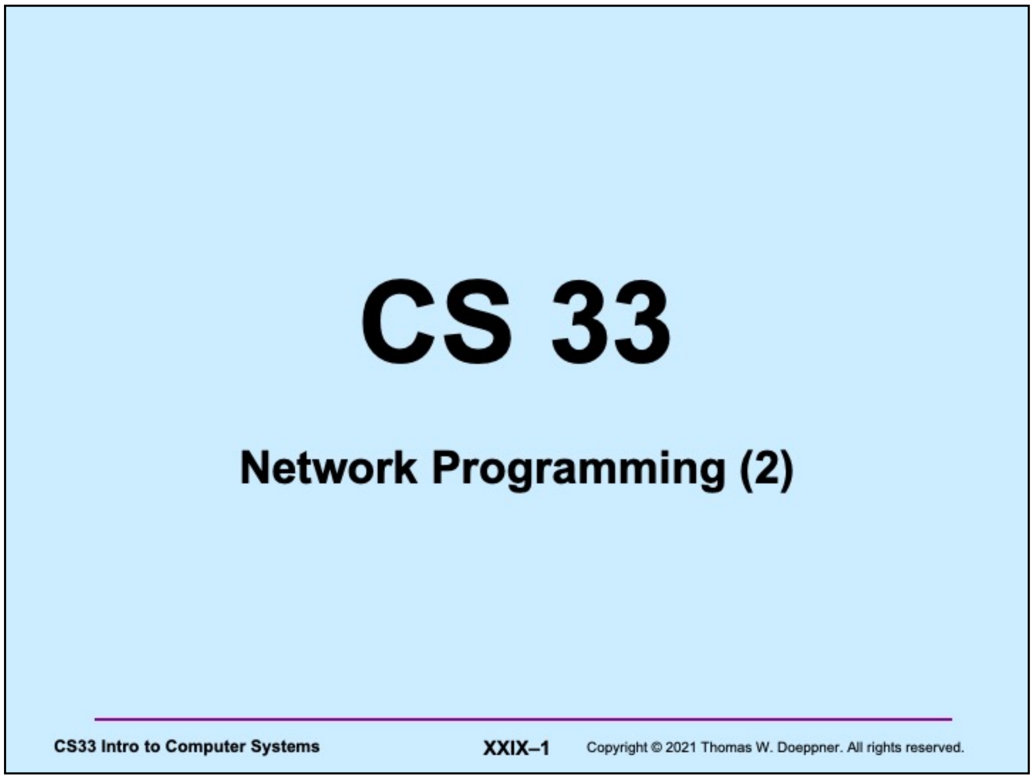
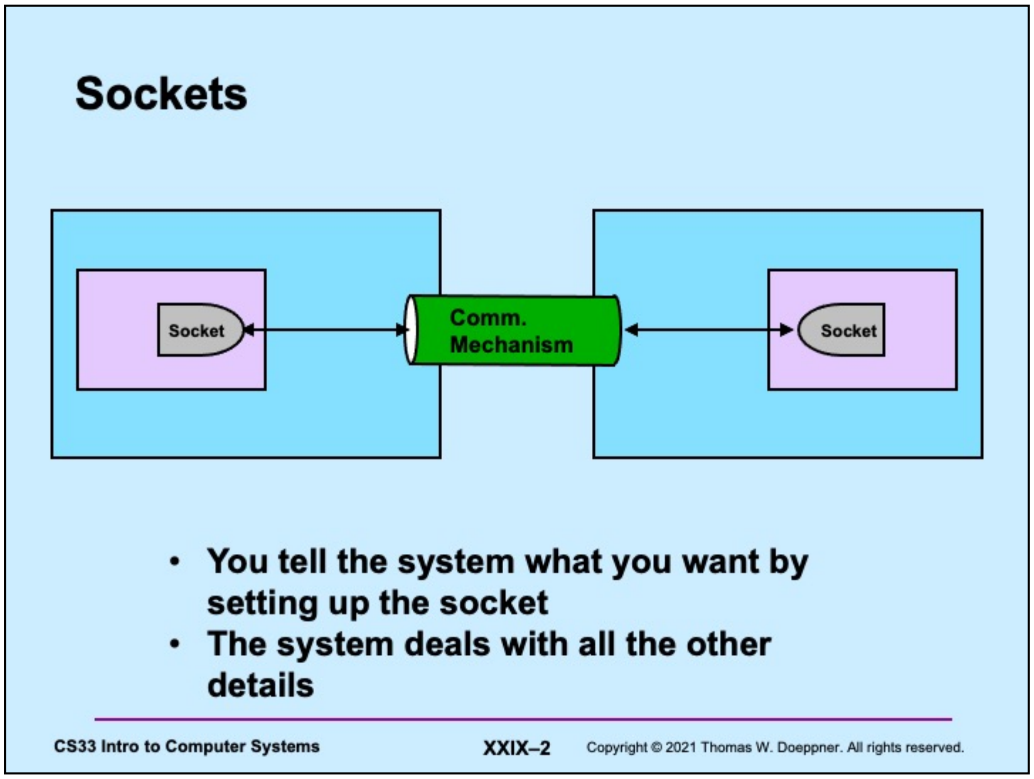


CS 33

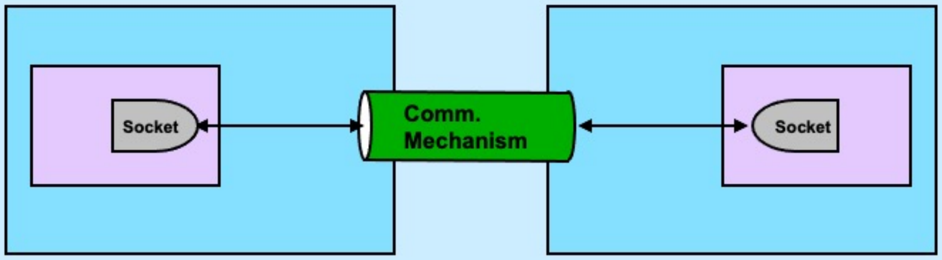
Network Programming (2)

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The source code used in this lecture, as well as some additional related source code, is on the course web page.



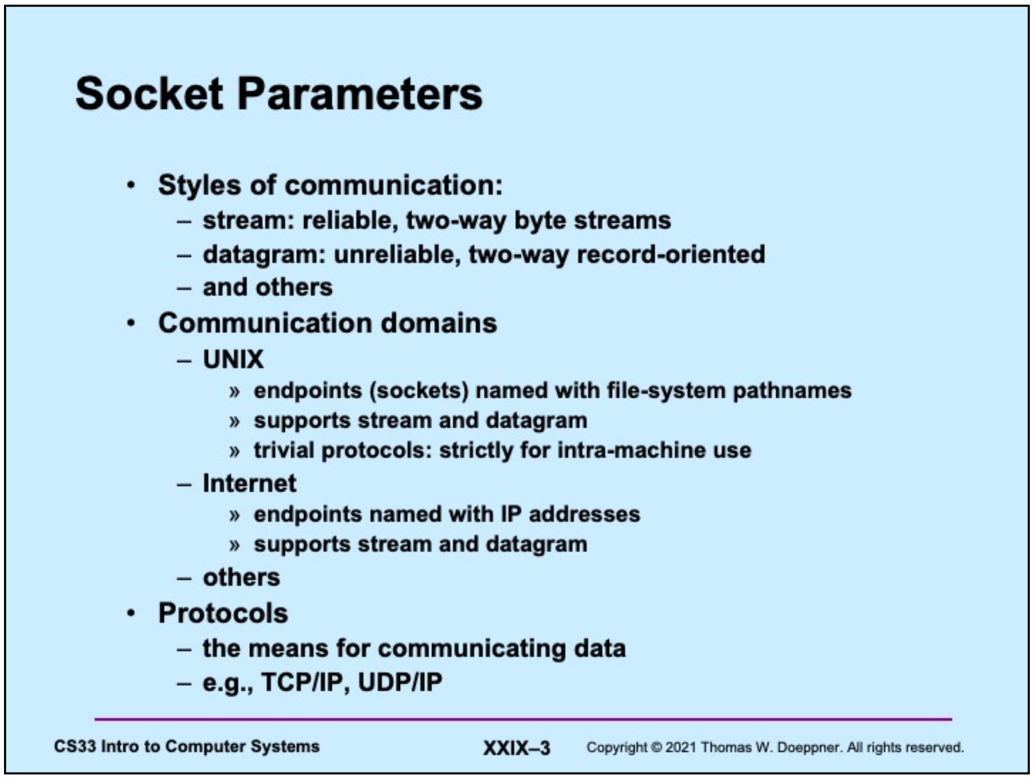
Sockets



- You tell the system what you want by setting up the socket
- The system deals with all the other details

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Sockets are the abstraction of the communication path. An application sets up a socket as the basis for communication. It refers to it via a file descriptor.

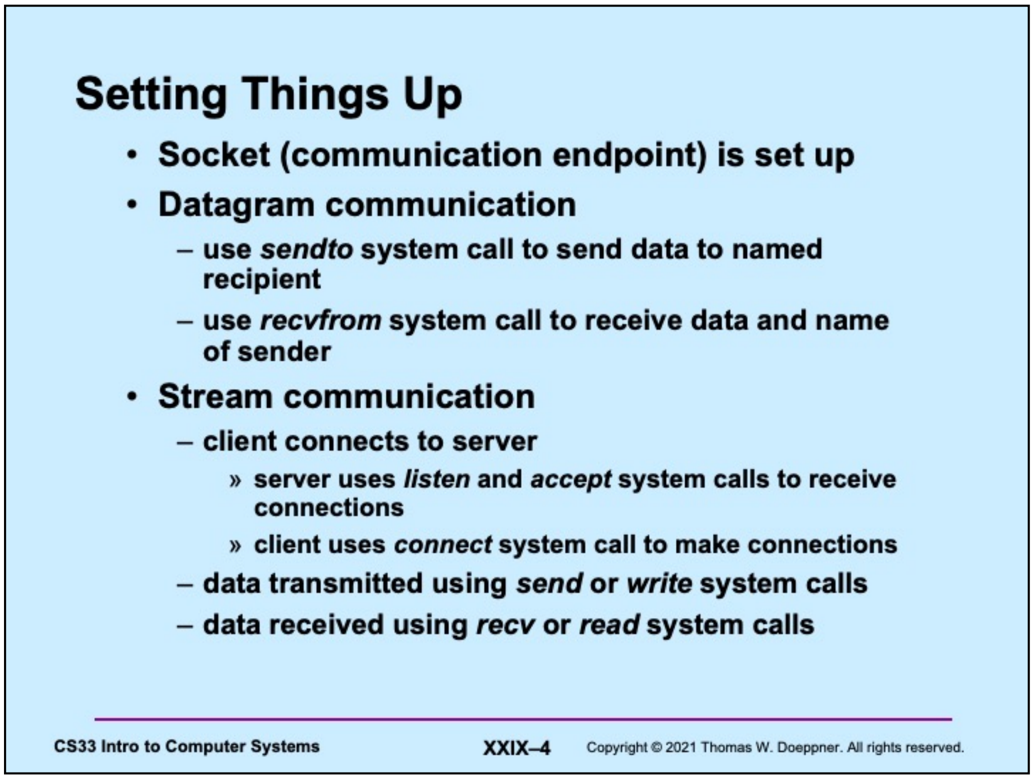


Socket Parameters

- Styles of communication:
  - stream: reliable, two-way byte streams
  - datagram: unreliable, two-way record-oriented
  - and others
- 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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We focus strictly on the internet domain.



Setting Things Up

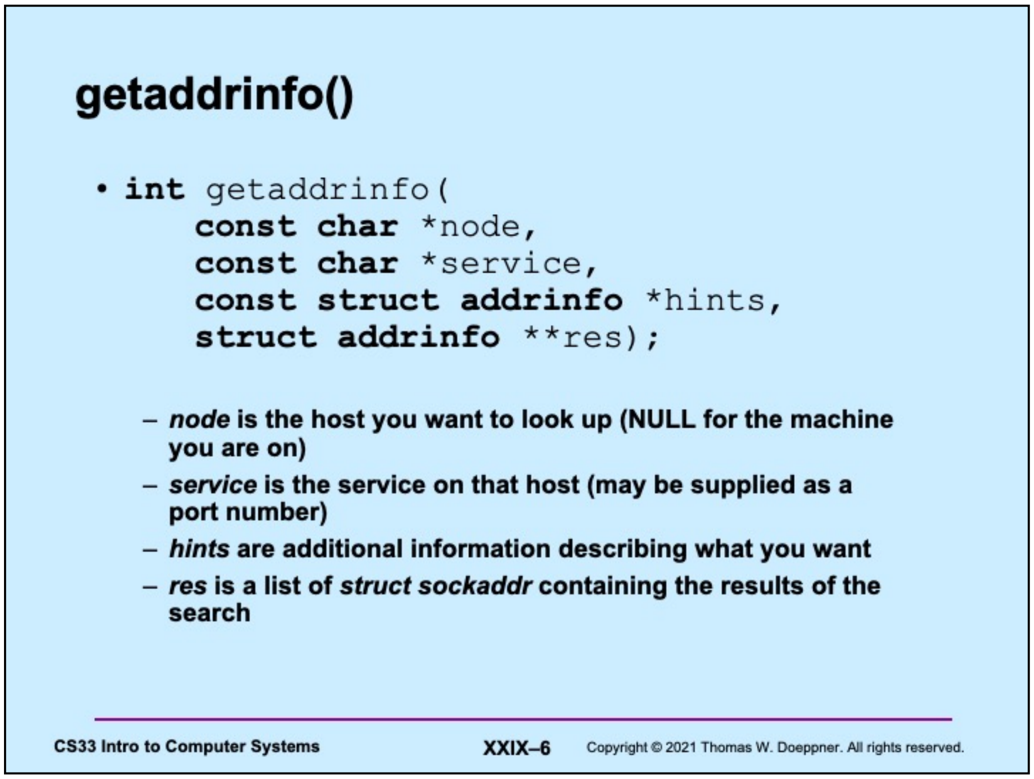
- Socket (communication endpoint) is set up
- Datagram communication
  - use *sendto* system call to send data to named recipient
  - use *recvfrom* system call to receive data and name of sender
- Stream communication
  - client connects to server
    - » server uses *listen* and *accept* system calls 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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Socket Addresses

- `struct sockaddr`
  - represents a network address
  - many sorts
    - » we use struct *sockaddr_in*
  - we can ignore the details
    - » embedded in layers of software
- getaddrinfo()
  - function used to obtain `struct sockaddr`'s

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

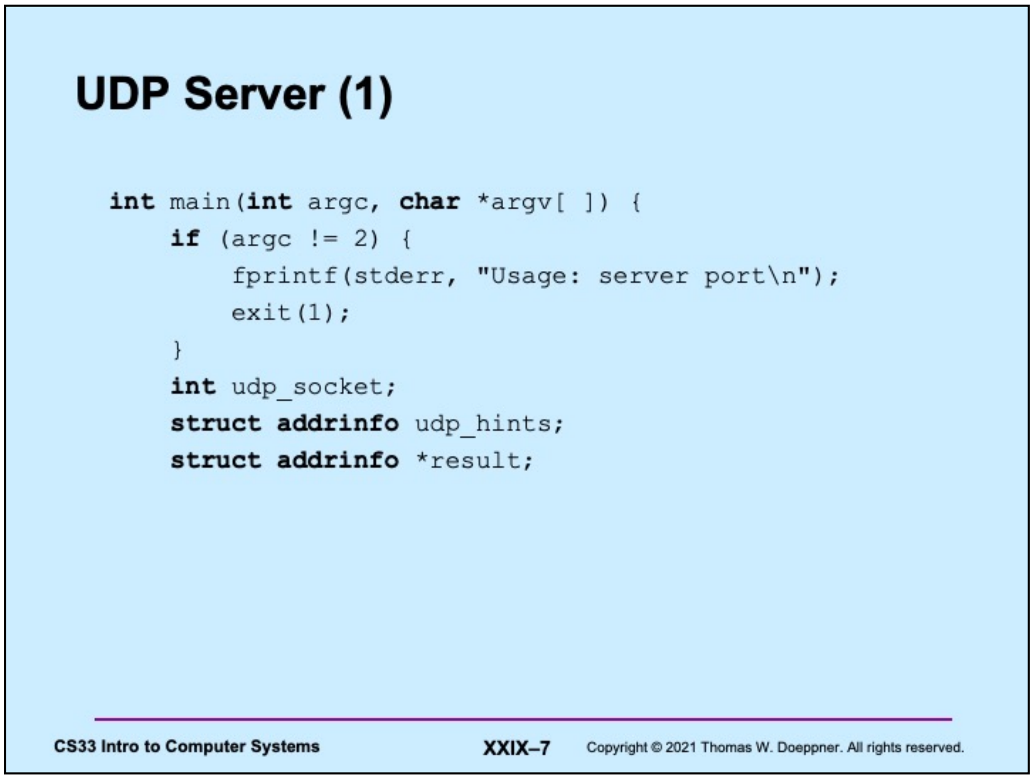


getaddrinfo()

-
  ```
  int getaddrinfo(
        const char *node,
        const char *service,
        const struct addrinfo *hints,
        struct addrinfo **res);
  ```
  - *node* is the host you want to look up (NULL for the machine you are on)
  - *service* is the service on that host (may be supplied as a port number)
  - *hints* are additional information describing what you want
  - *res* is a list of *struct sockaddr* containing the results of the search

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The general idea of using **getaddrinfo** is that you supply the name of the host you'd like to contact (*node*), which service on that host (*service*), and a description of how you'd like to communicate (**hints**). It returns a list of possible means for contacting the server in the form of a list of addrinfo structures (**res**). If the node argument is neither NULL nor the name of the local machine, getaddrinfo looks up what it needs in the domain name service (DNS) – the internet-wide distributed name service.



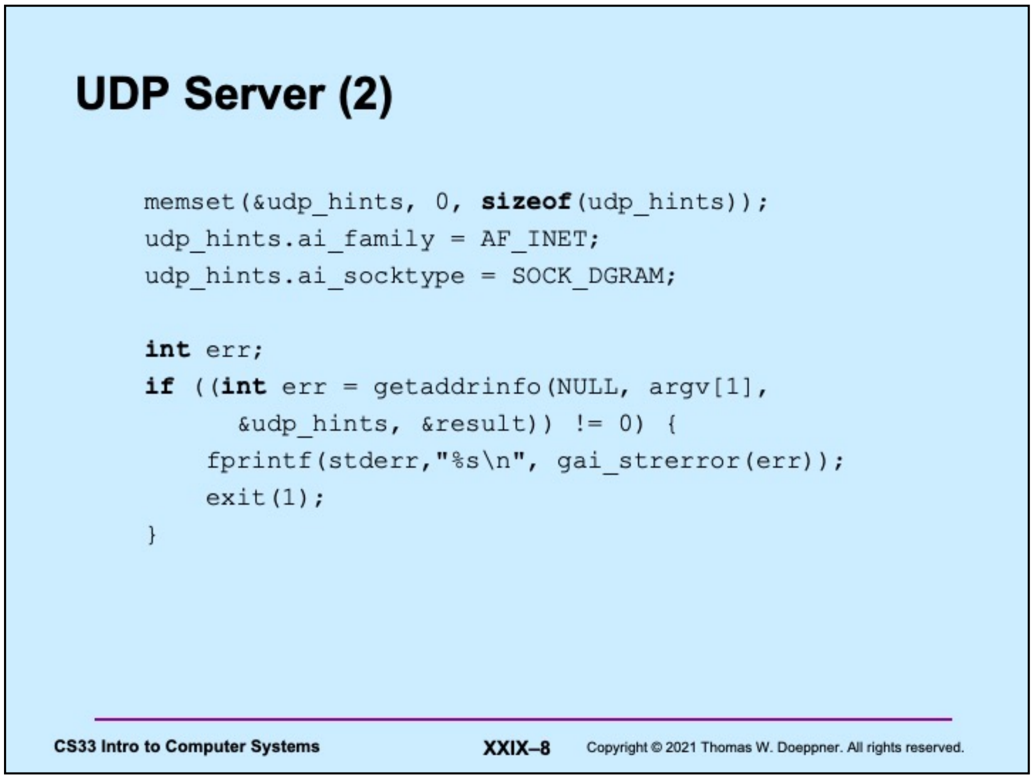
UDP Server (1)

```
int main(int argc, char *argv[ ]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: server port\n");
        exit(1);
    }
    int udp_socket;
    struct addrinfo udp_hints;
    struct addrinfo *result;
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Here we begin an example of a simple UDP server that receives messages from clients, prints them along with an indication of who sent the message, and politely responds.

In this first slide we check that we're invoked correctly (the command line should include the port number we're expecting to receive messages on) and have some initial declarations.



UDP Server (2)

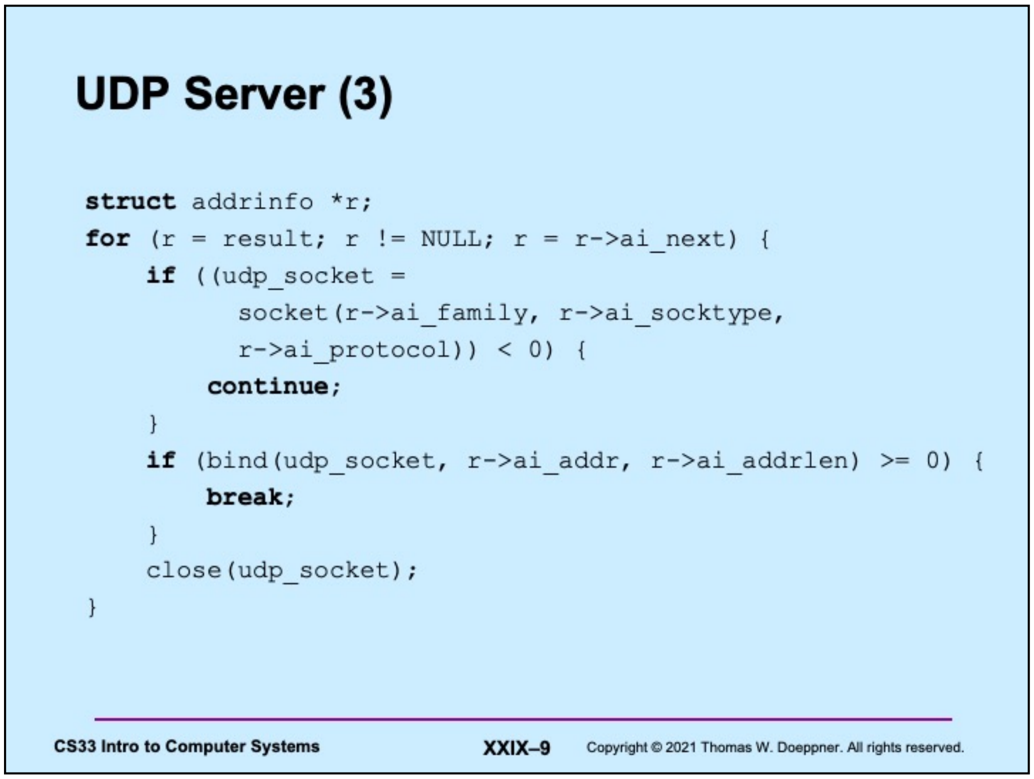
```
memset(&udp_hints, 0, sizeof(udp_hints));
udp_hints.ai_family = AF_INET;
udp_hints.ai_socktype = SOCK_DGRAM;

int err;
if ((int err = getaddrinfo(NULL, argv[1],
      &udp_hints, &result)) != 0) {
    fprintf(stderr,"%s\n", gai_strerror(err));
    exit(1);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The next step is to set up an address for our socket so that clients can contact us. In the *hints* structure, which we initialize to zeroes so that components we don't set are zero, we specify that we're using IPv4 (AF_INET), that we are using datagrams (which, over IPv4, means UDP).

We call **getaddrinfo** to get an appropriate address to bind to our socket (next slide). Note the use of **gai_strerror** to produce an error message given an error return from **getaddrinfo**. Note that its first (name) argument is NULL, which means that we want the address of the machine we're on.



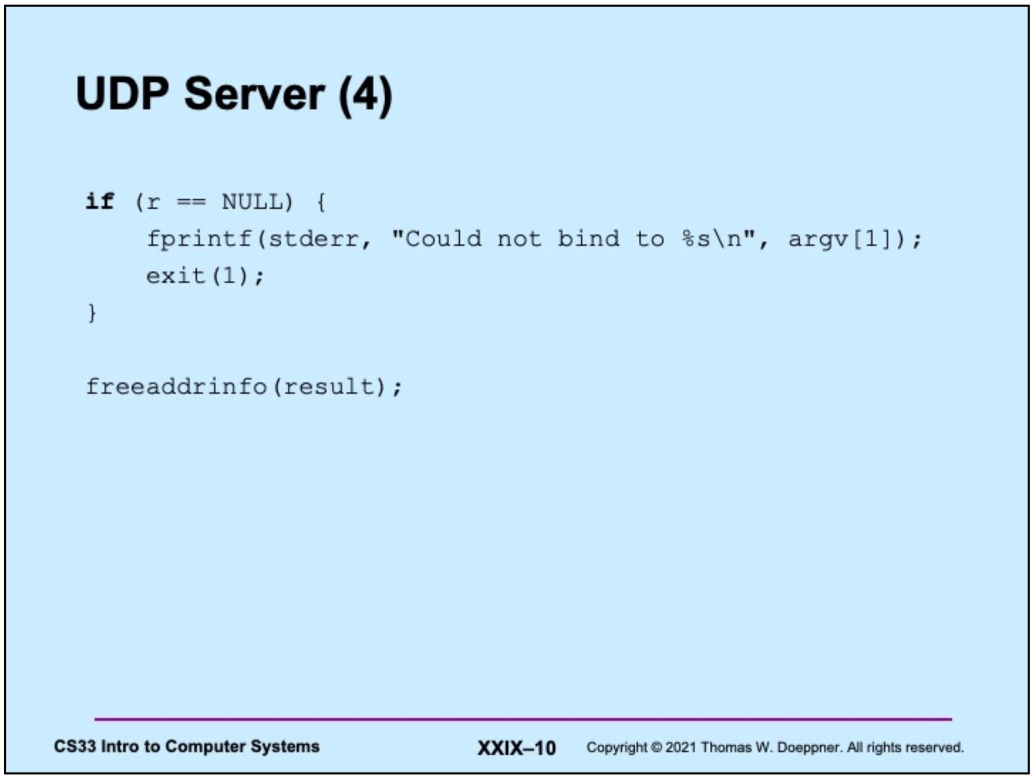
UDP Server (3)

```
struct addrinfo *r;
for (r = result; r != NULL; r = r->ai_next) {
    if ((udp_socket =
          socket(r->ai_family, r->ai_socktype,
          r->ai_protocol)) < 0) {
        continue;
    }
    if (bind(udp_socket, r->ai_addr, r->ai_addrlen) >= 0) {
        break;
    }
    close(udp_socket);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Next we iterate over the output of **getaddrinfo** (the list pointed to by its *result* argument). Though the length of this list is normally exactly one, it could be greater than one if our computer has multiple network interfaces. (The length could also be zero if it has no network interfaces, or none of the right sort.)

We try to create a socket that matches our desired socket type (by calling **socket**, giving it the **family**, **socket type** (e.g., stream or datagram), and **protocol** returned by **getaddrinfo**). Assuming we get the socket (which is referred to by the file descriptor **udp_socket**), we then try to bind it to the address returned by **getaddrinfo** (by calling **bind**). If all this works, we assume we're good to go. Otherwise, we try the next address in the list, if there are any more.

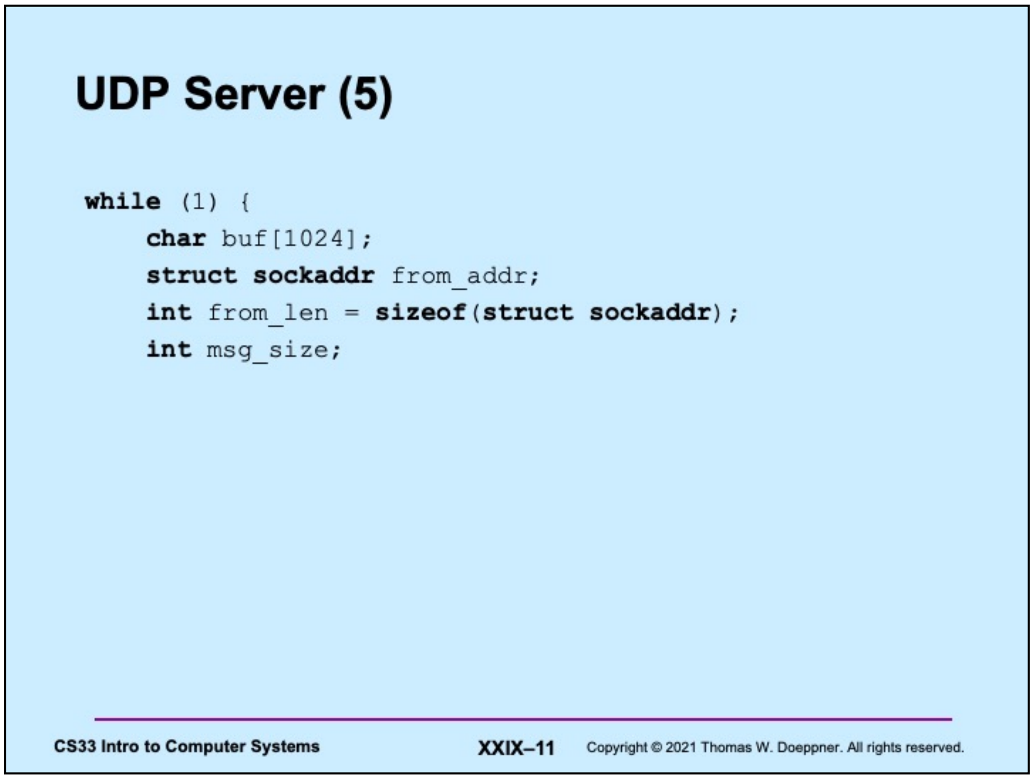

UDP Server (4)

```
if (r == NULL) {
    fprintf(stderr, "Could not bind to %s\n", argv[1]);
    exit(1);
}

freeaddrinfo(result);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

If we couldn't find anything that worked, we terminate the program. Otherwise we free up the list of addresses, since we don't need them anymore. Note the use of **freeaddrinfo** for this purpose.

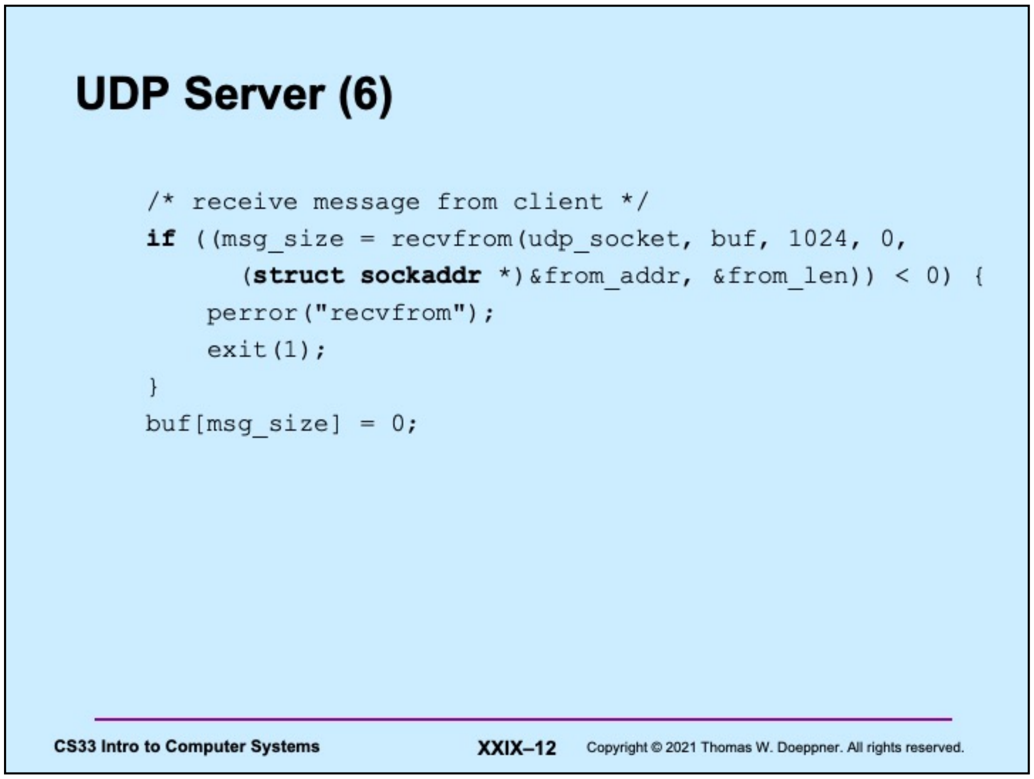


UDP Server (5)

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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Now that we've set up a socket and bound it to an address that clients can send messages to, we enter a loop to deal with all the incoming messages.



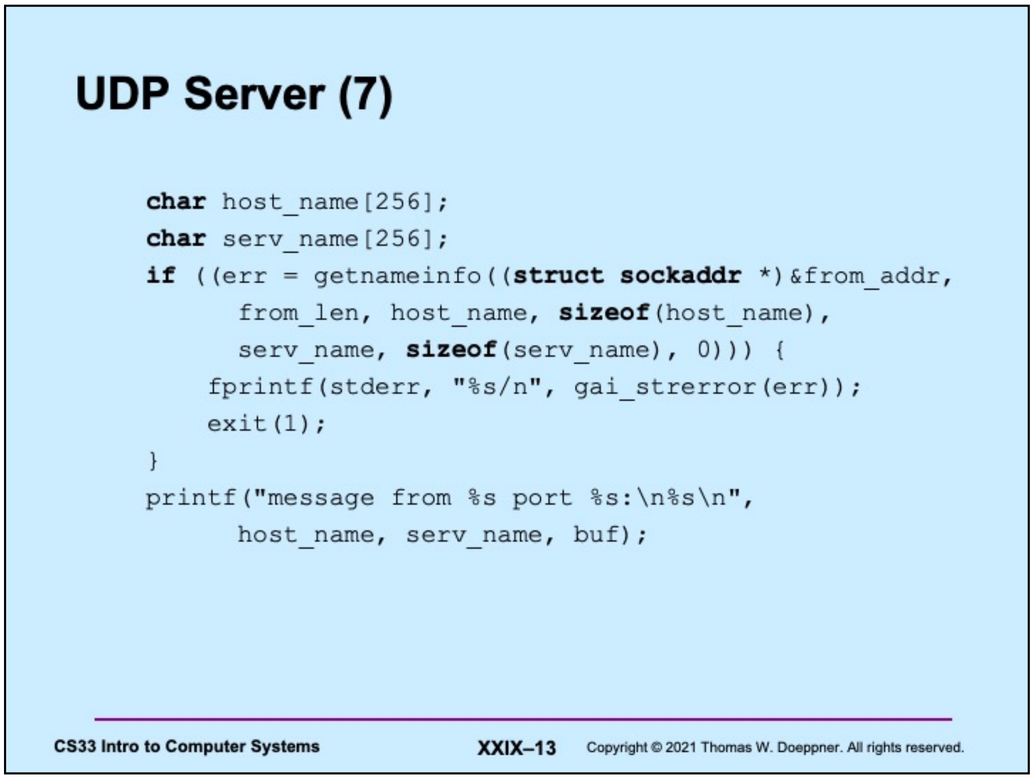
UDP Server (6)

```
/* receive message from client */
if ((msg_size = recvfrom(udp_socket, buf, 1024, 0,
      (struct sockaddr *)&from_addr, &from_len)) < 0) {
    perror("recvfrom");
    exit(1);
}
buf[msg_size] = 0;
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We call **recfrom** (which is just like read, but with extra arguments) to get the next message from a client. The fourth argument could specify some flags, but we don't need any here (or in the networking lab). The fifth and sixth arguments, if not zeroes, give an address of memory to receive the network name of the caller, as well as its length. The length argument serves two purposes: on entry to the function it indicates how much memory we have to receive the network address. On return from the function it tells us how many bytes were actually used.

Note that we put a zero at the end of buf, so we can safely print it (next slide).

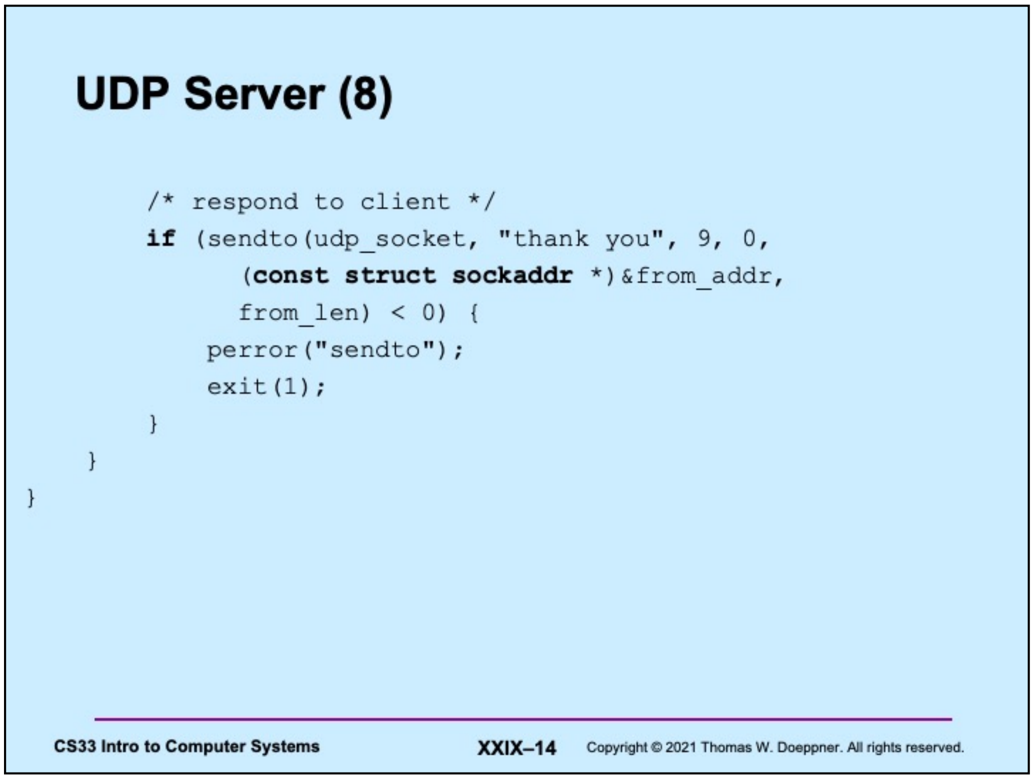


UDP Server (7)

```
char host_name[256];
char serv_name[256];
if ((err = getnameinfo((struct sockaddr *)&from_addr,
      from_len, host_name, sizeof(host_name),
      serv_name, sizeof(serv_name), 0))) {
    fprintf(stderr, "%s/n", gai_strerror(err));
    exit(1);
}
printf("message from %s port %s:\n%s\n",
      host_name, serv_name, buf);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

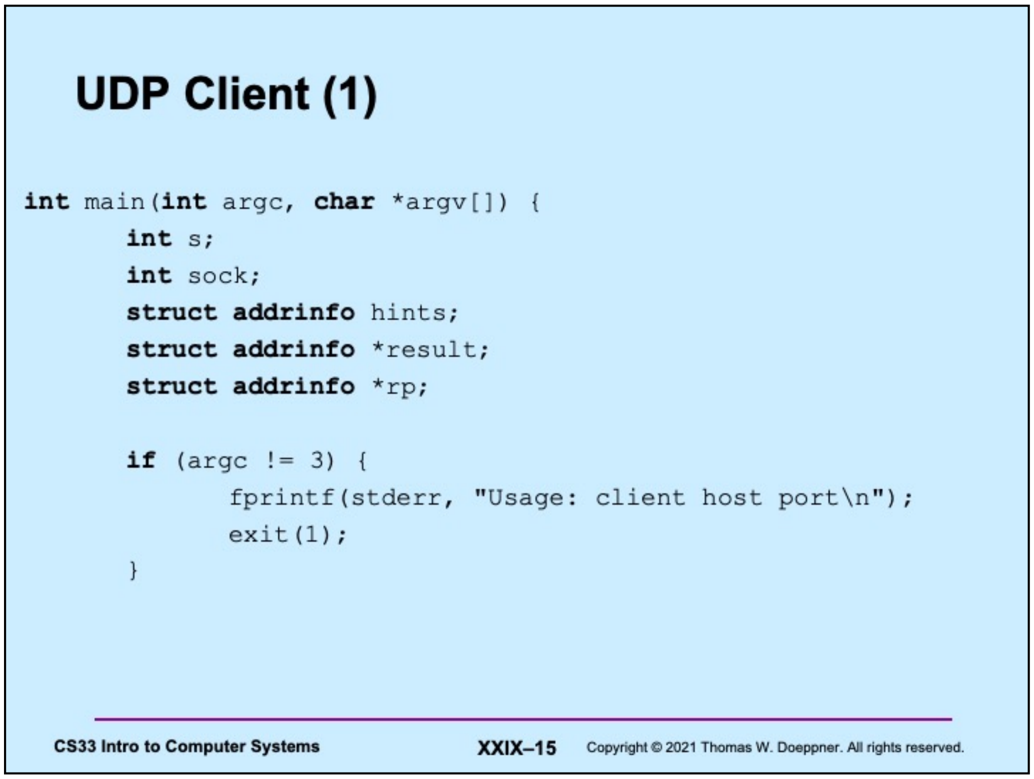
Next, we print out who the client was and what its message was. The function **getnameinfo** is sort of the inverse of **getaddrinfo**: given a struct sockaddr (as produced by **recvfrom**), it tells us the name of the machine and the service requested (or port number). We then print the name of the machine, the service name (or port number), and the message itself. Note the use of **gai_strerror** for interpreting an error return from **getnameinfo**.



UDP Server (8)

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

Finally, to be polite, we send a response to the client, thanking it for its message. The function **sendto** is like write, but with extra arguments. As with **recvfrom**, we set the flags argument (4th) to zero, but the next two arguments indicate whom we're sending the message to (the client, in this case).



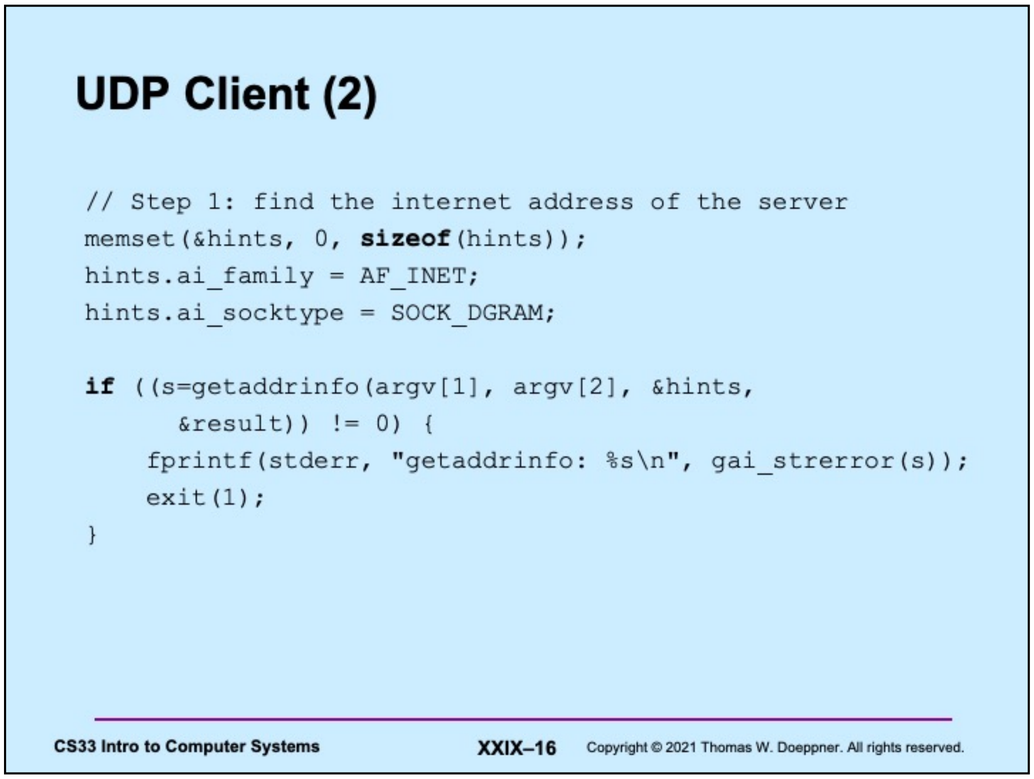
UDP Client (1)

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

	if (argc != 3) {
		fprintf(stderr, "Usage: client host port\n");
		exit(1);
	}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Now we look at the code for a client that communicates with our UDP server. Note that the command line of the client specifies both the host the server is on, as well as the port number. If the server is on the same host as the client, host may be specified as "localhost".



UDP Client (2)

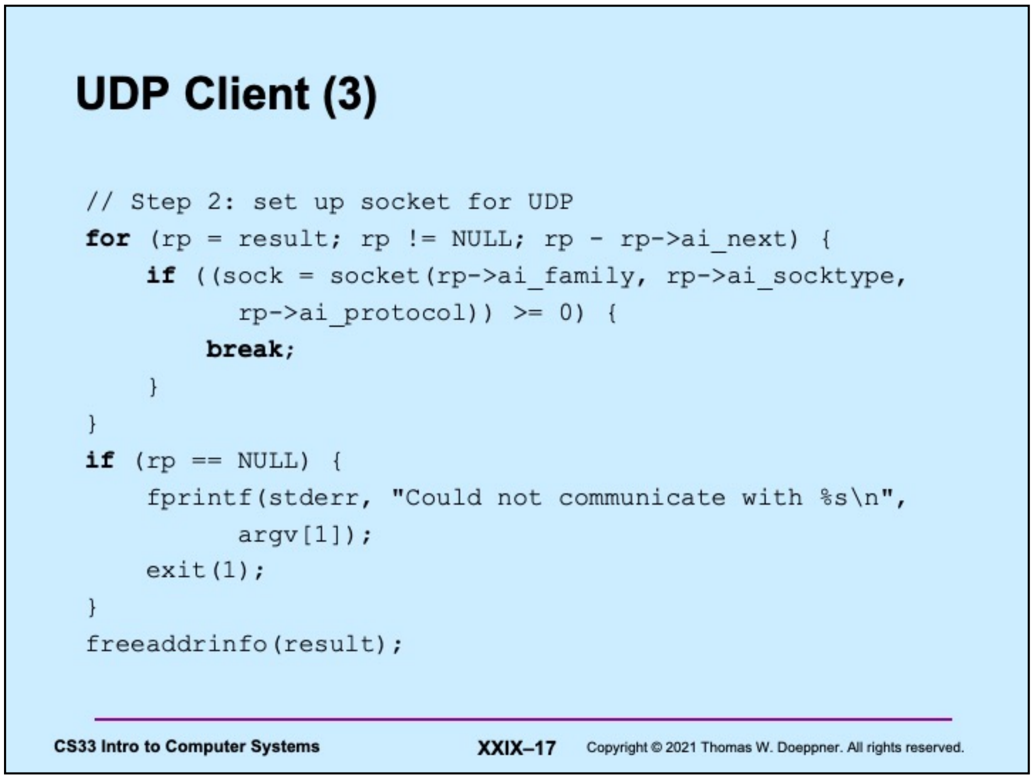
```
// Step 1: find the internet address of the server
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;

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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We start by looking up the internet address of the server. To do this, we first fill in the hints structure to make it clear that we want a server with an internet (IPv4) interface and that we want UDP (datagrams). We call **getaddrinfo** to get a list of addresses. Again, note the use of **gai_strerror** to give us an error message.

Unlike what we did for the server code, we supply a non-null first argument to **getaddrinfo**, indicating which server we want to communicate with.



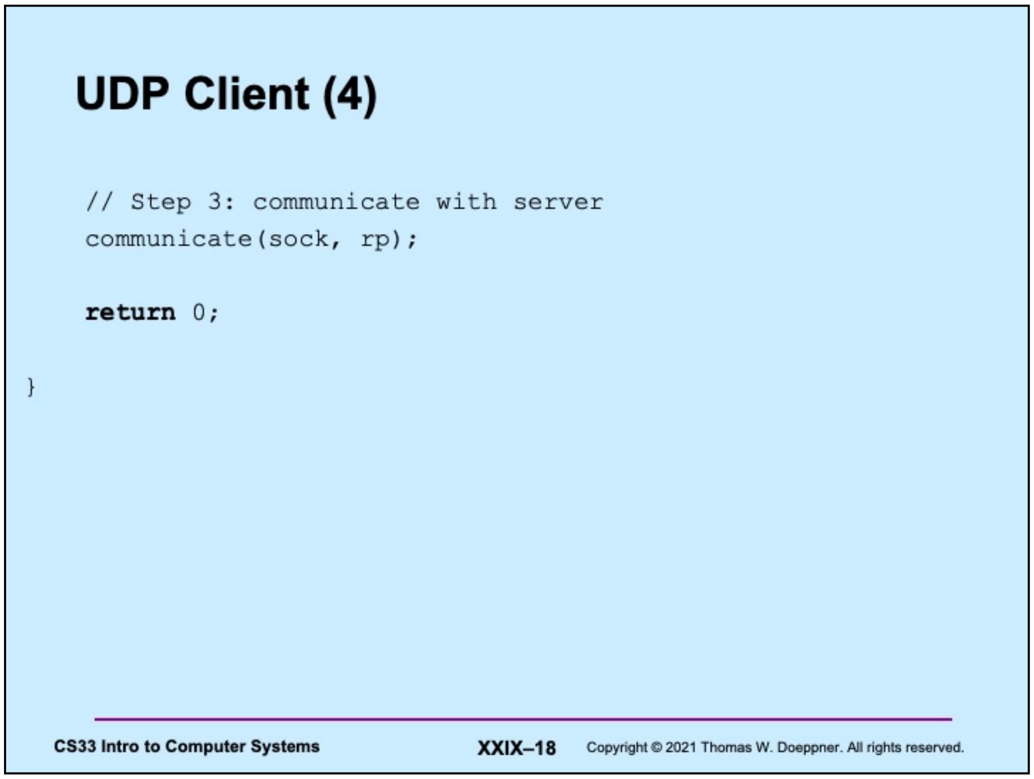
UDP Client (3)

```
// Step 2: set up socket for UDP
for (rp = result; rp != NULL; rp - rp->ai_next) {
    if ((sock = socket(rp->ai_family, rp->ai_socktype,
          rp->ai_protocol)) >= 0) {
        break;
    }
}
if (rp == NULL) {
    fprintf(stderr, "Could not communicate with %s\n",
          argv[1]);
    exit(1);
}
freeaddrinfo(result);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Next, we go through the addresses returned by **getaddrinfo** and use the first one for which we can successfully set up a socket. The list's length is usually one, and that one usually works. Note that we aren't using **bind** here to assign a port number to the socket we've set up (as we did for the server). If we wanted our client to use a particular port, we could call **bind** here, but in most situations we don't care what the client's port is, and thus the operating systems chooses an unused port number for us.

We free up list (by calling **freeaddrinfo**) since we no longer need it.



UDP Client (4)

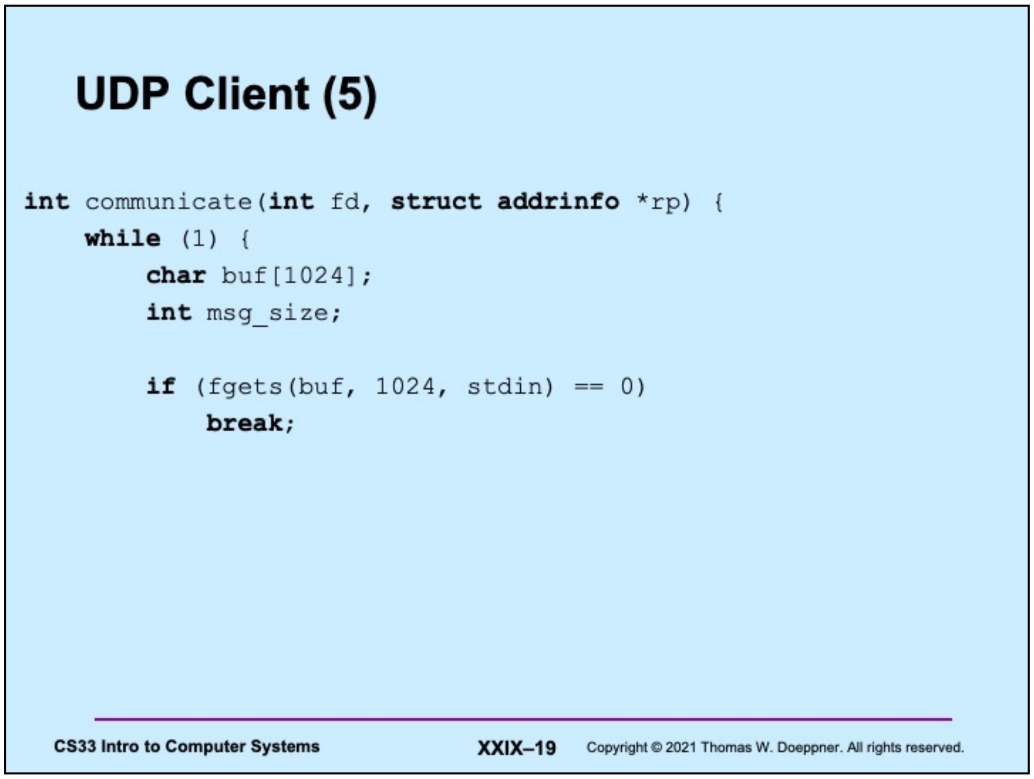
```
    // Step 3: communicate with server
    communicate(sock, rp);

    return 0;

}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Next we call our communicate function that will exchange messages with the server (although we don't know yet whether the server is up and running).



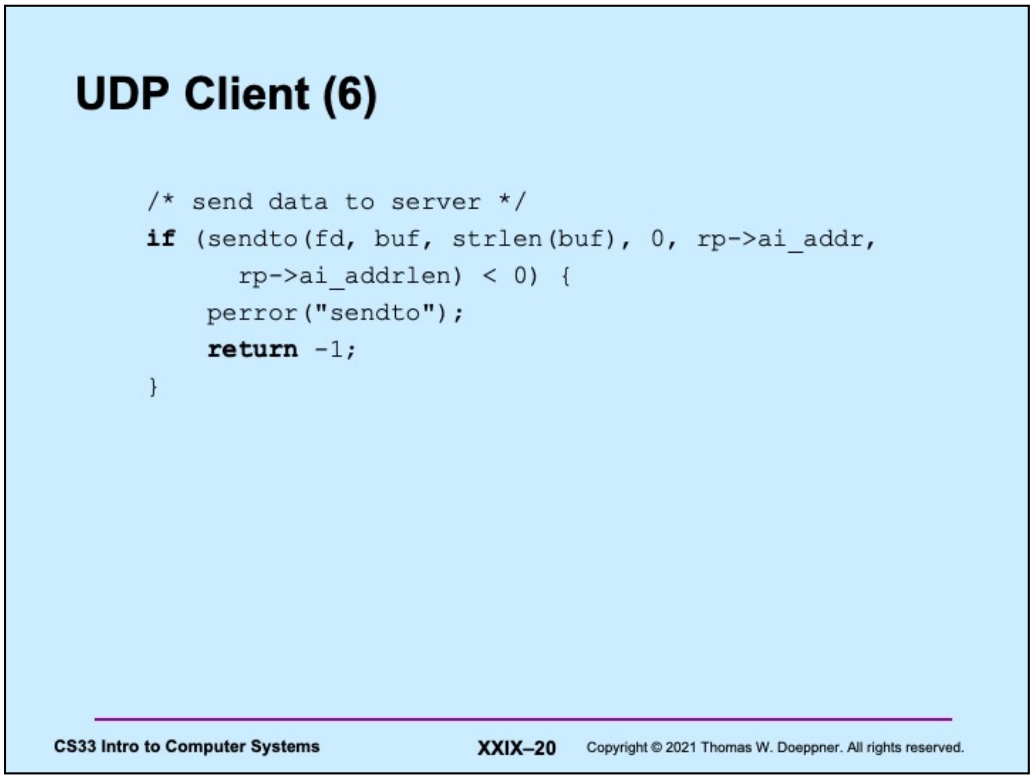
UDP Client (5)

```
int communicate(int fd, struct addrinfo *rp) {
    while (1) {
        char buf[1024];
        int msg_size;

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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

In our *communicate* function, we first read a line from stdin (which will be sent to the server).

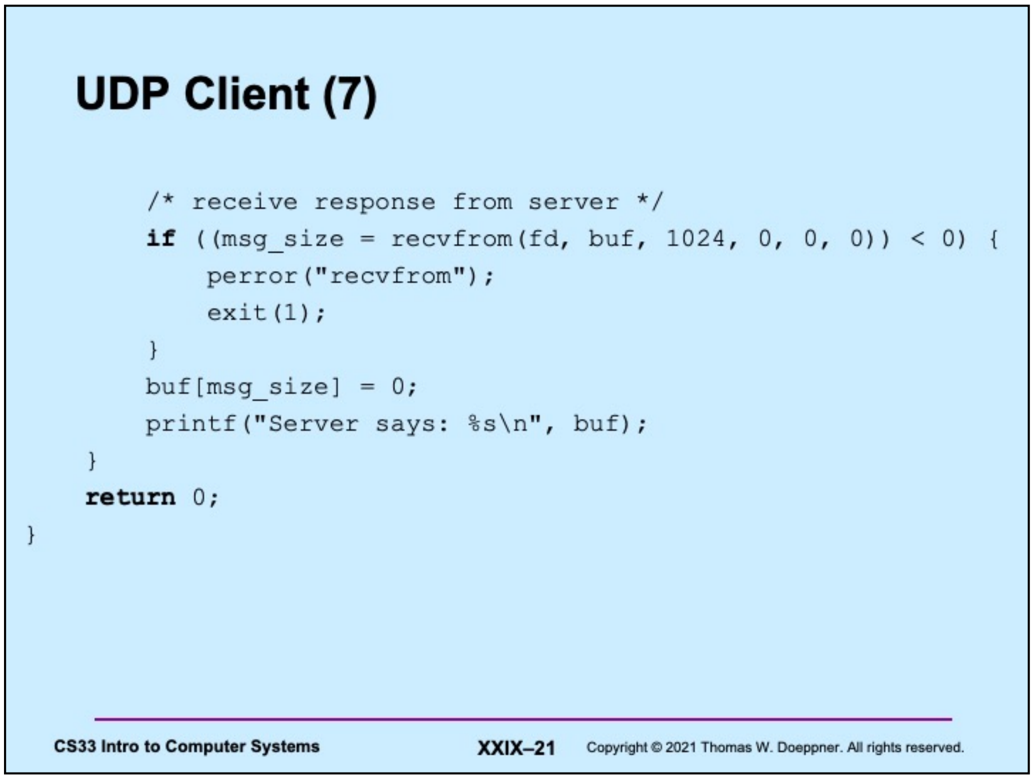


UDP Client (6)

```
/* send data to server */
if (sendto(fd, buf, strlen(buf), 0, rp->ai_addr,
      rp->ai_addrlen) < 0) {
    perror("sendto");
    return -1;
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The client sends to the server what was just read from stdin.



UDP Client (7)

```
        /* 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;
}
```

The client receives the server's response, makes sure it's null-terminated, and prints it out.

Quiz 1

Suppose a process on one machine sends a datagram to a process on another machine. The sender uses *sendto* and the receiver uses *recvfrom*. There's a momentary problem with the network and the datagram doesn't make it to the receiving process. Its call to *recvfrom*

a) returns –1 (indicating an error)

b) returns 0

c) returns some other value

d) doesn't return

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Reliable Communication

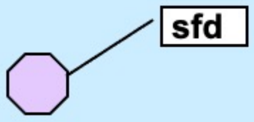
- The promise …
  - what is sent is received
  - order is preserved
- Set-up is required
  - two parties agree to communicate
  - within the implementation of the protocol:
    - » each side keeps track of what is sent, what is received
    - » received data is acknowledged
    - » unack'd data is re-sent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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Streams in the Inet Domain (1)

- **Server steps**
  1) create socket

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

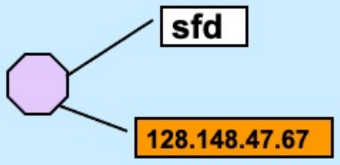


Copyright © 2021 Thomas W. Doeppner. All rights reserved.

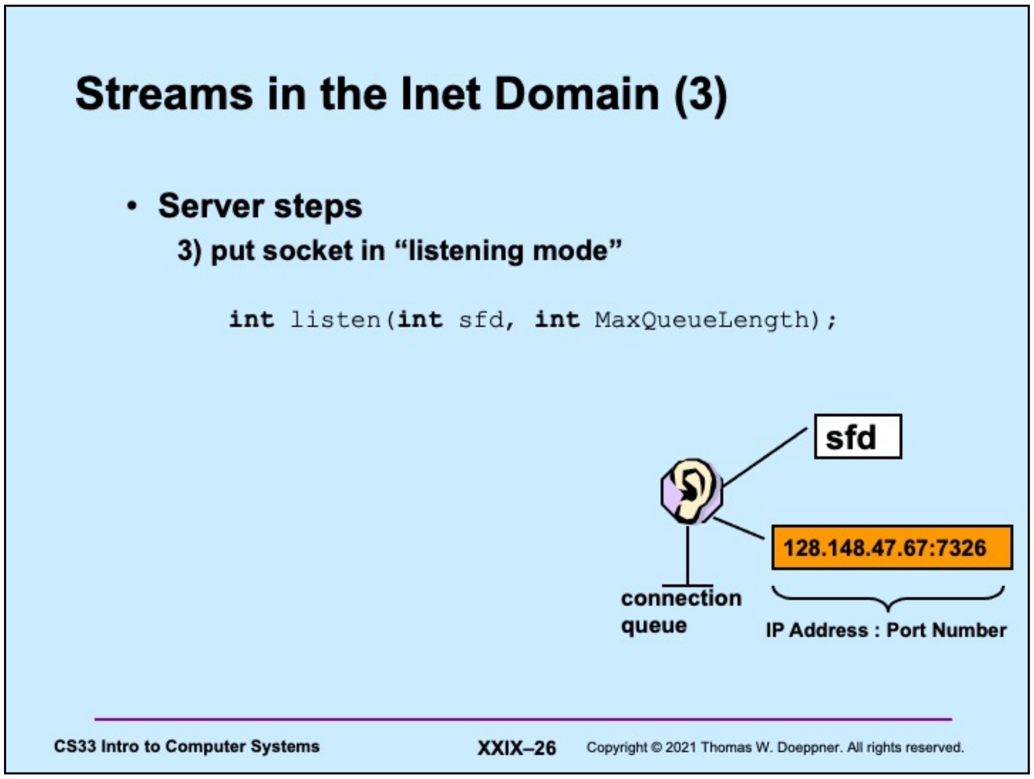
Streams in the Inet Domain (2)

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

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



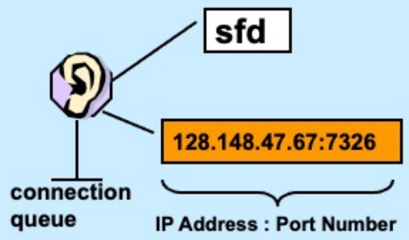
Copyright © 2021 Thomas W. Doeppner. All rights reserved.



Streams in the Inet Domain (3)

- **Server steps**
  - 3) put socket in "listening mode"

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



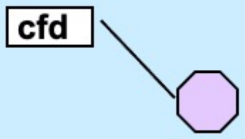
Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The **listen** system call tells the OS that the process would like to receive connections from clients via the indicated socket. The **MaxQueueLength** argument is the maximum number of connections that may be queued up, waiting to be accepted. Its maximum value is in /proc/sys/net/core/somaxconn (and is currently 128).

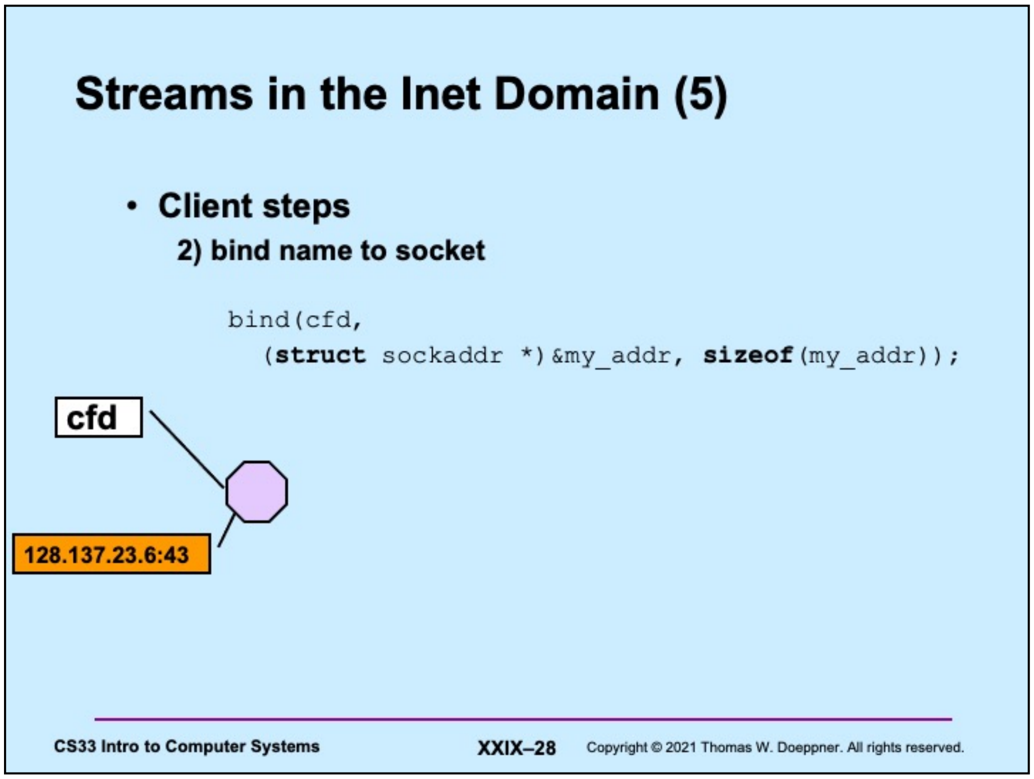
Streams in the Inet Domain (4)

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

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



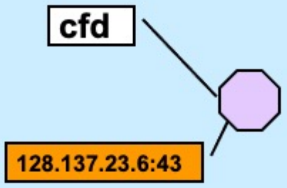
Copyright © 2021 Thomas W. Doeppner. All rights reserved.



Streams in the Inet Domain (5)

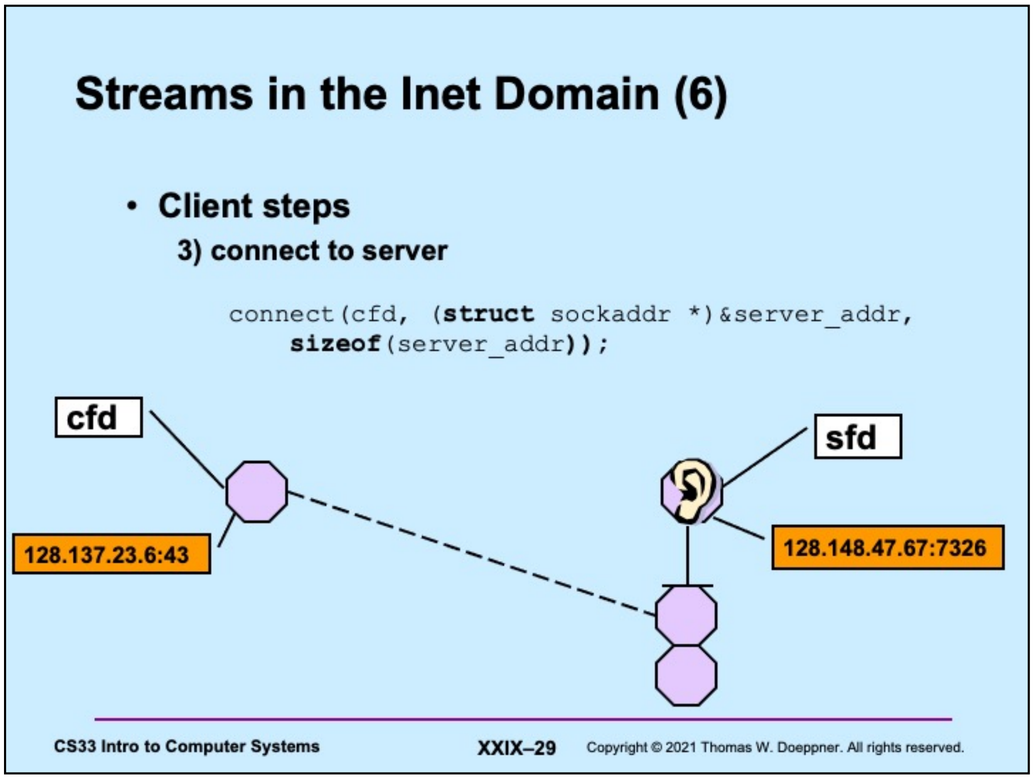
- **Client steps**
  - **2) bind name to socket**

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



Copyright © 2021 Thomas W. Doeppner. All rights reserved.

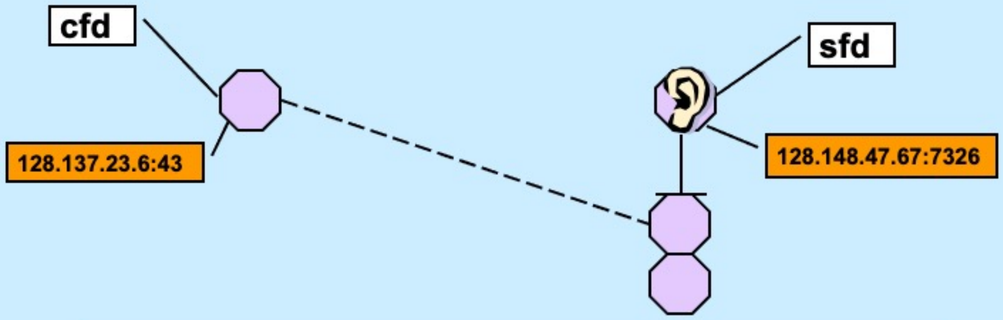
This step is optional – if not done, the OS does it automatically, supplying some available port number.



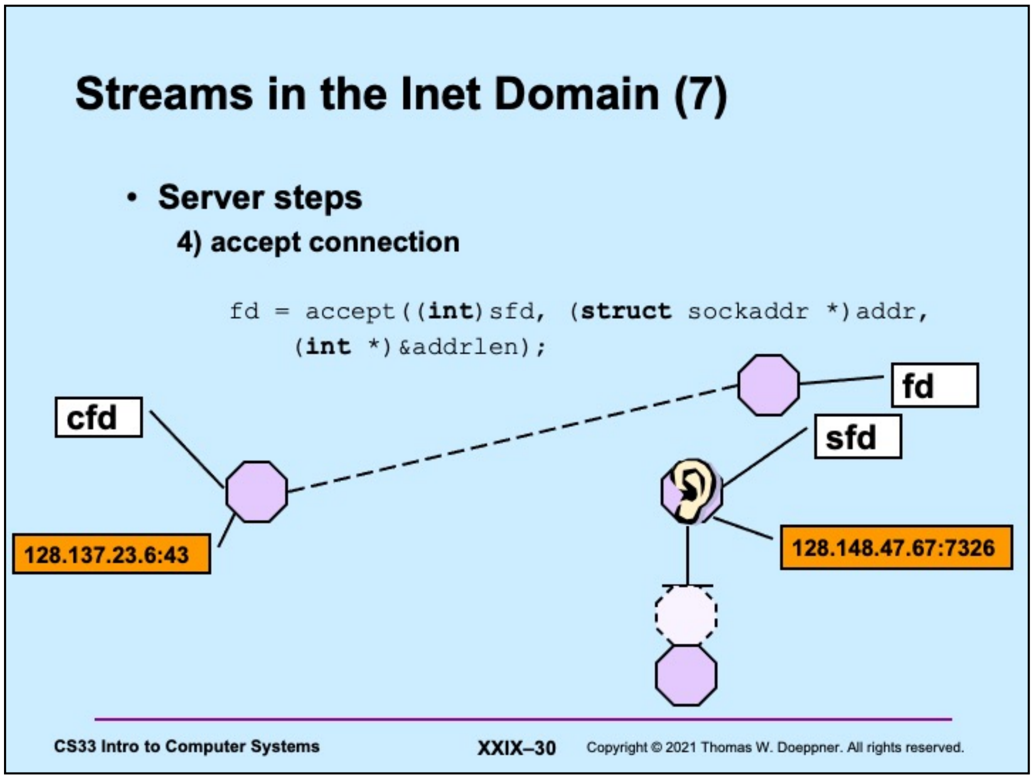
Streams in the Inet Domain (6)

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

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



The client issues the **connect** system call to initiate a connection with the server. The first argument is a file descriptor referring to the client's socket. Ultimately this socket will be connected to a socket on the server. Behind the scenes the client OS communicates with the server OS via a protocol-specific exchange of messages. Eventually a connection is established and a new socket is created on the server to represent its end of the connection. This socket is queued on the server's listening socket, where it stays until the server process accepts the connection (as shown in the next slide).



Streams in the Inet Domain (7)

- **Server steps**

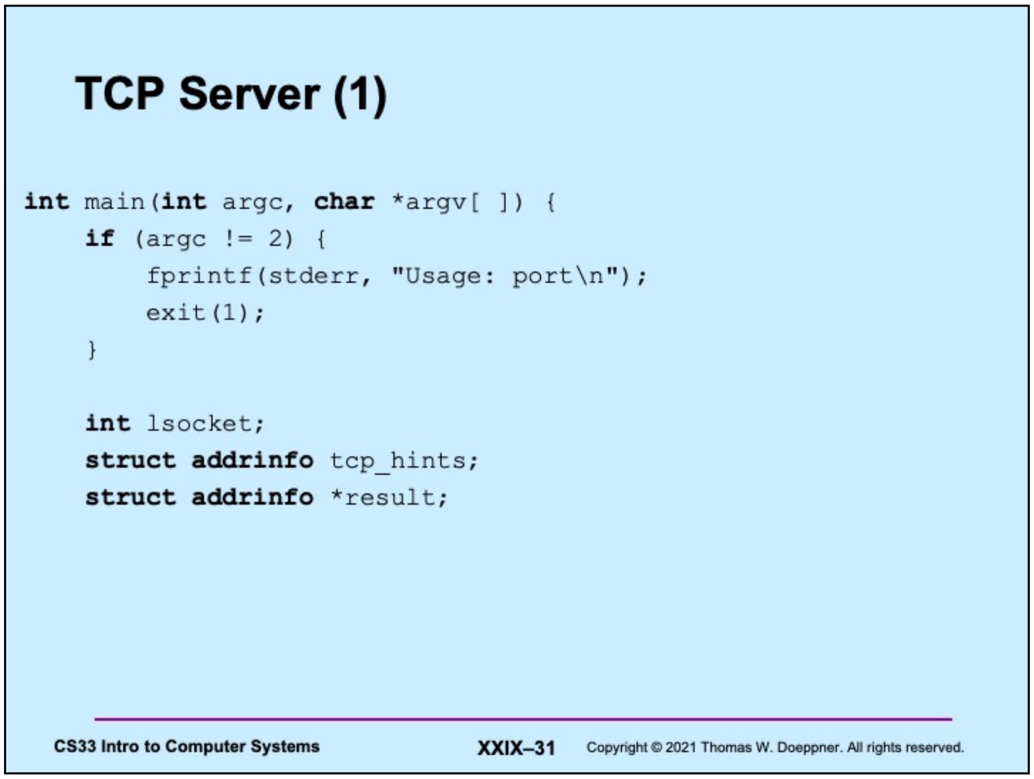
  4) accept connection

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

cfd
fd
sfd
128.137.23.6:43
128.148.47.67:7326

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

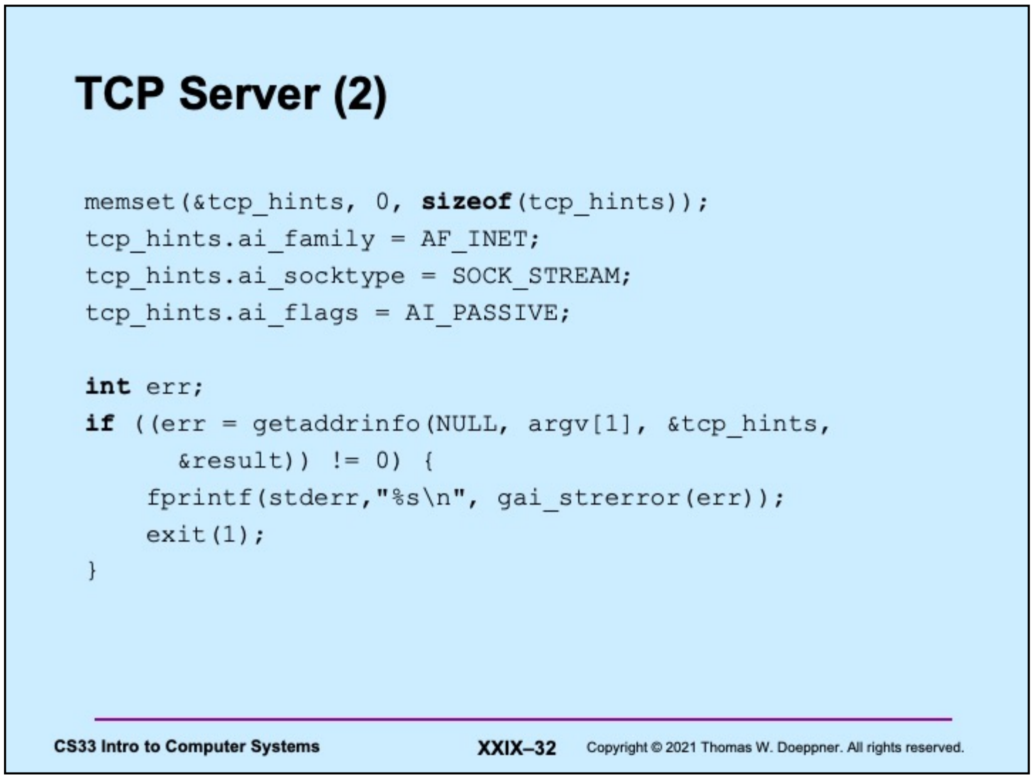
The server process issues an **accept** system which waits if necessary for a connected socked to appear on the listening socket's queue, then pulls the first such socket from the queue. This socket is the server end of a connection from a client. A file descriptor is returned that refers to that socket, allowing the process to now communicate with the client.


TCP Server (1)

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

    int lsocket;
    struct addrinfo tcp_hints;
    struct addrinfo *result;
```

We begin looking at a TCP example similar to our UDP example. Clients will contact our server, which prints everything its clients send to it.



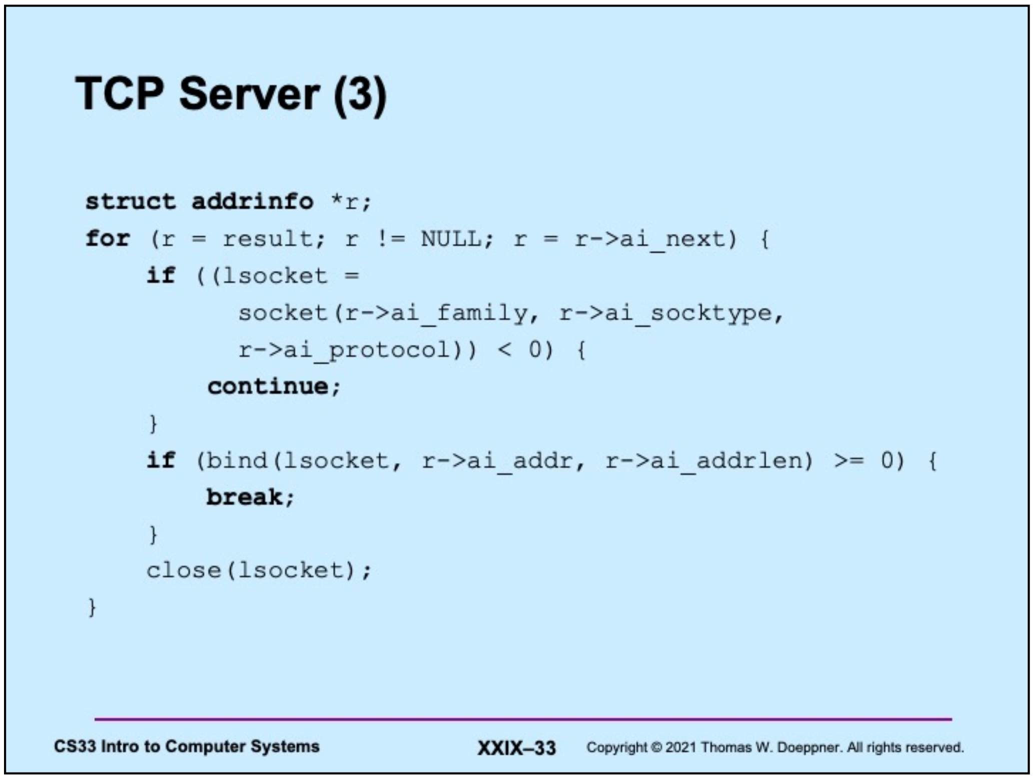
TCP Server (2)

```
memset(&tcp_hints, 0, sizeof(tcp_hints));
tcp_hints.ai_family = AF_INET;
tcp_hints.ai_socktype = SOCK_STREAM;
tcp_hints.ai_flags = AI_PASSIVE;

int err;
if ((err = getaddrinfo(NULL, argv[1], &tcp_hints,
      &result)) != 0) {
    fprintf(stderr,"%s\n", gai_strerror(err));
    exit(1);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The server starts by using **getaddrinfo** to obtain information about its interfaces. Via **tcp_hints**, we request information about IPv4 (AF_INET) interfaces supporting TCP (SOCK_STREAM). The value to which **ai_flags** is set (AI_PASSIVE) indicates that our socket will be put into listening mode and its address will be set to allow it to receive connections on any network it's attached to.

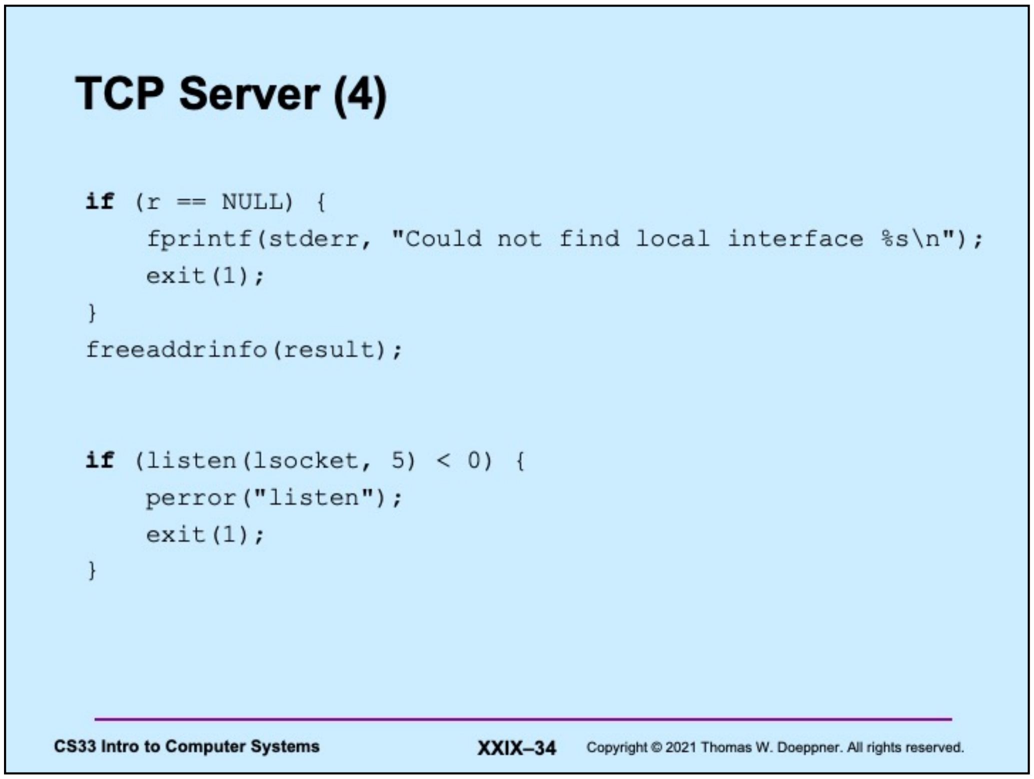


TCP Server (3)

```
struct addrinfo *r;
for (r = result; r != NULL; r = r->ai_next) {
    if ((lsocket =
          socket(r->ai_family, r->ai_socktype,
          r->ai_protocol)) < 0) {
        continue;
    }
    if (bind(lsocket, r->ai_addr, r->ai_addrlen) >= 0) {
        break;
    }
    close(lsocket);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Here we look at the list of **addrinfo** structures returned by **getaddrinfo** and use the first for which we can create a socket and bind to (as usual with this, it will probably be the first and only item in the list).



TCP Server (4)

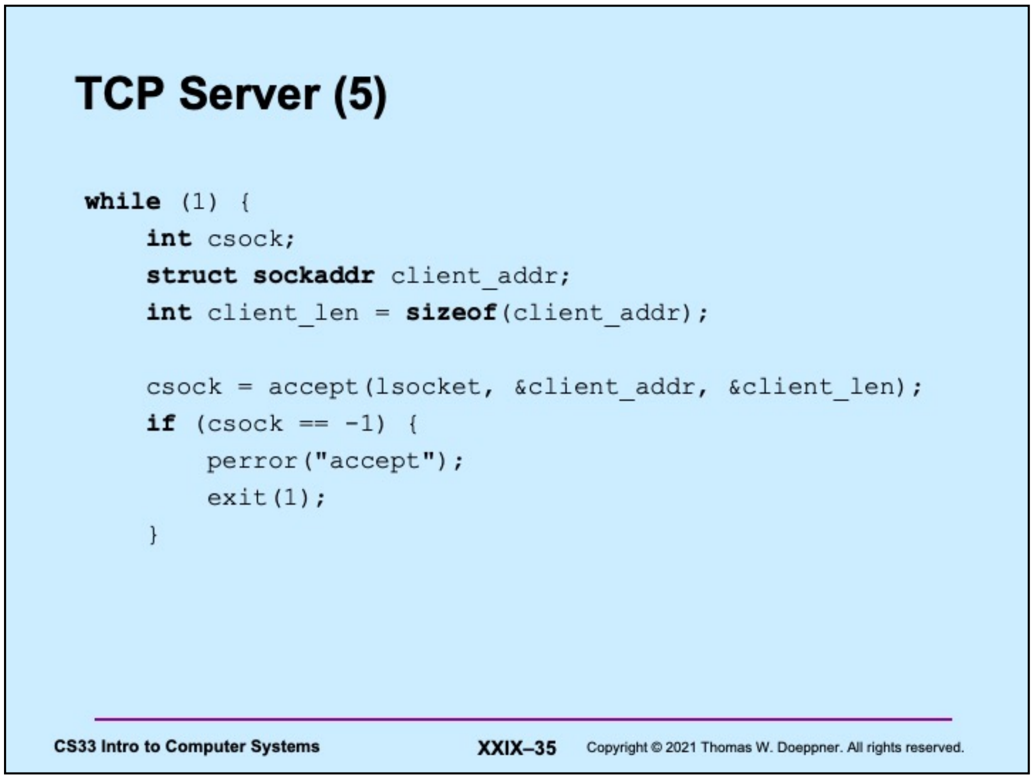
```
if (r == NULL) {
    fprintf(stderr, "Could not find local interface %s\n");
    exit(1);
}
freeaddrinfo(result);


if (listen(lsocket, 5) < 0) {
    perror("listen");
    exit(1);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We check to make sure we found a suitable local address. Assuming we did, we free list of addresses, since we don't need them anymore.

Now that we have a socket, we put it in listening mode, indicating a maximum queue length of 5 (an arbitrarily chosen value).



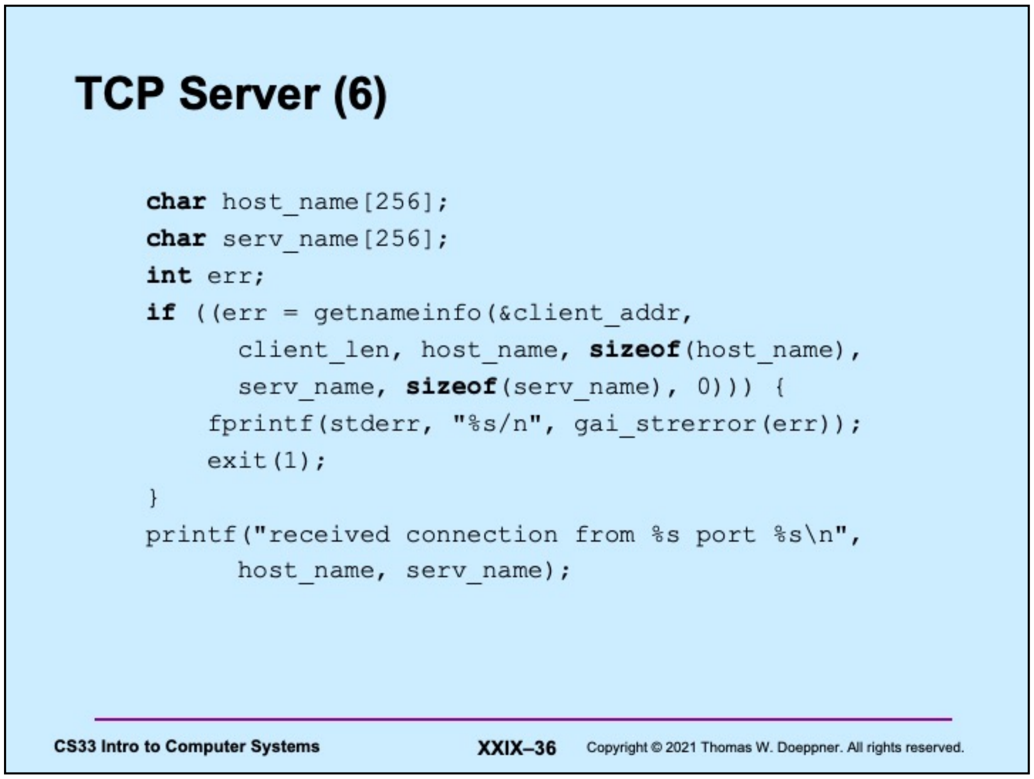
TCP Server (5)

```
while (1) {
    int csock;
    struct sockaddr client_addr;
    int client_len = sizeof(client_addr);

    csock = accept(lsocket, &client_addr, &client_len);
    if (csock == -1) {
        perror("accept");
        exit(1);
    }
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The server now begins a loop, accepting incoming connection requests from clients. Each time *accept* returns (assuming no errors), we have a file descriptor (**csock**) for the new client connection.

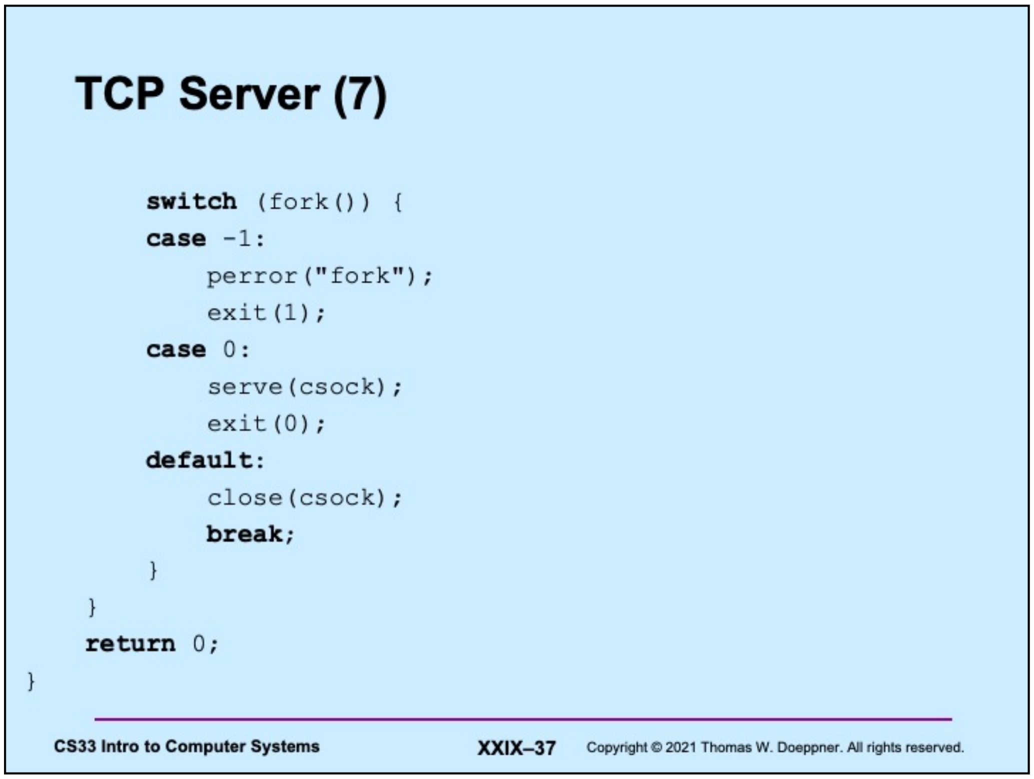


TCP Server (6)

```
char host_name[256];
char serv_name[256];
int err;
if ((err = getnameinfo(&client_addr,
      client_len, host_name, sizeof(host_name),
      serv_name, sizeof(serv_name), 0))) {
    fprintf(stderr, "%s/n", gai_strerror(err));
    exit(1);
}
printf("received connection from %s port %s\n",
      host_name, serv_name);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We figure how who the client is, based on the information returned by accept. We use **getnameinfo** to decode the host name and the service name (port number). Note the use of **gai_strerror** to deal with errors.

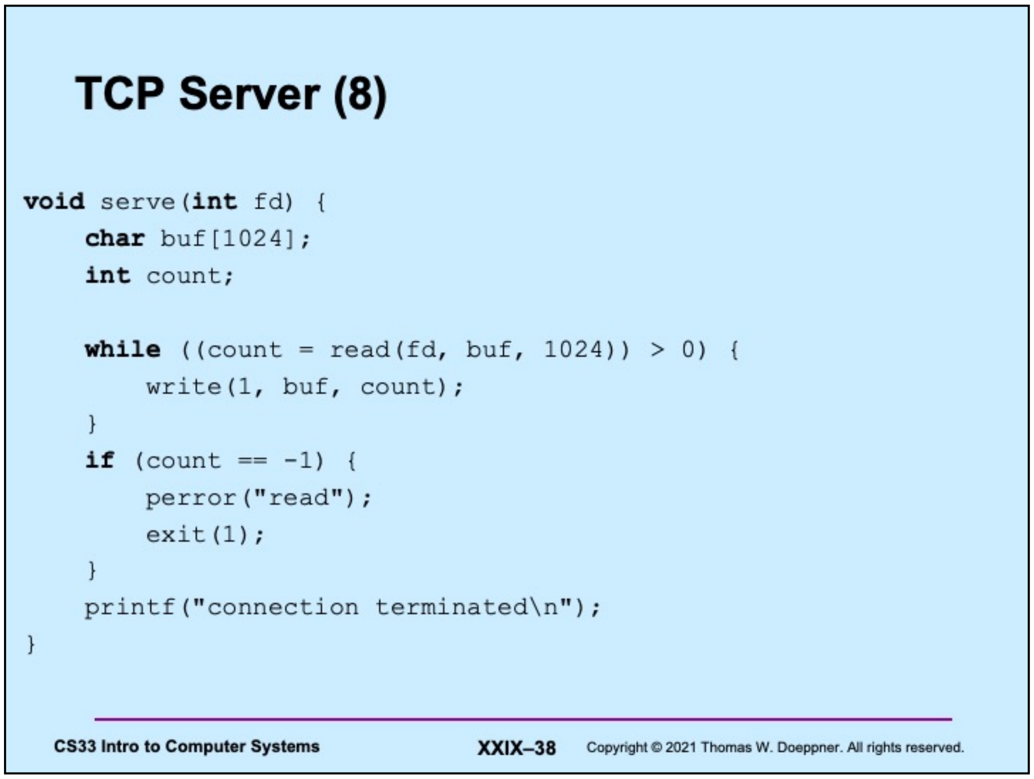


TCP Server (7)

```
        switch (fork()) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            serve(csock);
            exit(0);
        default:
            close(csock);
            break;
        }
    }
    return 0;
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The server, having just received a connection from the client, creates a new process to handle that client's connection. The new (child) process calls serve, passing it the file descriptor for the connected socket. The parent has no further use for that file descriptor, so it closes it.



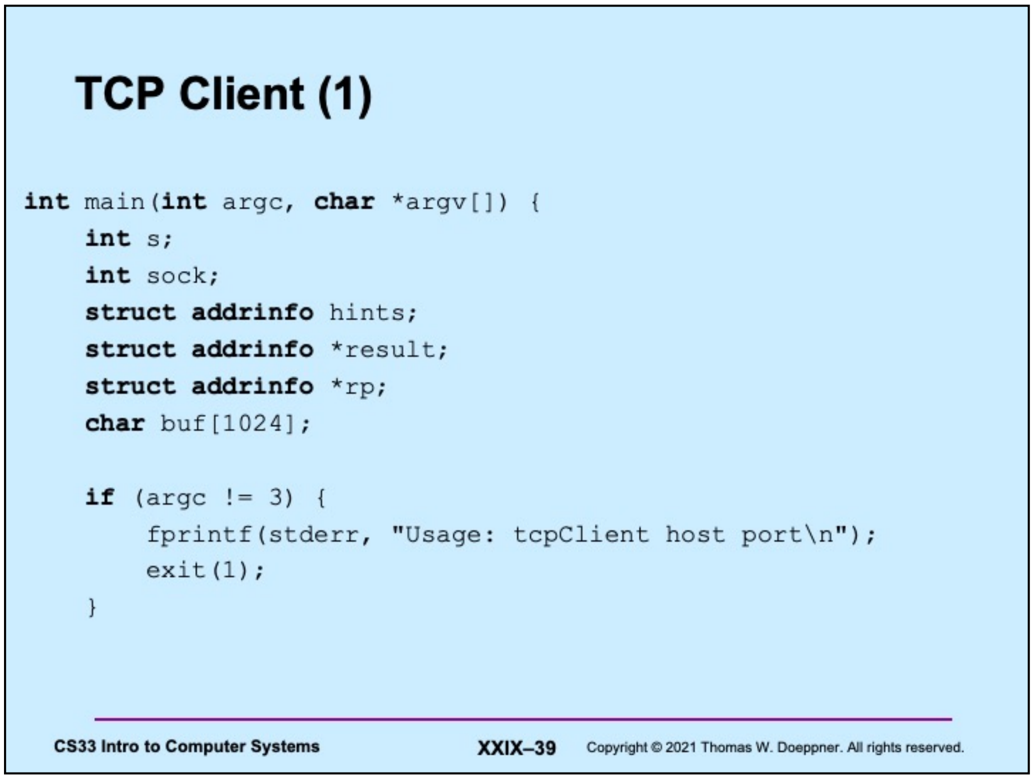
TCP Server (8)

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

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

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Finally, we have the *serve* function, which reads incoming data from the client and write it to file descriptor 1.

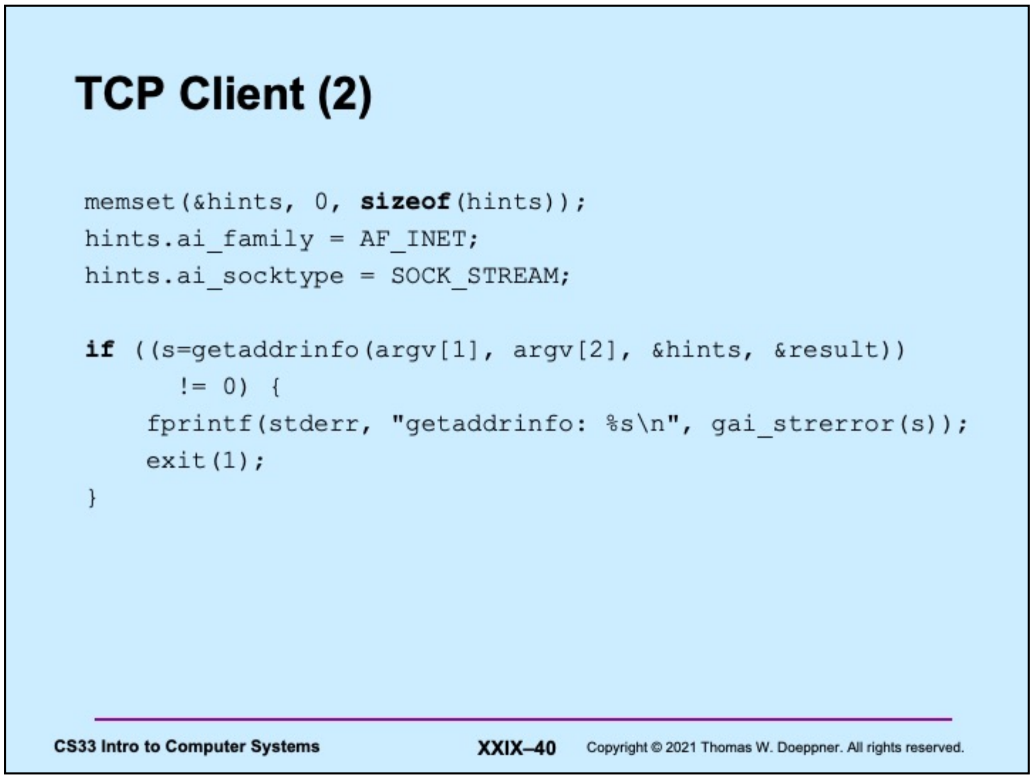


TCP Client (1)

```
int main(int argc, char *argv[]) {
    int s;
    int sock;
    struct addrinfo hints;
    struct addrinfo *result;
    struct addrinfo *rp;
    char buf[1024];

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

And lastly we have the code for our TCP client.



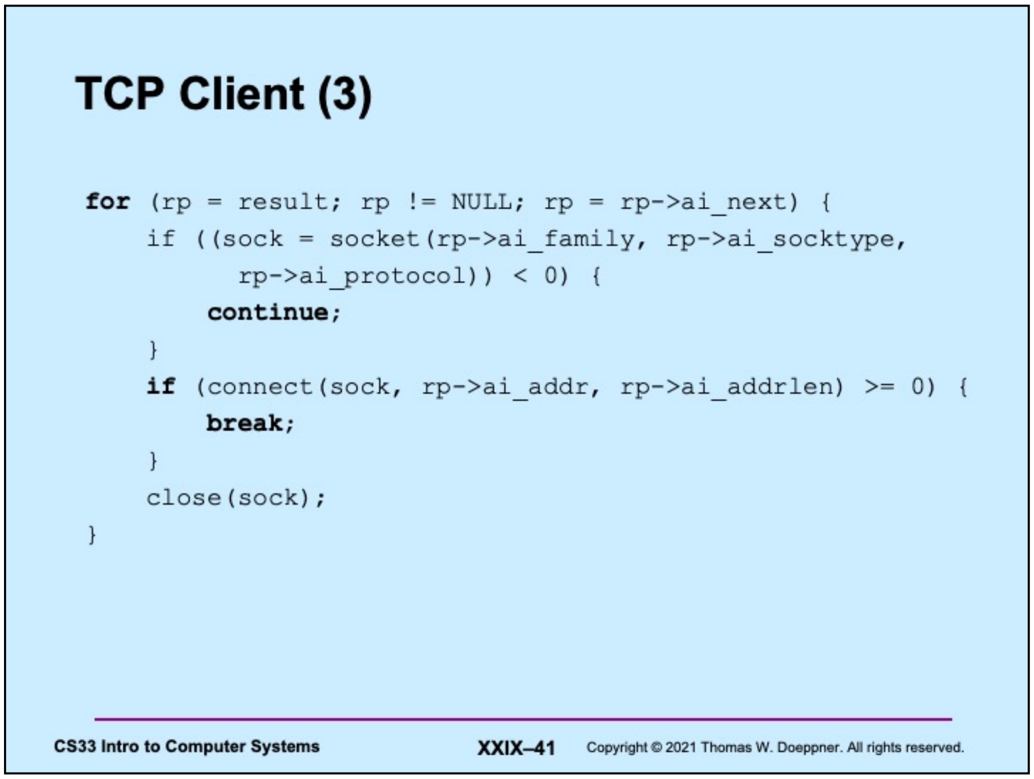
TCP Client (2)

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
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);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The client begins by looking up, via **getaddrinfo**, possible addresses for the server.

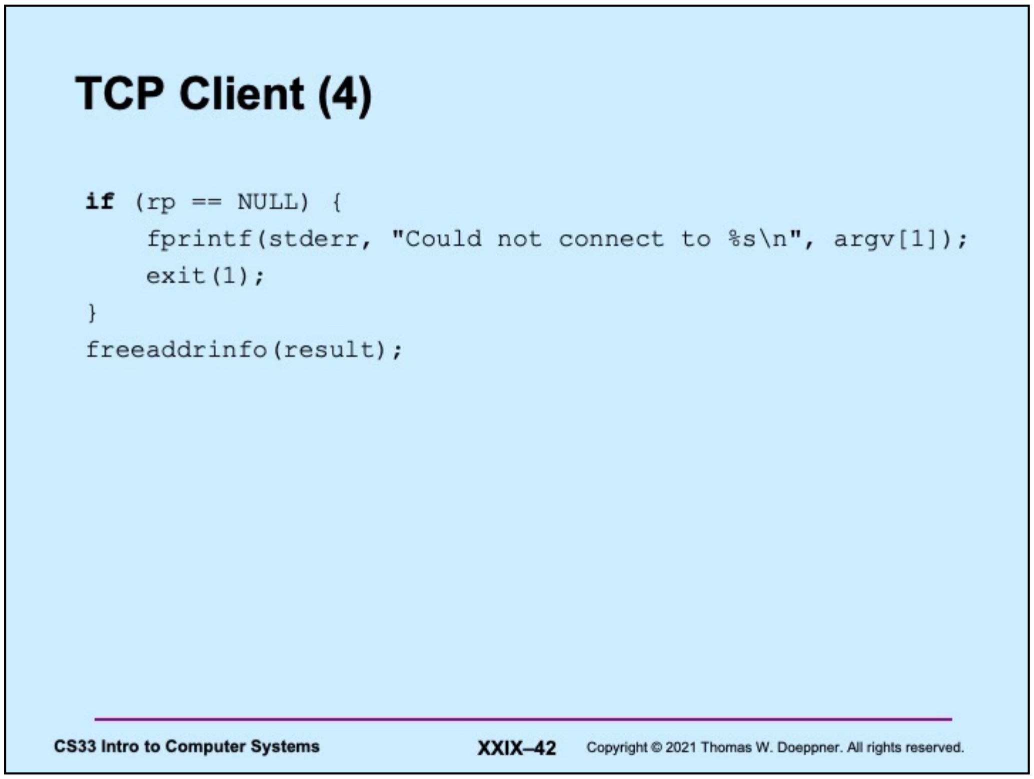


TCP Client (3)

```
for (rp = result; rp != NULL; rp = rp->ai_next) {
    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);
}
```

The client chooses an address for which it can create a socket and connect to. Thus, if this code completes successfully, the client is now connected to the server via *sock*.

Note that no port number (or service) is associated with the client's socket. Usually what port the client is using is unimportant and one is assigned arbitrarily when the client calls connect. If it's important that the client's socket have a particularly port associated with it, **bind** can be called on the socket before its used for communication.

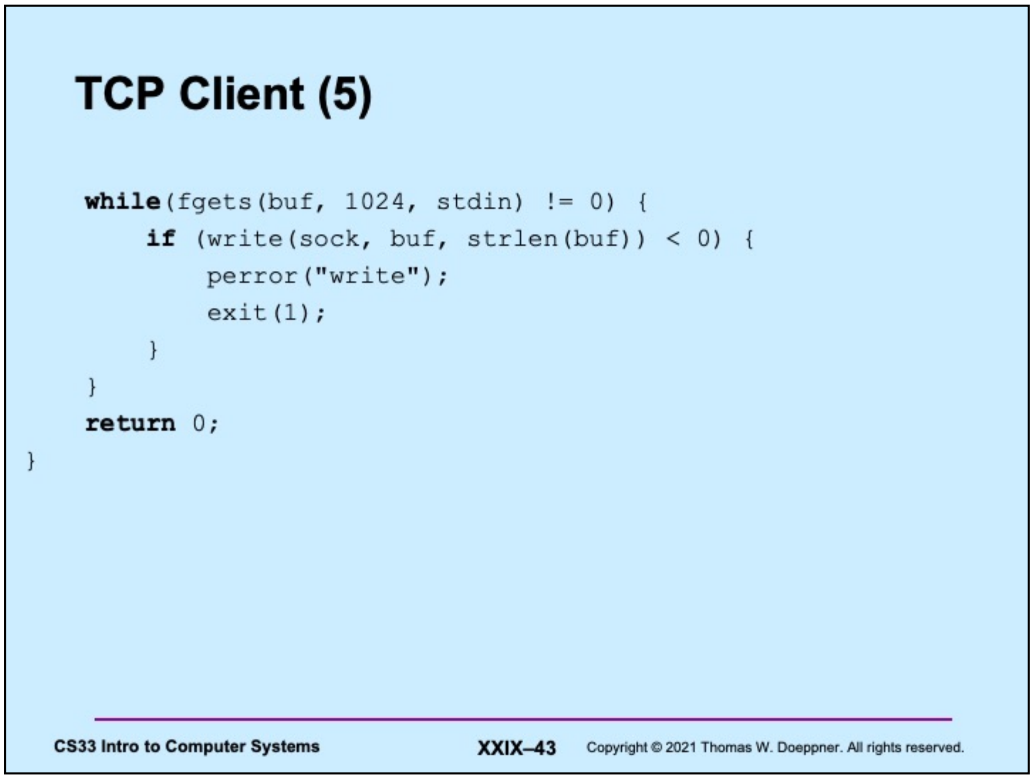


TCP Client (4)

```
if (rp == NULL) {
    fprintf(stderr, "Could not connect to %s\n", argv[1]);
    exit(1);
}
freeaddrinfo(result);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

If no satisfactory address was found, the client terminates. Otherwise it frees up the no-longer-needed list of addresses.



TCP Client (5)

```
    while(fgets(buf, 1024, stdin) != 0) {
        if (write(sock, buf, strlen(buf)) < 0) {
            perror("write");
            exit(1);
        }
    }
    return 0;
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Finally, the clients reads from stdin and sends whatever it reads to the server.

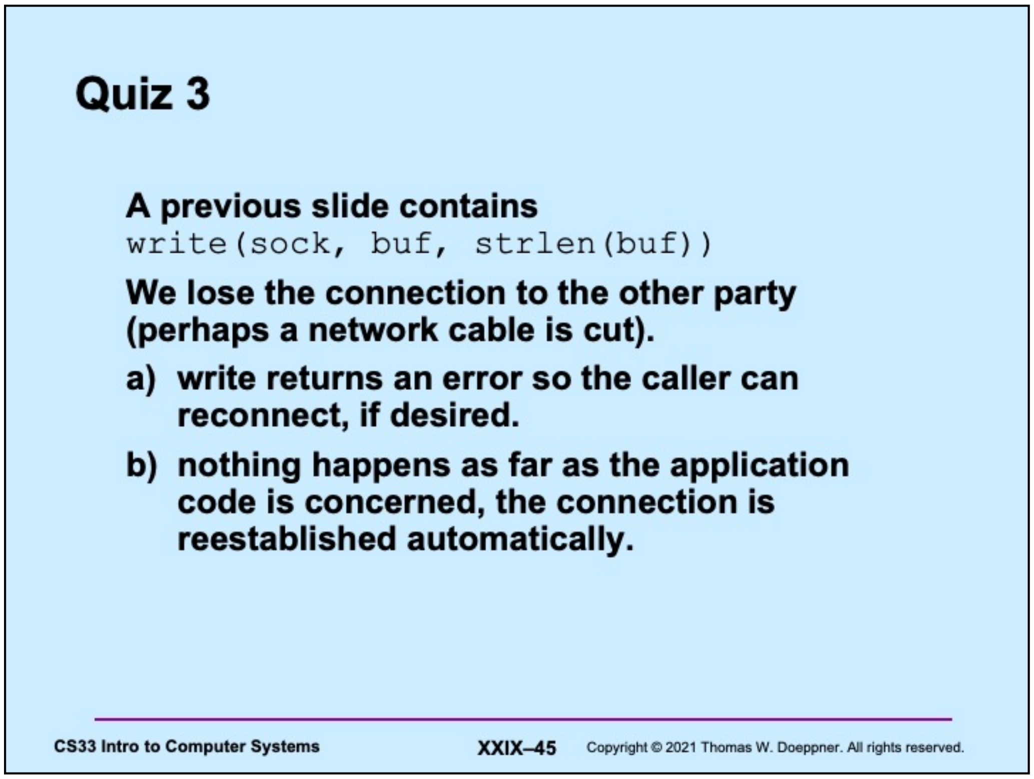
Quiz 2

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.

Copyright © 2021 Thomas W. Doeppner. All rights reserved.



Quiz 3

A 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.

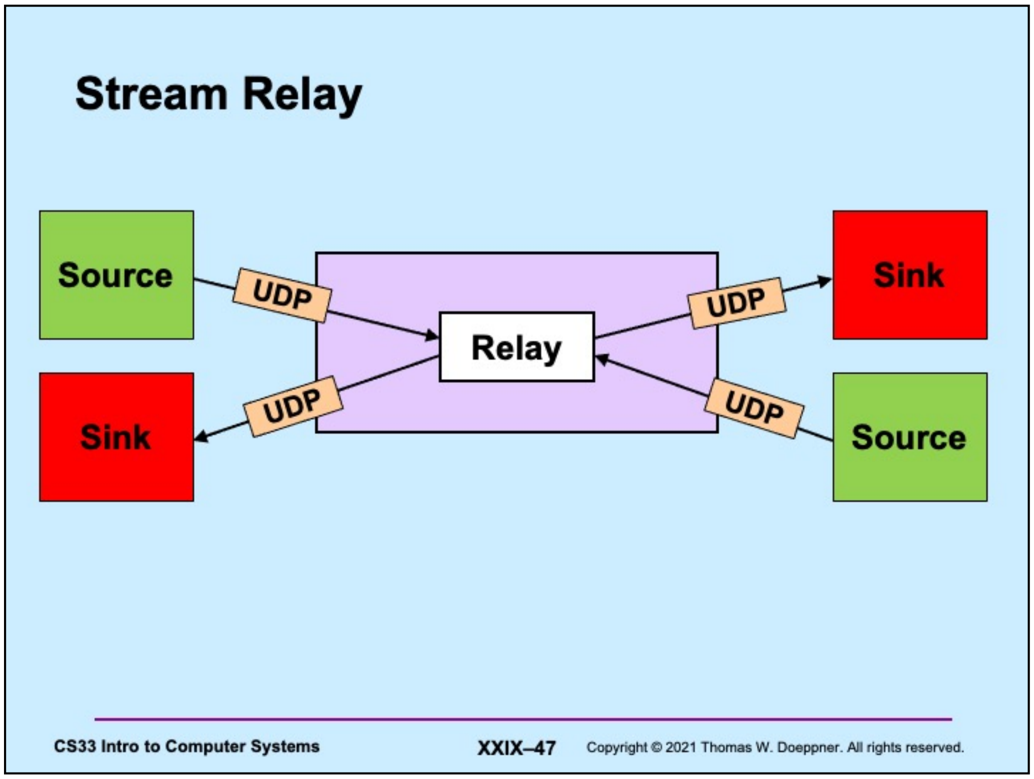
b) nothing happens as far as the application code is concerned, the connection is reestablished automatically.

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

CS 33

Event-Based Programming

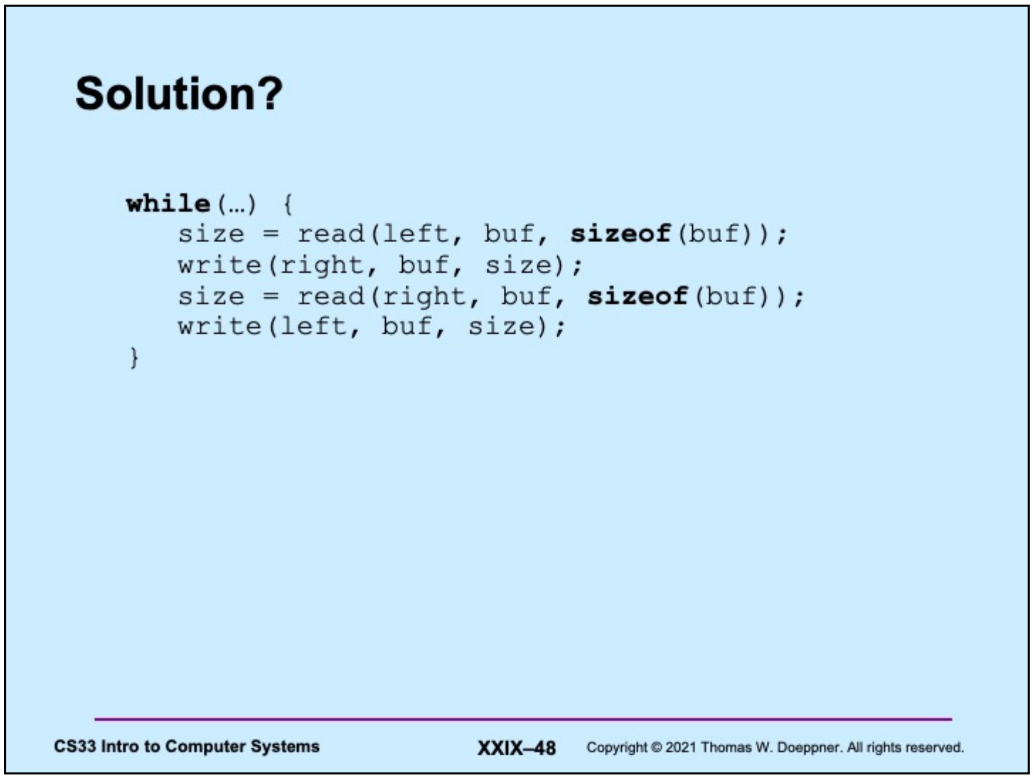
Copyright © 2021 Thomas W. Doeppner. All rights reserved.



Stream Relay

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We look at what's known as **event-based programming**: we write code that responds to events coming from a number of sources. As a simple example we examine a simple relay: we want to write a program that takes data received via UDP from a source on the left and forwards it (via UDP) to a sink on the right. At the same time, it's taking data received from a source on the right and forwards it (via UDP) to a sink on the left.



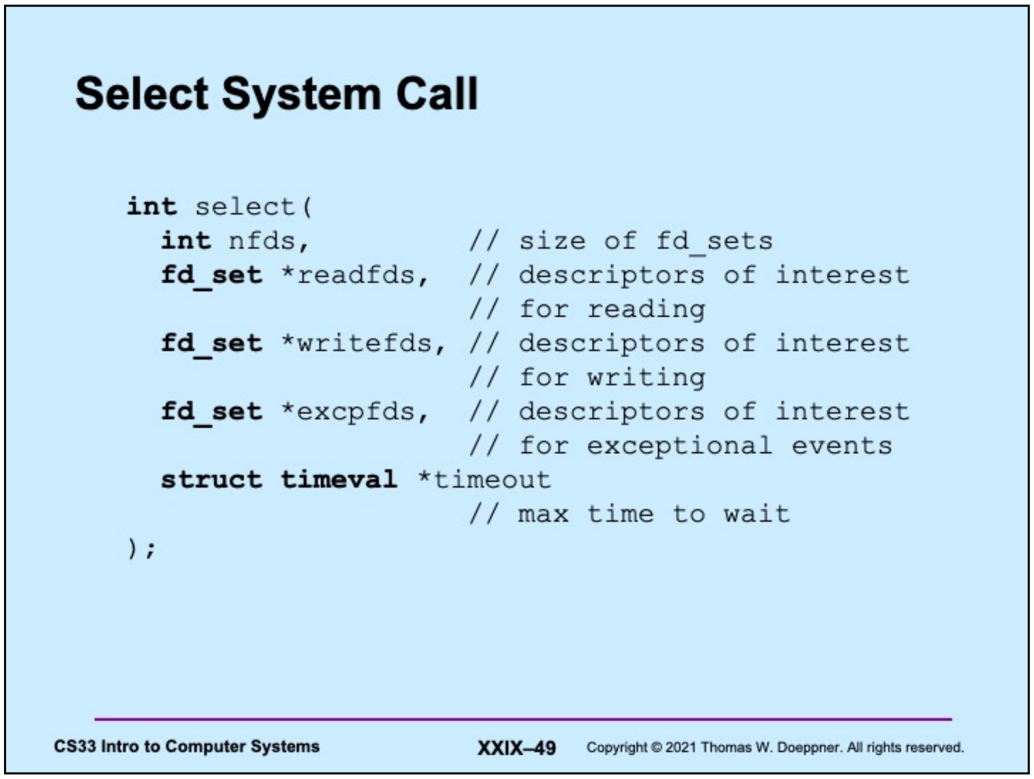
Solution?

```
while(…) {
   size = read(left, buf, sizeof(buf));
   write(right, buf, size);
   size = read(right, buf, sizeof(buf));
   write(left, buf, size);
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

This solution is probably not what we'd want, since it strictly alternates between processing the data stream in one direction and then the other.

Note that to simply the slides a bit, even though we're using UDP, we'll use read and write system calls – the source and destination are assumed in each case.

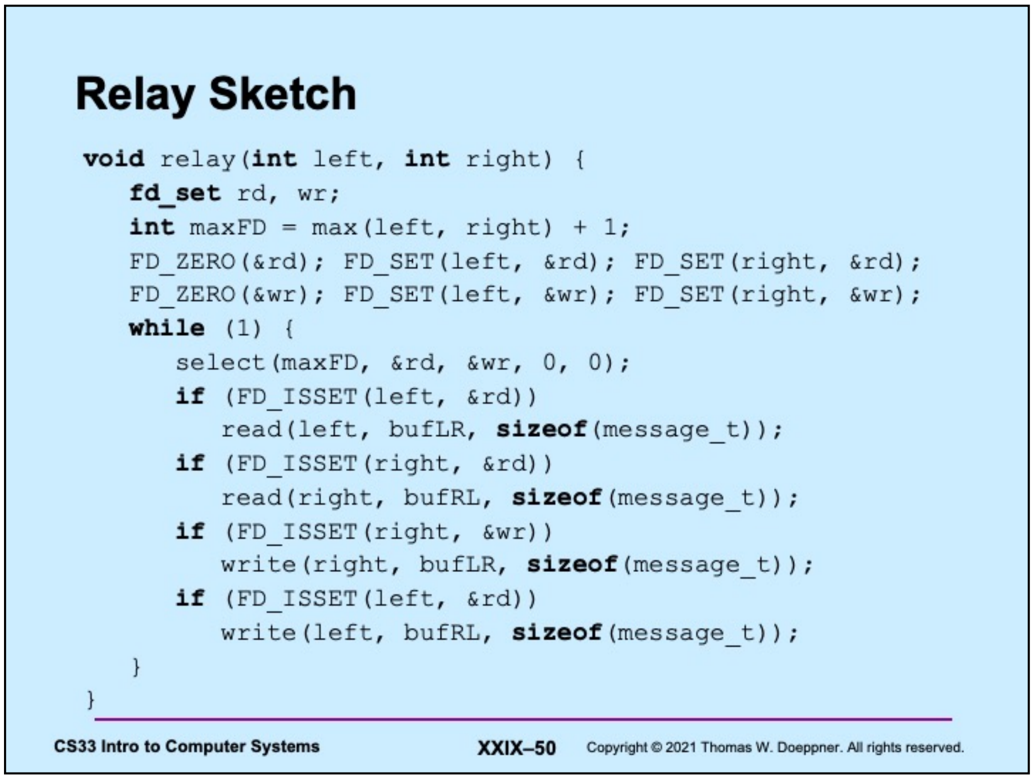


Select System Call

```
int select(
  int nfds,           // size of fd_sets
  fd_set *readfds,    // descriptors of interest
                      // for reading
  fd_set *writefds,   // descriptors of interest
                      // for writing
  fd_set *excpfds,    // descriptors of interest
                      // for exceptional events
  struct timeval *timeout
                      // max time to wait
);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

The **select** system call operates on three sets of file descriptors: one of fie descriptors we're interested in reading from, one of file descriptors we're interested in writing to, and one of file descriptors that might have exceptional conditions pending (we haven't covered any examples of such things – they come up as a peculiar feature of TCP known as out-of-band data, which is beyond the scope of this course). A call to **select** waits until at least one of the file descriptors in the given sets has something of interest. In particular, for a file descriptor in the read set, it's possible to read data from it; for a file descriptor in the write set, it's possible to write data to it. The **nfds** parameter indicates the maximum file descriptor number in any of the sets. The **timeout** parameter may be used to limit how long **select** waits. If set to zero, select waits indefinitely.



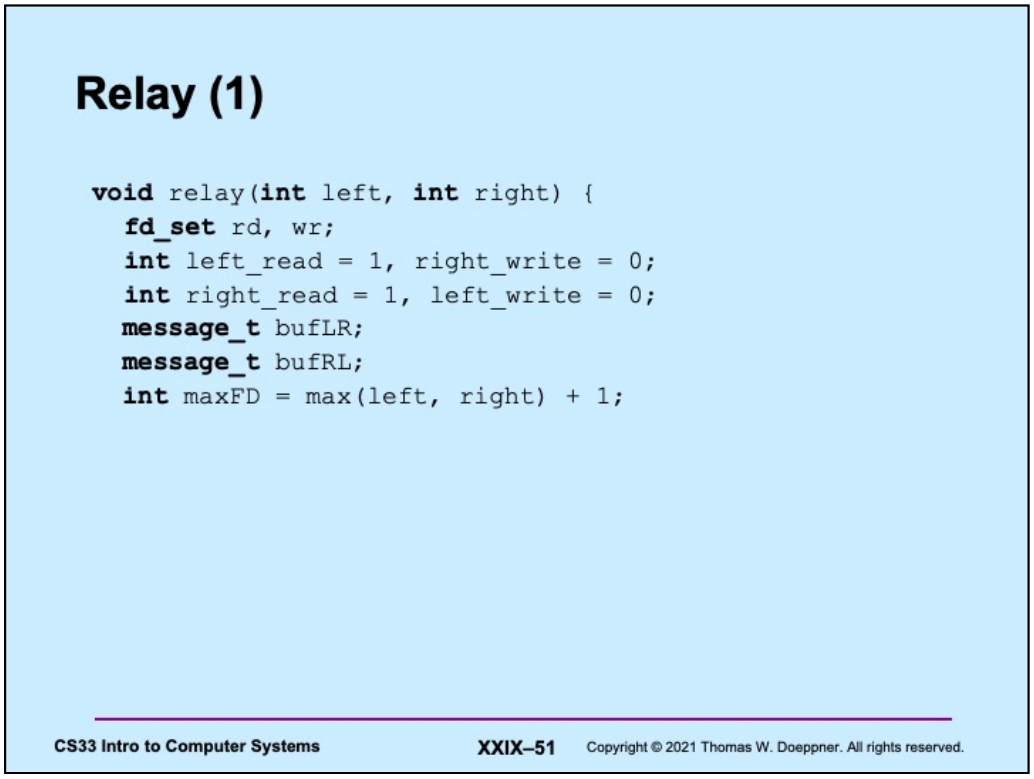
Relay Sketch

```
void relay(int left, int right) {
   fd_set rd, wr;
   int maxFD = max(left, right) + 1;
   FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
   FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
   while (1) {
      select(maxFD, &rd, &wr, 0, 0);
      if (FD_ISSET(left, &rd))
         read(left, bufLR, sizeof(message_t));
      if (FD_ISSET(right, &rd))
         read(right, bufRL, sizeof(message_t));
      if (FD_ISSET(right, &wr))
         write(right, bufLR, sizeof(message_t));
      if (FD_ISSET(left, &rd))
         write(left, bufRL, sizeof(message_t));
   }
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

Here a simplified version of a program to handle the relay problem using *select*. An **fd_set** is a data type that represents a set of file descriptors. FD_ZERO, FD_SET, and FD_ISSET are macros for working with fd_sets; the first makes such a set represent the null set, the second sets a particular file descriptor to be included in the set, the last checks to see if a particular file descriptor is included in the set.

This sketch doesn't quite work because it doesn't take into account the fact that we have limited buffer space: we can't read two messages in a row from one side without writing the first to the other side before reading the second. Furthermore, even though select may say it's possible to write to either the left or the right side, we can't do so until we're read in some data from the other side. Also, the fd_sets that are select's arguments are modified on return from select to indicate if it's now possible to read or write on the associated file descriptor. Thus if, on return from select, it's not possible to use that file descriptor, its associated bit will be zero. We need to explicitly set it to one for the next call so that select knows we're still interested.



Relay (1)

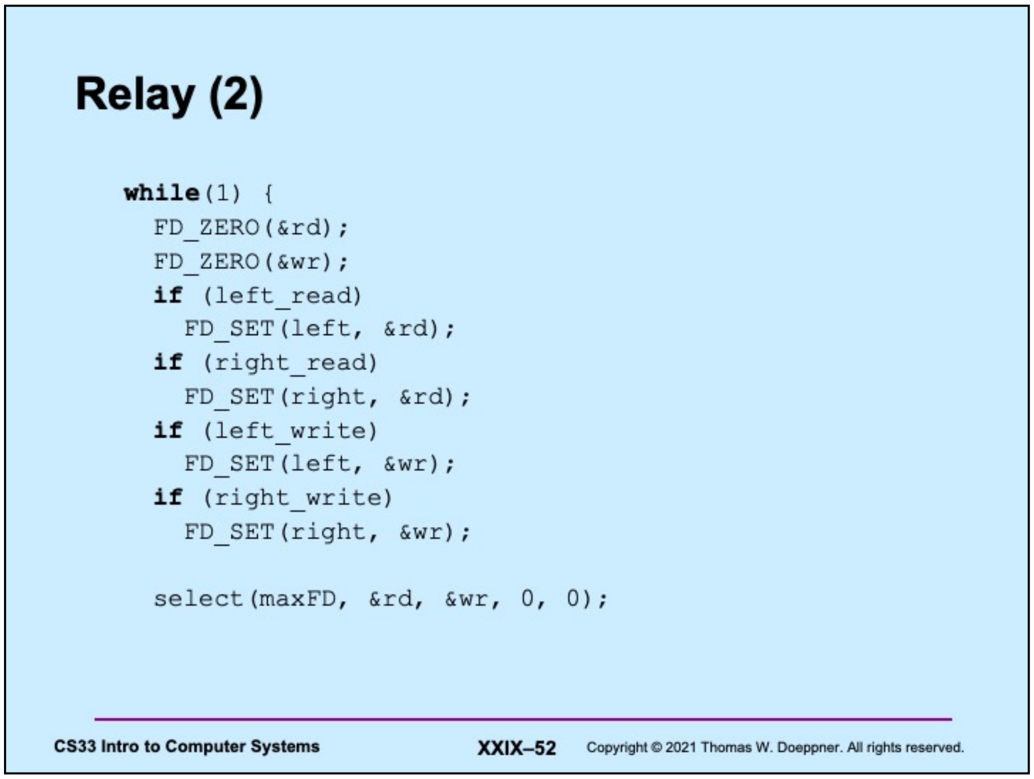
```
void relay(int left, int right) {
  fd_set rd, wr;
  int left_read = 1, right_write = 0;
  int right_read = 1, left_write = 0;
  message_t bufLR;
  message_t bufRL;
  int maxFD = max(left, right) + 1;
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

This and the next three slides give a more complete version of the relay program.

Initially our program is prepared to read from either the left or the right side, but it's not prepared to write, since it doesn't have anything to write. The variables **left_read** and **right_read** are set to one to indicate that we want to read from the left and right sides. The variables **right_write** and **left_write** are set to zero to indicate that we don't yet want to write to either side.

The two variables of type **message_t** are used as buffers to hold a messages received from the left and to be written to the right, or vice versa.



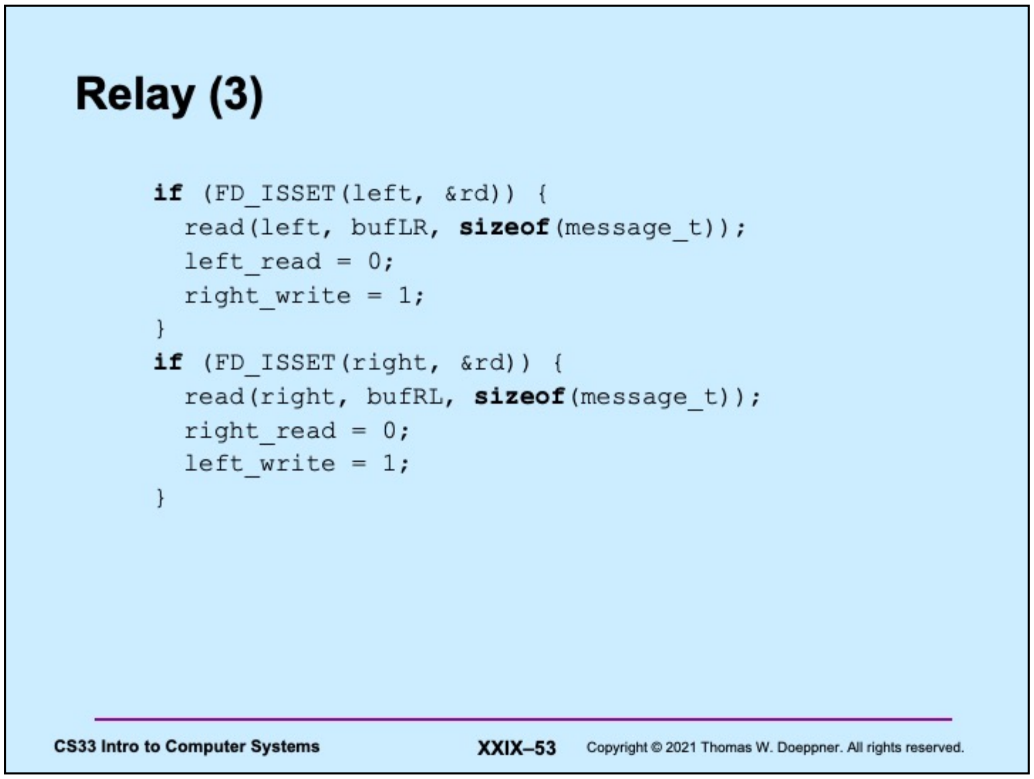
Relay (2)

```
while(1) {
  FD_ZERO(&rd);
  FD_ZERO(&wr);
  if (left_read)
    FD_SET(left, &rd);
  if (right_read)
    FD_SET(right, &rd);
  if (left_write)
    FD_SET(left, &wr);
  if (right_write)
    FD_SET(right, &wr);

  select(maxFD, &rd, &wr, 0, 0);
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

We set up the fd_sets **rd** and **wr** to indicate what we are interested in reading from and writing to (initially we have no interest in writing, but are interested in reading from either side).



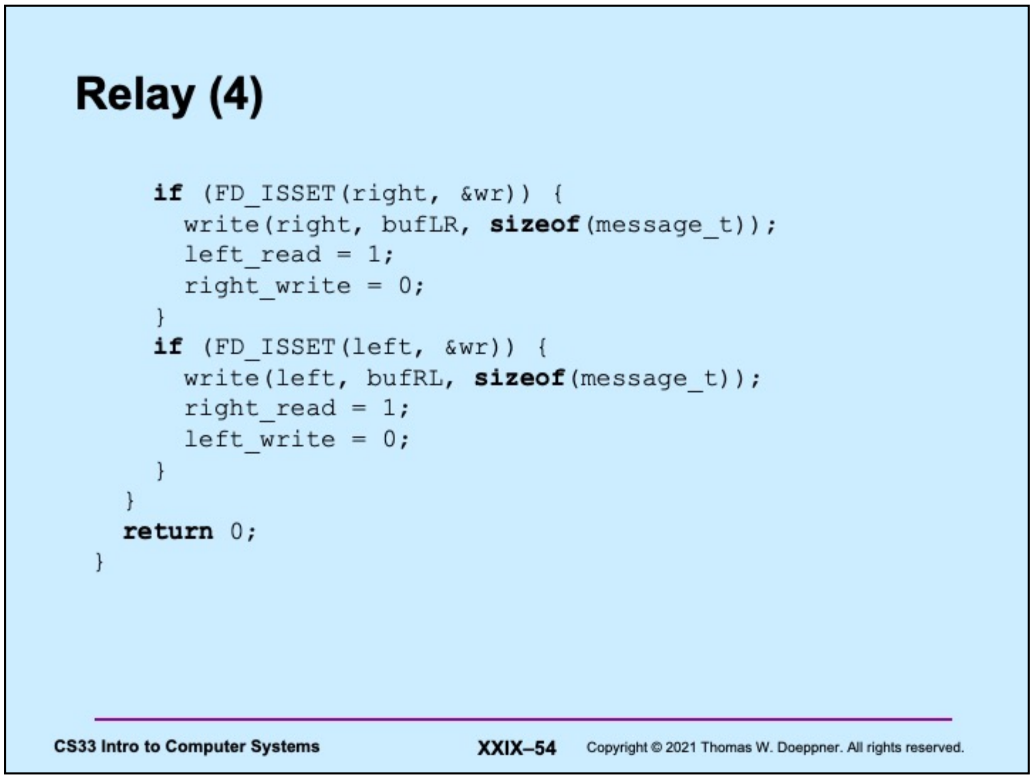
Relay (3)

```
if (FD_ISSET(left, &rd)) {
  read(left, bufLR, sizeof(message_t));
  left_read = 0;
  right_write = 1;
}
if (FD_ISSET(right, &rd)) {
  read(right, bufRL, sizeof(message_t));
  right_read = 0;
  left_write = 1;
}
```

Copyright © 2021 Thomas W. Doeppner. All rights reserved.

If there is something to read from the left side, we read it. Having read it, we're temporarily not interested in reading anything further from the left side, but now want to write to the right side.

In a similar fashion, if there is something to read from the right side, we read it.



Relay (4)

```
      if (FD_ISSET(right, &wr)) {
        write(right, bufLR, sizeof(message_t));
        left_read = 1;
        right_write = 0;
      }
      if (FD_ISSET(left, &wr)) {
        write(left, bufRL, sizeof(message_t));
        right_read = 1;
        left_write = 0;
      }
    }
    return 0;
  }
```

Similarly for writing: if we've written something to one side, we have nothing more to write to that side, but are now interested in reading from the other side.