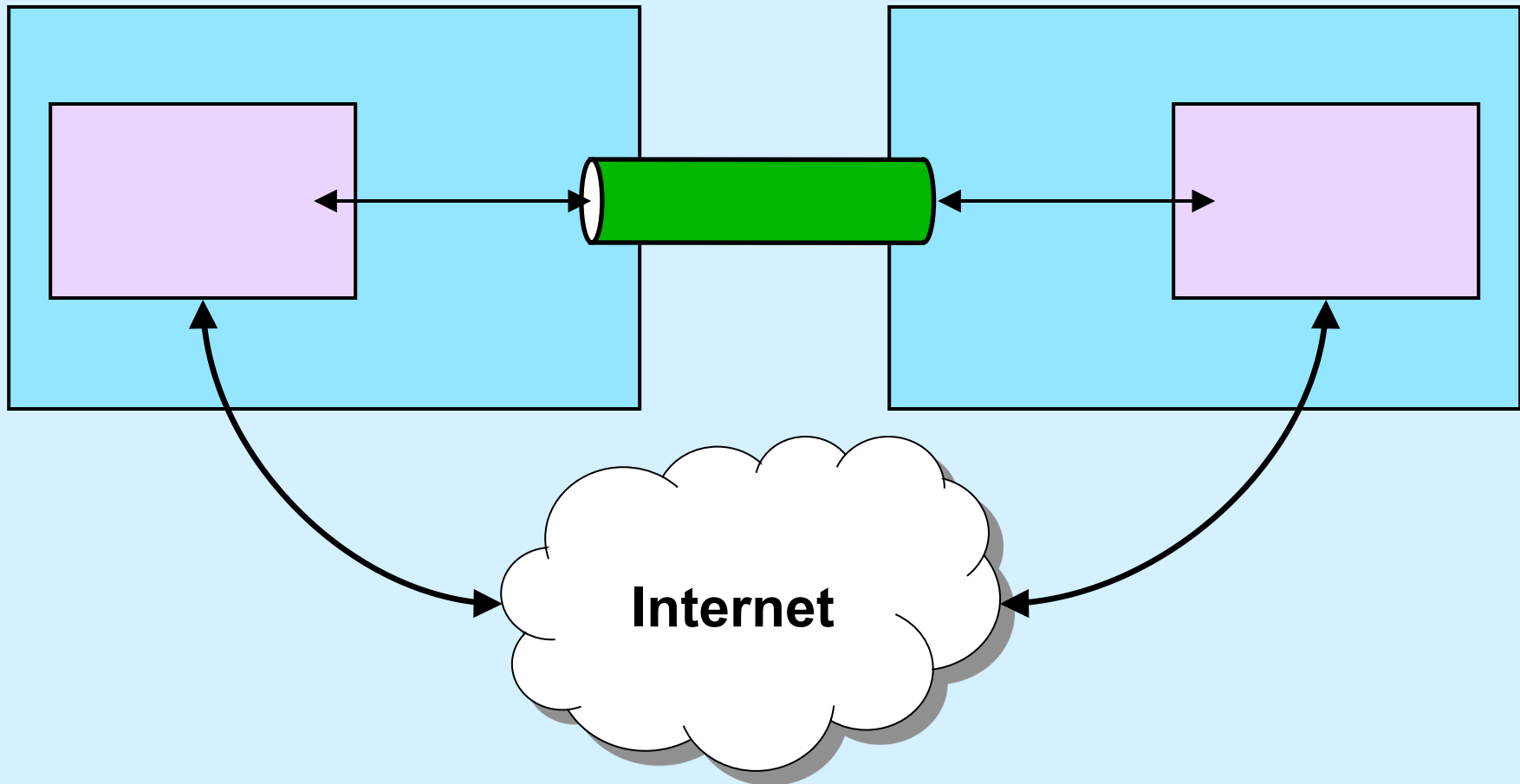


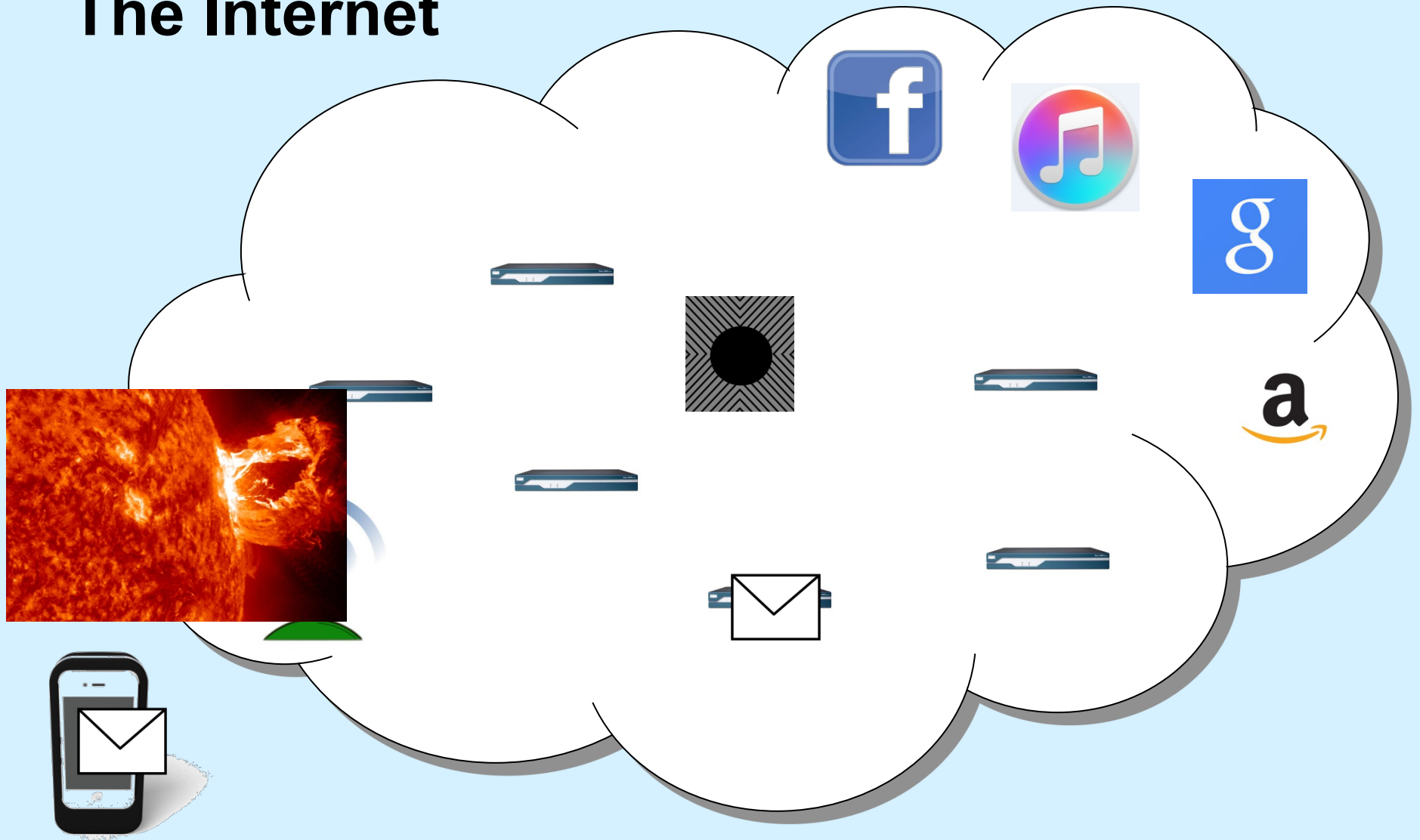
CS 33

Network Programming

Communicating Over the Internet



The Internet



Reliability

- **Two possibilities**
 - **don't worry about it**
 - » **just send it**
 - **if it arrives at its destination, that's good!**
 - **no verification**
 - **worry about it**
 - » **keep track of what's been successfully communicated**
 - » **retransmit until**
 - **data is received**
 - or
 - **it appears that “the network is down”**

Reliability vs. Unreliability

- **Good vs. evil**
 - (but which is which?)
 - **Reliable communication**
 - good for
 - » email
 - » texting
 - » distributed file systems
 - » web pages
 - bad for
 - » streaming audio
 - » streaming video
- } a little noise is better than a long pause

The Data Abstraction

- **Byte stream**
 - sequence of bytes
 - » as in pipes
 - any notion of a larger data aggregate is the responsibility of the programmer
- **Discrete records**
 - sequence of variable-size “records”
 - boundaries between records maintained
 - receiver receives discrete records, as sent by sender

What's Supported

- **Stream**
 - byte-stream data abstraction
 - reliable transmission
- **Datagram**
 - discrete-record data abstraction
 - unreliable transmission

Quiz 1

The following code is used to transmit data over a reliable byte-stream communication channel. Assume `sizeof(record_t)` is large.

```
// sender
record_t data=getData();
write(fd, &data,
    sizeof(data));
```

```
// receiver
read(fd, &data,
    sizeof(data));
useData(data);
```

Does it work?

- a) always, assuming no network problems
- b) sometimes
- c) never

What We'd Like ...

```
int fd = open("remote party", stream|O_RDWR);  
write(fd, request, request_size);  
read(fd, response, sizeof(response));
```

- **But ...**
 - **remote party is an active participant**
 - **must agree to participate**
 - **must support multiple styles of communication**
 - **must handle various sorts of errors**

Client-Server Interaction

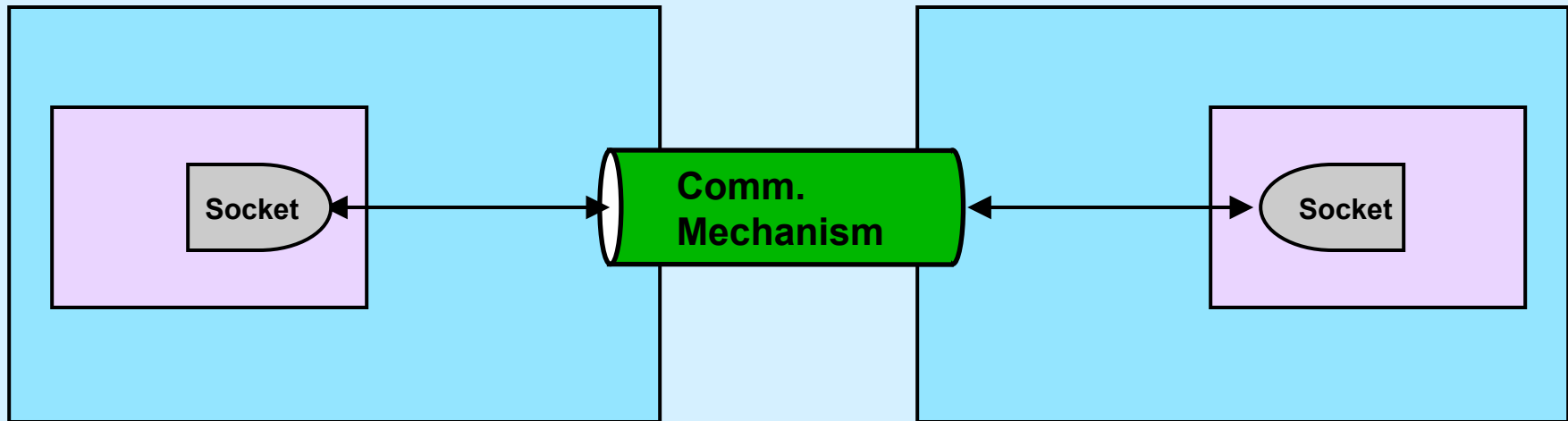
- **Client sends requests to server**
- **Server responds**
- **Server may deal with multiple clients at once**
- **Client may contact multiple servers**

Sockets



**Communication abstraction
endpoint of communication path
referenced via file descriptor**

Sockets



Socket Parameters

- **Styles of communication:**
 - **stream: reliable, two-way byte streams**
 - **datagram: unreliable, two-way record-oriented**
 - **and others, if needed**
- **Communication domains**
 - **UNIX**
 - » **endpoints (sockets) named with file-system pathnames**
 - » **supports stream and datagram**
 - » **trivial protocols: strictly for intra-machine use**
 - **Internet**
 - » **endpoints named with IP addresses**
 - » **supports stream and datagram**
 - **others**
- **Protocols**
 - **the means for communicating data**
 - **e.g., TCP/IP, UDP/IP**

Setting Things Up

- **Socket (communication endpoint) is given a name**
 - *bind* system call
 - **Datagram communication**
 - use *sendto* system call to send data to named recipient
 - use *recvfrom* system call to receive data and name of sender
 - **Stream communication**
 - client connects to server
 - » server uses *listen* and *accept* system calls to receive connections
 - » client uses *connect* system call to make connections
 - data transmitted using *send* or *write* system calls
 - data received using *recv* or *read* system calls
-

Datagrams in the UNIX Domain (1)

- **Steps**

- 1) **create socket**

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

```
fd = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Datagrams in the UNIX Domain (2)

2) set up name

```
struct sockaddr_un {  
    short sun_family;      /* AF_UNIX */  
    char sun_path[108];  /* path name */  
} name;
```

```
name.sun_family = AF_UNIX;  
memcpy(name.sun_path, path, strlen(path));
```


Datagrams in the UNIX Domain (3)

3) bind name to socket

```
name_len = sizeof(name.sun_family) +  
           strlen(name.sun_path);  
bind(fd, (struct sockaddr *) &name,  
      name_len);
```

Datagrams in the UNIX Domain (4)

4) send data

```
ssize_t sendto(int fd, const void *buf,  
              ssize_t len, int flags,  
              const struct sockaddr *to,  
              socklen_t to_len);
```

```
struct sockaddr_un to_name;  
socklen_t to_len = sizeof(to_name);
```

```
sendto(fd, buf, sizeof(buf), 0,  
       (struct sockaddr *)&to_name,  
       to_len);
```

Datagrams in the UNIX Domain (5)

5) receive data

```
ssize_t recvfrom(int s, void *buf,  
                ssize_t len,  
                int flags, struct sockaddr *from,  
                socklen_t *from_len);
```

```
struct sockaddr_un from_name;  
int from_len = sizeof(from_name);
```

```
recvfrom(fd, buf, sizeof(buf), 0,  
         (struct sockaddr *)&from_name,  
         &from_len);
```

UNIX Datagram Example (1)

- **Server side**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define NAME "/home/twd/server"

main( ) {
    struct sockaddr_un    sock_name;
    int fd, len;
    /* Step 1: create socket in UNIX domain for datagram
       communication.  The third argument specifies the
       protocol, but since there's only one such protocol in
       this domain, it's set to zero */
    if ((fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
}
```

UNIX Datagram Example (2)

```
/* Step 2: set up a sockaddr structure to contain the
   name we want to assign to the socket */
sock_name.sun_family = AF_UNIX;
strcpy(sock_name.sun_path, NAME);
len = strlen(NAME) + sizeof(sock_name.sun_family);

/* Step 3: bind the name to the socket */
if (bind(fd, (struct sockaddr *)&sock_name, len) < 0) {
    perror("bind");
    exit(1);
}
```

UNIX Datagram Example (3)

```
while (1) {
    char buf[1024];
    struct sockaddr_un from_addr;
    int from_len = sizeof(from_addr);
    int msg_size;

    /* Step 4: receive message from client */
    if ((msg_size = recvfrom(fd, buf, 1024, 0,
        (struct sockaddr *)&from_addr, &from_len)) < 0) {
        perror("recvfrom");
        exit(1);
    }
    buf[msg_size] = 0;
    printf("message from %s:\n%s\n", from_addr.sun_path,
        buf);
}
```

UNIX Datagram Example (4)

```
/* Step 5: respond to client */
if (sendto(fd, "thank you", 9, 0,
          (const struct sockaddr *)&from_addr,
          from_len) < 0) {
    perror("sendto");
    exit(1);
}
}
```

UNIX Datagram Example (5)

- **Client Side**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SNAME "/home/twd/server"
#define CNAME "/home/twd/client"

int main( ) {
    struct sockaddr_un server_name;
    struct sockaddr_un client_name;
    int fd, server_len, client_len;
```


UNIX Datagram Example (6)

```
/* Step 1: create socket in UNIX domain for datagram
   communication. The third argument specifies the
   protocol, but since there's only one such protocol
   in this domain, it's set to zero */
if ((fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    exit(1);
}

/* Step 2: set up a sockaddr structure to contain the
   name we want to assign to the socket */
client_name.sun_family = AF_UNIX;
strcpy(client_name.sun_path, CNAME);
client_len = strlen(CNAME) +
    sizeof(client_name.sun_family);
```

UNIX Datagram Example (7)

```
/* Step 3: bind the name to the socket */  
if (bind(fd, (struct sockaddr *)&client_name,  
        client_len) < 0) {  
    perror("bind");  
    exit(1);  
}
```

UNIX Datagram Example (8)

```
/* Step 4: set up server's name */
server_name.sun_family = AF_UNIX;
strcpy(server_name.sun_path, SNAME);
server_len = strlen(SNAME) +
    sizeof(server_name.sun_family);

while (1) {
    char buf[1024];
    int msg_size;

    if (fgets(buf, 1024, stdin) == 0)
        break;
```

UNIX Datagram Example (9)

```
/* Step 5: send data to server */
if (sendto(fd, buf, strlen(buf), 0,
           (const struct sockaddr *) &server_name,
           server_len) < 0) {
    perror("sendto");
    exit(1);
}
```

UNIX Datagram Example (10)

```
/* Step 6: receive response from server */
if ((msg_size = recvfrom(fd, buf, 1024, 0, 0, 0))
    < 0) {
    perror("recvfrom");
    exit(1);
}
buf[msg_size] = 0;
printf("Server says: %s\n", buf);
}
return(0);
}
```

Quiz 2

In the previous slide was

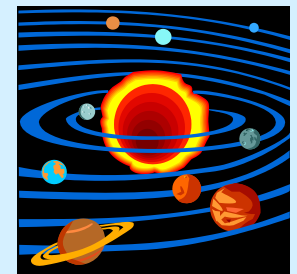
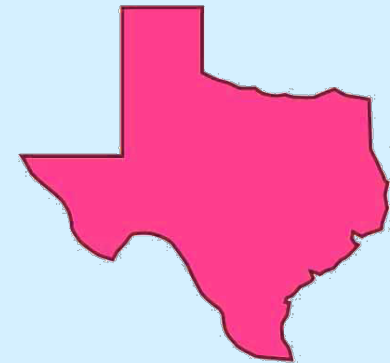
```
recvfrom(fd, buf, 1024, 0, 0, 0);
```

The 0's indicate the caller is not interested in who sent the datagram.

- a) **This makes sense: having sent a datagram to the server, we now have a connection to the server and anything coming back must be from that server.**
- b) **This doesn't make sense: anyone could be sending to the pathname the socket is bound to, and thus the response could be coming from some other sender.**

Internet Addresses

- **IP (internet protocol) address**
 - one per network interface
 - **32 bits (IPv4)**
 - » 5527 per acre of RI
 - » 25 per acre of Texas
 - **128 bits (IPv6)**
 - » 1.6 billion per cubic mile of a sphere whose radius is the mean distance from the Sun to the (former) planet Pluto
- **Port number**
 - one per application instance per machine
 - **16 bits**
 - » port numbers less than 1024 are reserved for privileged applications



Notation

- **Addresses (assume IPv4: 32-bit addresses)**
 - written using dot notation
 - » 128.48.37.1
 - dots separate bytes

Host Names

- **Hosts are referred to by “DNS names”**
 - e.g. `nfs.cs.brown.edu`
- **DNS (Domain Name Service) is a distributed database**
 - translates names to addresses
 - `nfs.cs.brown.edu`
 - » `10.116.110.153`
 - » `10.116.110.154`
 - » `10.116.110.155`
- **The library routine *getaddrinfo* performs DNS lookups**

Some Details ...

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};

my_addr.sin_family = AF_INET;
inet_pton(AF_INET, "10.116.72.109", &my_addr.sin_addr.s_addr);
my_addr.sin_port = htons(port);
```

getaddrinfo()

- `int getaddrinfo(
 const char *node,
 const char *service,
 const struct addrinfo *hints,
 struct addrinfo **res);`

```
struct addrinfo {  
    int          ai_flags;  
    int          ai_family;  
    int          ai_socktype;  
    int          ai_protocol;  
    socklen_t    ai_addrlen;  
    struct sockaddr ai_addr;  
    char         ai_canonname;  
    struct addrinfo *ai_next;  
};
```

Using *getaddrinfo* (1)

```
struct addrinfo hints, **res, *rp;
// zero out hints
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
    // want IPv4
hints.ai_socktype = SOCK_STREAM;
    // want stream communication

getaddrinfo("cslab1a.cs.brown.edu", "3333",
    &hints, &res);
```

Using *getaddrinfo* (2)

```
for (rp = res; rp != NULL; rp = rp->ai_next) {  
    // try each interface till we find one that works  
    if ((sock = socket(rp->ai_family, rp->ai_socktype,  
        rp->ai_protocol)) < 0) {  
        continue;  
    }  
    if (connect(sock, rp->ai_addr, rp->ai_addrlen) >= 0)  
        break;  
    close(sock);  
}  
if (rp == NULL) {  
    fprintf(stderr, "Could not connect to %s\n", argv[1]);  
    exit(1);  
}  
freeaddrinfo(res); // free up storage allocated for list
```

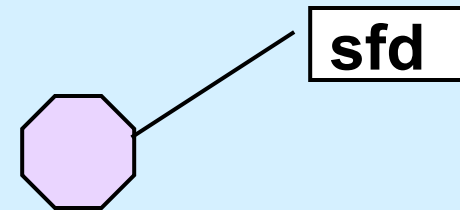
Reliable Communication

- **The promise ...**
 - what is sent is received
 - order is preserved
- **Set-up is required**
 - two parties agree to communicate
 - » each side keeps track of what is sent, what is received
 - » received data is acknowledged
 - » unack'd data is re-sent
- **The standard scenario**
 - server receives connection requests
 - client makes connection requests

Streams in the Inet Domain (1)

- **Server steps**
 - 1) **create socket**

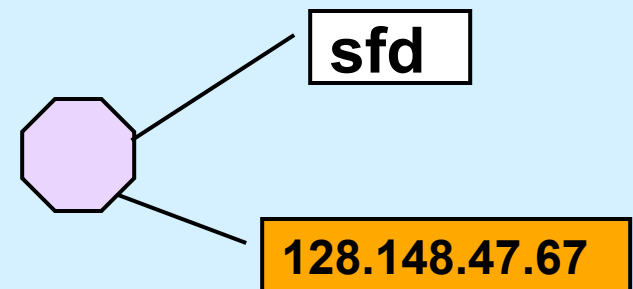
```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (2)

- **Server steps**
 - 2) **bind name to socket**

```
bind(sfd,  
    (struct sockaddr *) &my_addr, sizeof(my_addr));
```



Some Details ...

- **Server may have multiple interfaces; we want to be able to receive on all of them**

```
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];          /* padding */
} my_addr;
```

```
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(port);
```

**“Wildcard”
address**

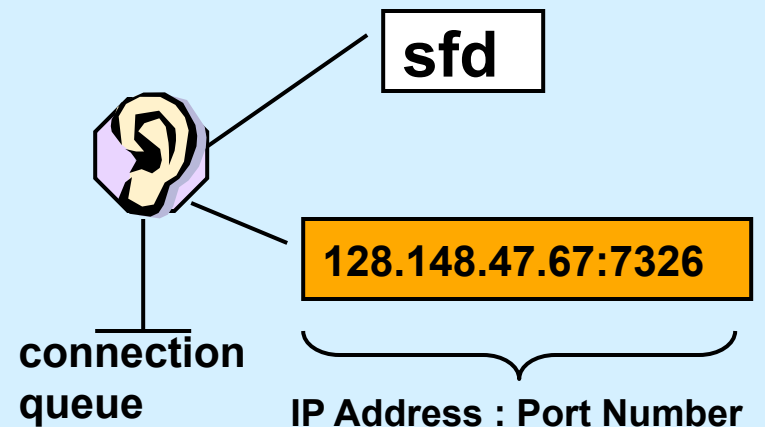


Streams in the Inet Domain (3)

- **Server steps**

- 3) put socket in “listening mode”

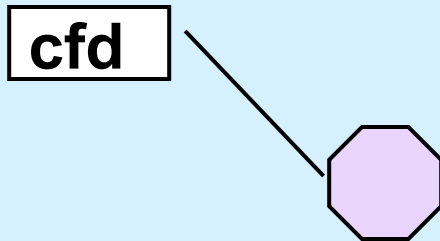
```
int listen(int sfd, int MaxQueueLength);
```



Streams in the Inet Domain (4)

- **Client steps**
 - 1) **create socket**

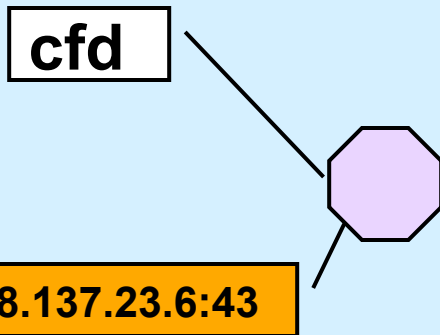
```
cfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (5)

- **Client steps**
 - 2) **bind name to socket**

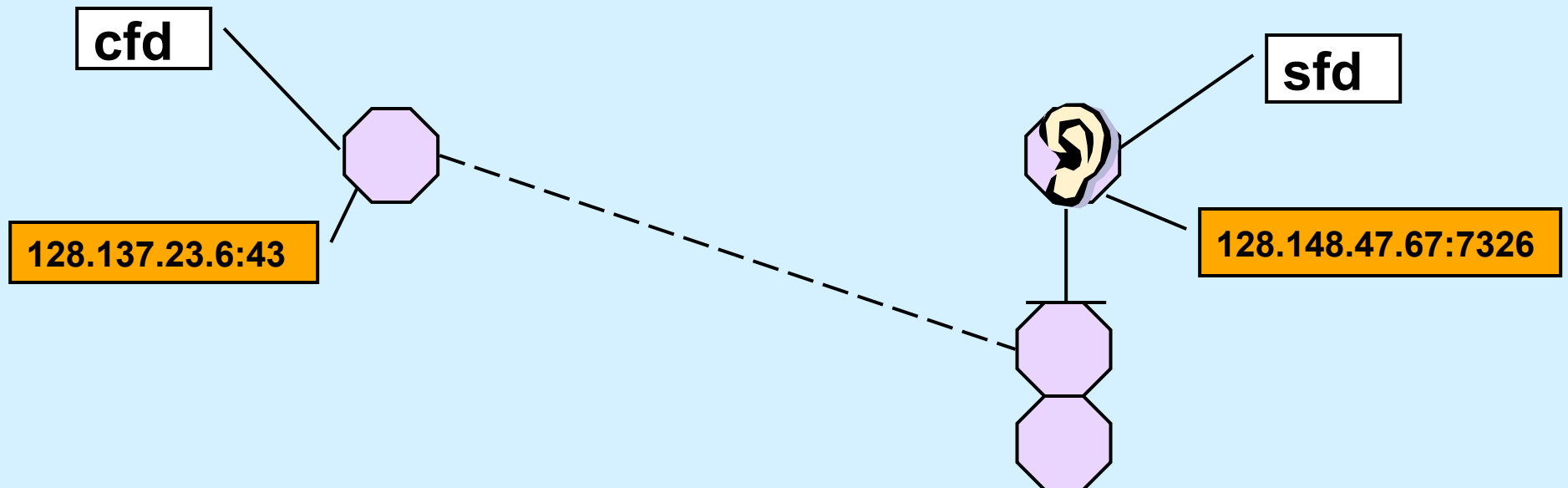
```
bind(cfd,  
     (struct sockaddr *) &my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (6)

- **Client steps**
 - 3) **connect to server**

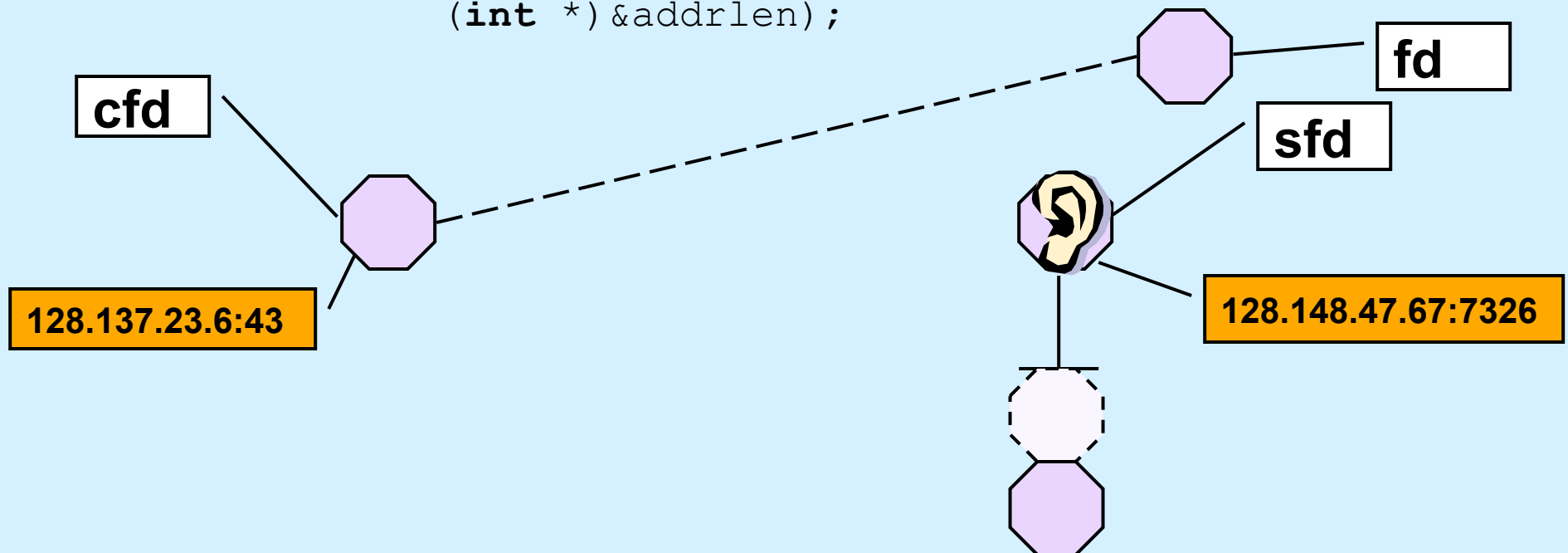
```
connect(cfd, (struct sockaddr *)&server_addr,  
        sizeof(server_addr));
```



Streams in the Inet Domain (6)

- **Server steps**
 - 4) **accept connection**

```
fd = accept((int) sfd, (struct sockaddr *) addr,  
           (int *) &addrlen);
```



Inet Stream Example (1)

- **Server side**

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

int main(int argc, char *argv[ ]) {
    struct sockaddr_in my_addr;
    int lsock;
    void serve(int);
    if (argc != 2) {
        fprintf(stderr, "Usage: tcpServer port\n");
        exit(1);
    }
}
```

Inet Stream Example (2)

```
// Step 1: establish a socket for TCP
if ((lsock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```


Inet Stream Example (3)

```
/* Step 2: set up our address */
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(atoi(argv[1]));

/* Step 3: bind the address to our socket */
if (bind(lsock, (struct sockaddr *)&my_addr,
        sizeof(my_addr)) < 0) {
    perror("bind");
    exit(1);
}
```

Inet Stream Example (4)

```
/* Step 4: put socket into "listening mode" */
if (listen(lsock, 100) < 0) {
    perror("listen");
    exit(1);
}
while (1) {
    int csock;
    struct sockaddr_in client_addr;
    int client_len = sizeof(client_addr);

    /* Step 5: receive a connection */
    csock = accept(lsock,
        (struct sockaddr *)&client_addr, &client_len);
    printf("Received connection from %s#%hu\n",
        inet_ntoa(client_addr.sin_addr), client_addr.sin_port);
}
```

Inet Stream Example (5)

```
switch (fork( )) {
case -1:
    perror("fork");
    exit(1);
case 0:
    // Step 6: create a new process to handle connection
    serve(csock);
    exit(0);
default:
    close(csock);
    break;
}
}
```

Inet Stream Example (6)

```
void serve(int fd) {
    char buf[1024];
    int count;

    // Step 7: read incoming data from connection
    while ((count = read(fd, buf, 1024)) > 0) {
        write(1, buf, count);
    }
    if (count == -1) {
        perror("read");
        exit(1);
    }
    printf("connection terminated\n");
}
```

Inet Stream Example (7)

- **Client side**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
// + more includes ...

int main(int argc, char *argv[]) {
    int s, sock;
    struct addrinfo hints, *result, *rp;

    char buf[1024];
    if (argc != 3) {
        fprintf(stderr, "Usage: tcpClient host port\n");
        exit(1);
    }
}
```

Inet Stream Example (8)

```
// Step 1: find the internet address of the server
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

if ((s=getaddrinfo(argv[1], argv[2], &hints, &result)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(1);
}
```

Inet Stream Example (9)

```
// Step 2: set up socket for TCP and connect to server
for (rp = result; rp != NULL; rp = rp->ai_next) {
    // try each interface till we find one that works
    if ((sock = socket(rp->ai_family, rp->ai_socktype,
        rp->ai_protocol)) < 0) {
        continue;
    }
    if (connect(sock, rp->ai_addr, rp->ai_addrlen) >= 0) {
        break;
    }
    close(sock);
}
if (rp == NULL) {
    fprintf(stderr, "Could not connect to %s\n", argv[1]);
    exit(1);
}
freeaddrinfo(result);
```

Inet Stream Example (10)

```
// Step 3: send data to the server
while (fgets(buf, 1024, stdin) != 0) {
    if (write(sock, buf, strlen(buf)) < 0) {
        perror("write");
        exit(1);
    }
}
return 0;
}
```


Quiz 3

The previous slide contains

```
write(sock, buf, strlen(buf))
```

If data is lost and must be retransmitted

- a) write returns an error so the caller can retransmit the data.**
- b) nothing happens as far as the application code is concerned, the data is retransmitted automatically.**

Quiz 4

The previous slide contains

```
write(sock, buf, strlen(buf))
```

We lose the connection to the other party (perhaps a network cable is cut).

- a) write returns an error so the caller can reconnect, if desired.**
- b) nothing happens as far as the application code is concerned, the connection is reestablished automatically.**