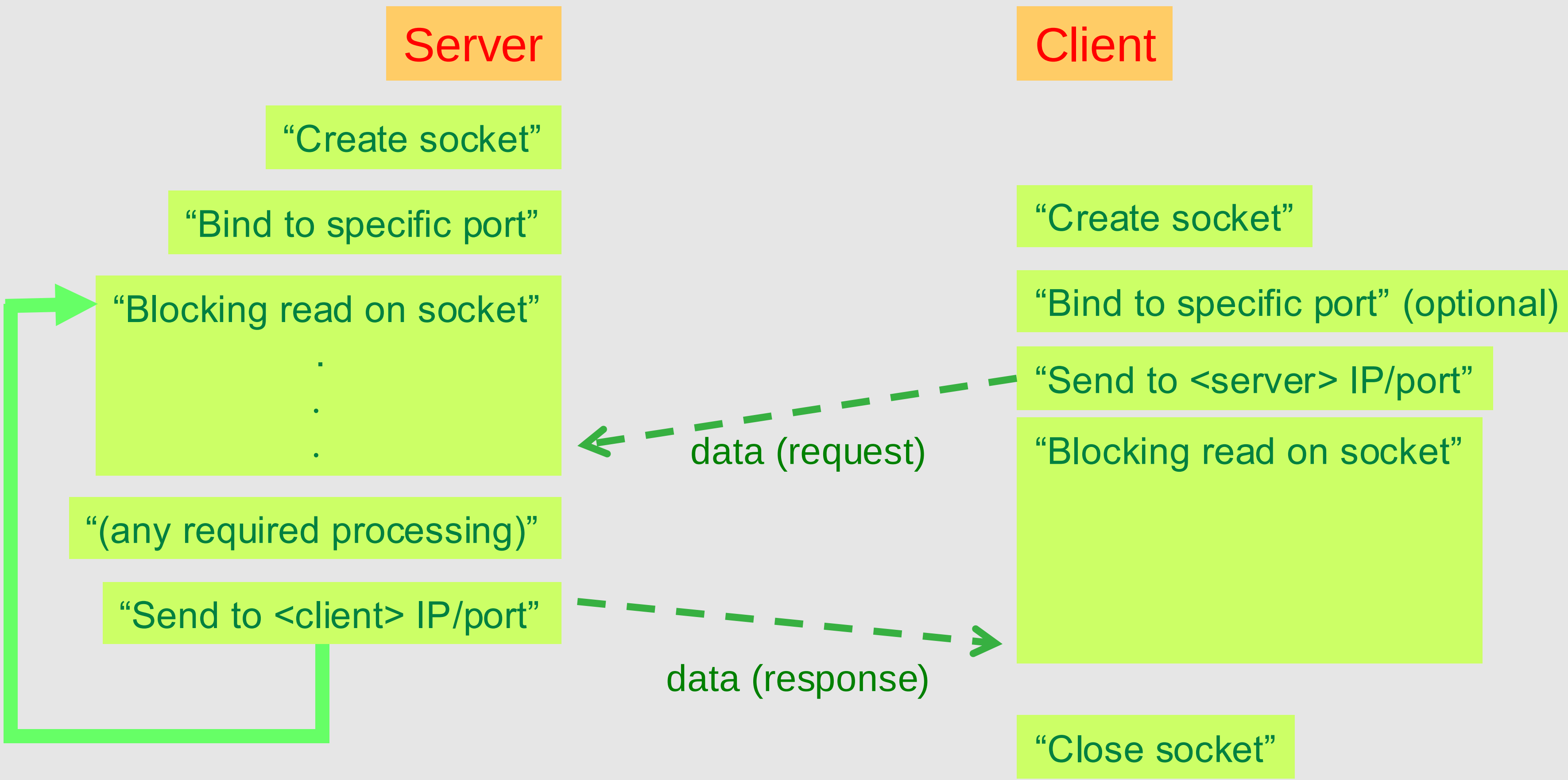
After Binding the Local Address



UDP Sockets – Logical Interaction

	Connectionless	**Connection-Oriented**
Iterative (one-at-a-time)	Iterative Connectionless (*normal* UDP)	Iterative Connection-Oriented (*less common*)
Con-current	Concurrent Connectionless (*uncommon*)	Concurrent Connection-Oriented (*normal* TCP)

UDP Sockets – Logical Interaction



Sample UDP Client (C)

Create
address data structure

Create space for
input and received strings

"Create socket"

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char**argv)
{
   int sockfd,n;
   struct sockaddr_in servaddr,cliaddr;
   char sendline[1000];
   char recvline[1000];

   if (argc != 2)
   {
      printf("usage:  udp-client <IP address>\n");
      exit(1);
   }

   sockfd=socket(AF_INET,SOCK_DGRAM,0);
```

Sample UDP Client (C)

Initialize
address data structure

"Send to IP/port"

"Receive datagram
from socket"

```
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr=inet_addr(argv[1]);
servaddr.sin_port=htons(32000);

while (fgets(sendline, 10000,stdin) != NULL)
{
   sendto(sockfd,sendline,strlen(sendline),0,
          (struct sockaddr *)&servaddr,sizeof(servaddr));
   n=recvfrom(sockfd,recvline,10000,0,NULL,NULL);
   recvline[n]=0;
   fputs(recvline,stdout);
}
}
```

Sample UDP Server (C)

Create
address data structure

"Create socket"

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char**argv)
{
   int sockfd,n;
   struct sockaddr_in servaddr,cliaddr;
   socklen_t len;
   char mesg[1000];

   sockfd=socket(AF_INET,SOCK_DGRAM,0);
```

Sample UDP Server (C)

Initialize address data structure

"Bind to port"

"Receive datagram from socket"

"Send to IP/port"

```
bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  servaddr.sin_port=htons(32000);
  bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));

  for (;;)
  {
     len = sizeof(cliaddr);
     n = recvfrom(sockfd,mesg,1000,0,(struct sockaddr *)&cliaddr,&len);
     sendto(sockfd,mesg,n,0,(struct sockaddr *)&cliaddr,sizeof(cliaddr));
     printf("-------------------------------------------------------\n");
     mesg[n] = 0;
     printf("Received the following:\n");
     printf("%s",mesg);
     printf("-------------------------------------------------------\n");
  }
}
```