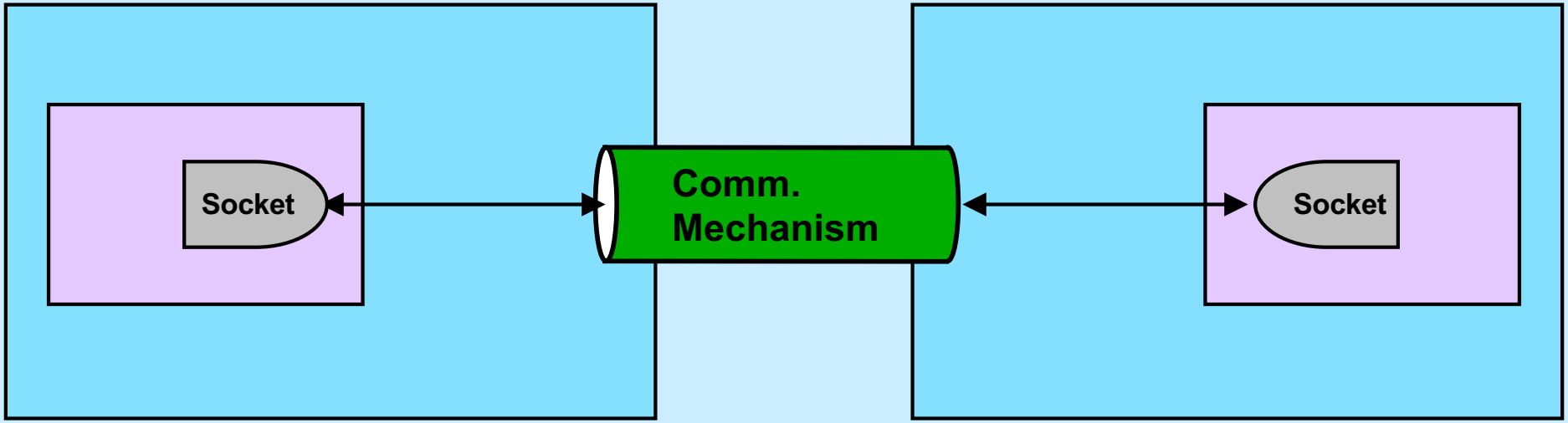
CS 33

Network Programming (2)

Sockets



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

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

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

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

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**
  ```

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;
```

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);
}
```

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);
}
```

UDP Server (4)

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

freeaddrinfo(result);
```

UDP Server (5)

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

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;
```

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);
```

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);
        }
    }
}
```

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);
        }
```

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);
}
```

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);
```

UDP Client (4)

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

    return 0;

}
```


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;
```

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;
}
```

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;
}
```

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

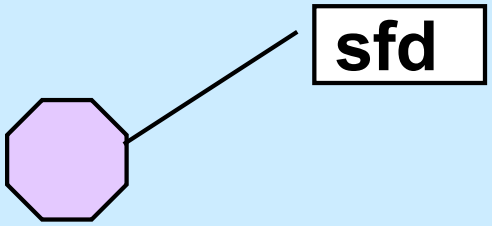
Reliable Communication

- 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

Streams in the Inet Domain (1)

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

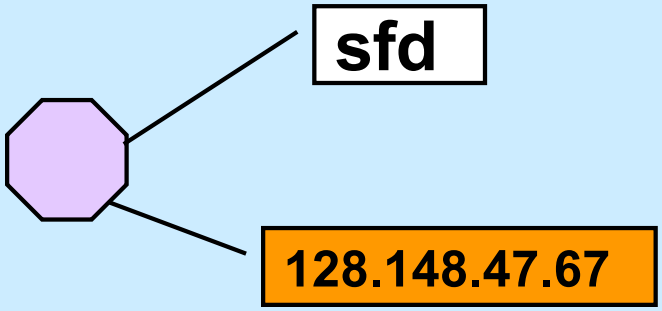
```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (2)

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

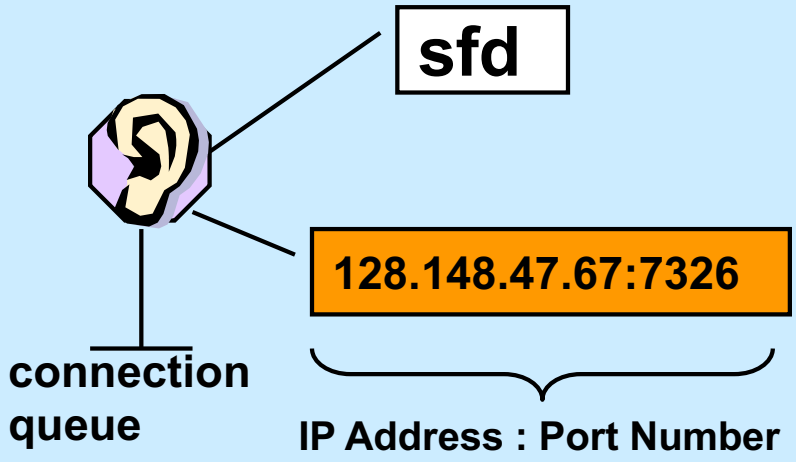
```
bind(sfd,
   (struct sockaddr *)&my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (3)

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

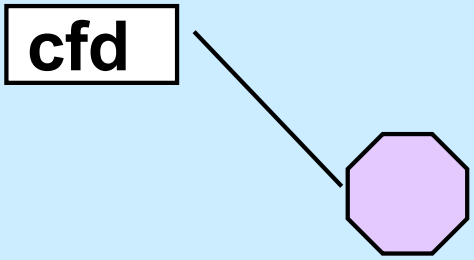
```
int listen(int sfd, int MaxQueueLength);
```



Streams in the Inet Domain (4)

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

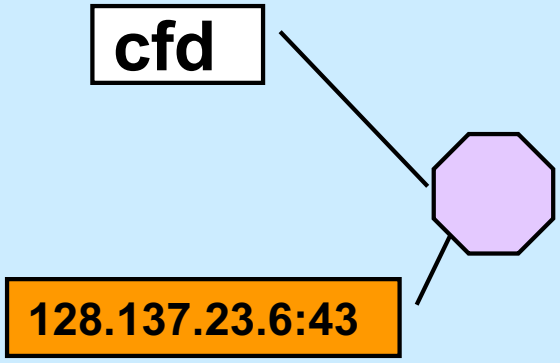
```
cfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (5)

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

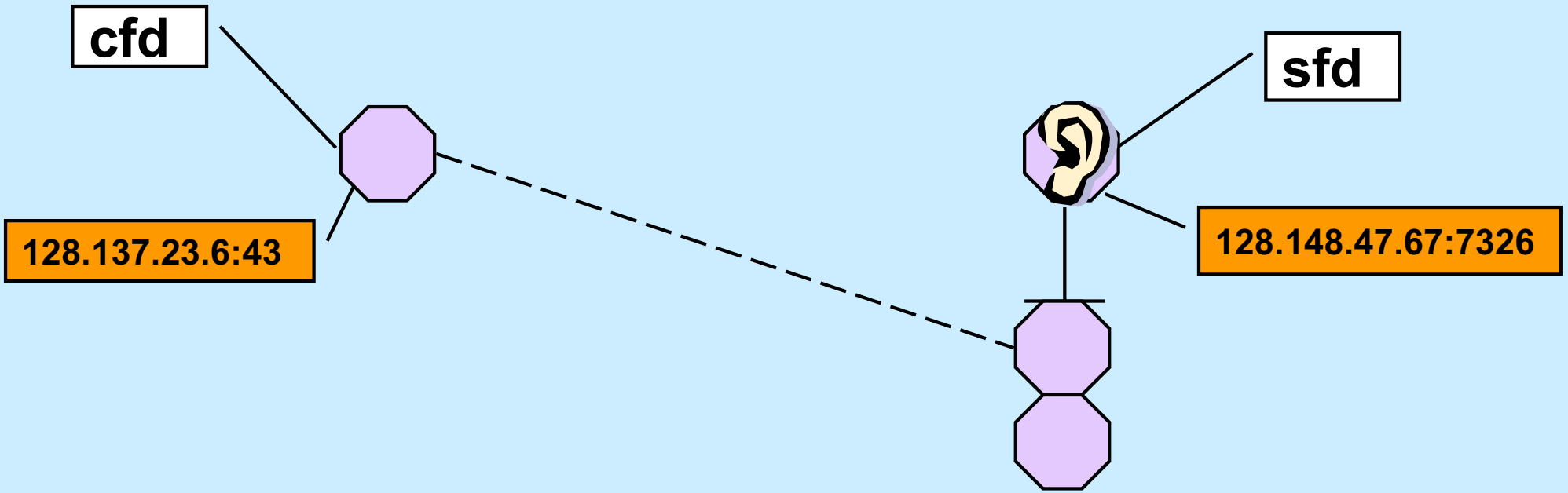
```
bind(cfd,
   (struct sockaddr *)&my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (6)

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

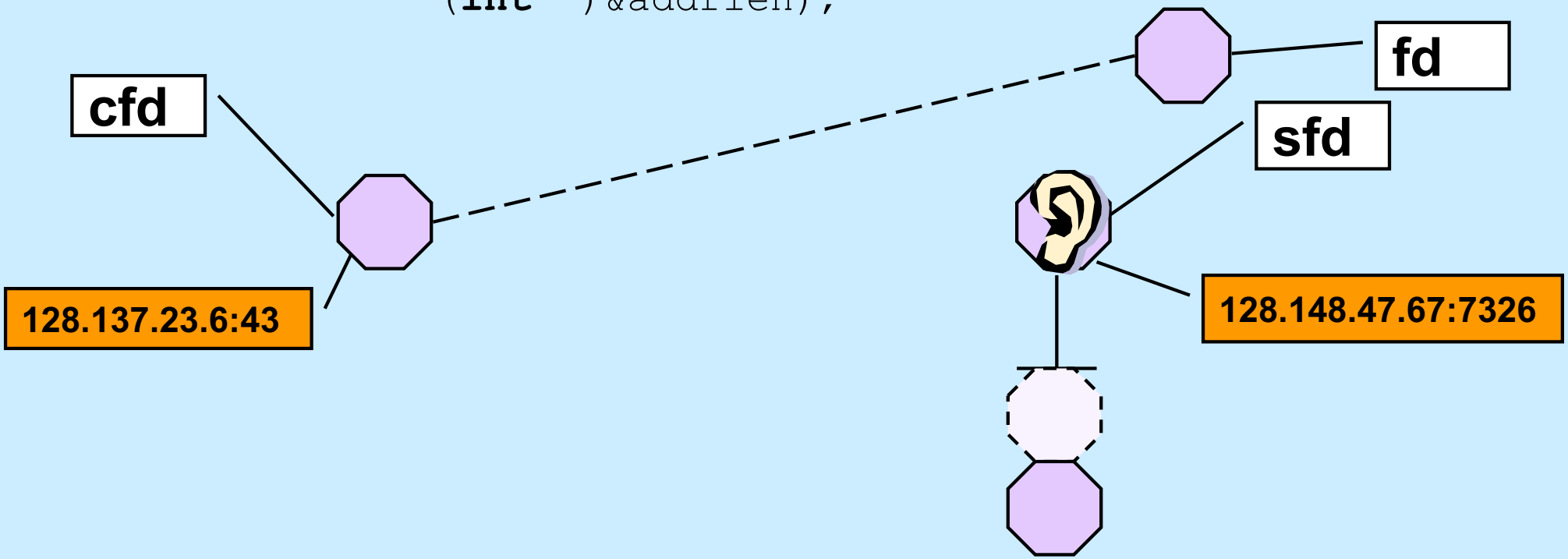
```
connect(cfd, (struct sockaddr *)&server_addr,
    sizeof(server_addr));
```



Streams in the Inet Domain (7)

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

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



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;
```

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);
}
```

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);
}
```

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);
}
```

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);
    }
```

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);
```

TCP Server (7)

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

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");
}
```

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);
    }
```

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);
}
```

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);
}
```

TCP Client (4)

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

TCP Client (5)

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

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.

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).

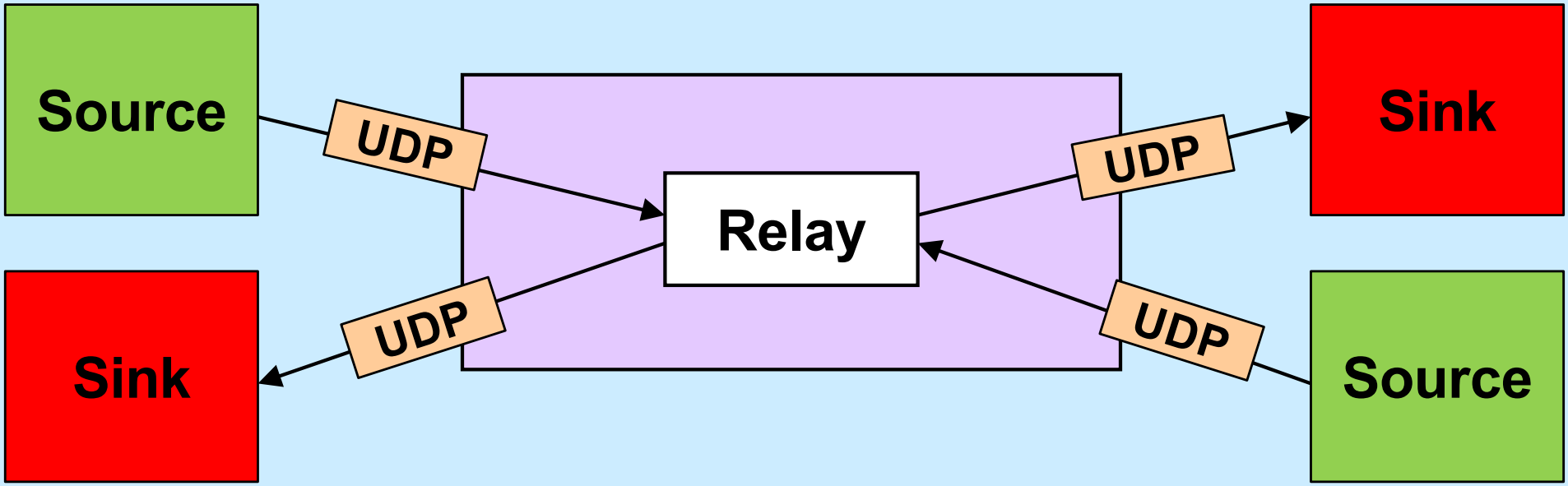
a) write returns an error so the caller can reconnect, if desired.

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

CS 33

Event-Based Programming

Stream Relay



Solution?

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

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
);
```

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));
    }
}
```

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;
```

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);
```

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;
}
```

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;
}
```