Network Programming

15-213 / 18-213: Introduction to Computer Systems
21st Lecture, Nov. 7, 2013

Instructors:

Randy Bryant, Dave O'Hallaron, and Greg Kesden

A Programmer’s View of the Internet

- **Hosts are mapped to a set of 32-bit *IP addresses***
  - 128.2.217.13

- **The set of IP addresses is mapped to a set of identifiers called Internet *domain names***
  - 128.2.217.13 is mapped to www.cs.cmu.edu

- **A process on one Internet host can communicate with a process on another Internet host over a *connection***

Internet Connections

- **Clients and servers communicate by sending streams of bytes over *connections*:**
  - Point-to-point, full-duplex (2-way communication), and reliable

- ***A socket* is an endpoint of a connection**
  - Socket address is an `IPaddress:port` pair

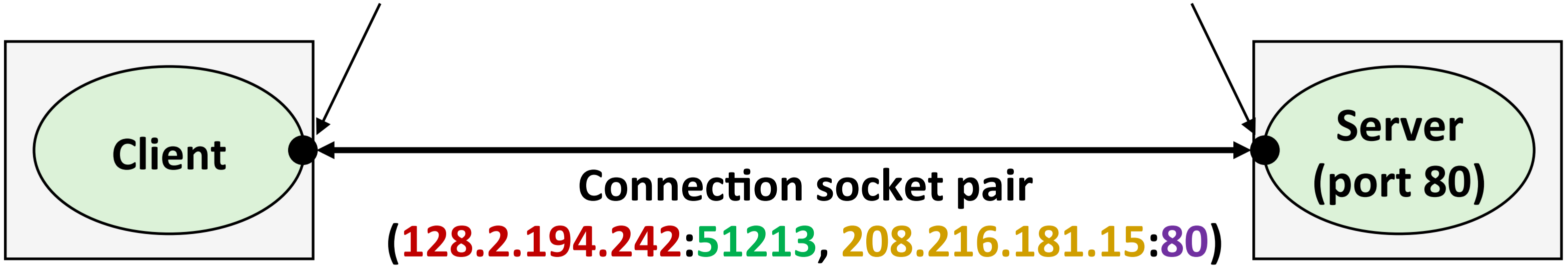
- **A *port* is a 16-bit integer that identifies a process:**
  - ***Ephemeral port***: Assigned automatically on client when client makes a connection request
  - ***Well-known port:*** Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

- **A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)**
  - `(cliaddr:cliport, servaddr:servport)`

Anatomy of an Internet Connection

Client socket address
128.2.194.242:51213

Server socket address
208.216.181.15:80



Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

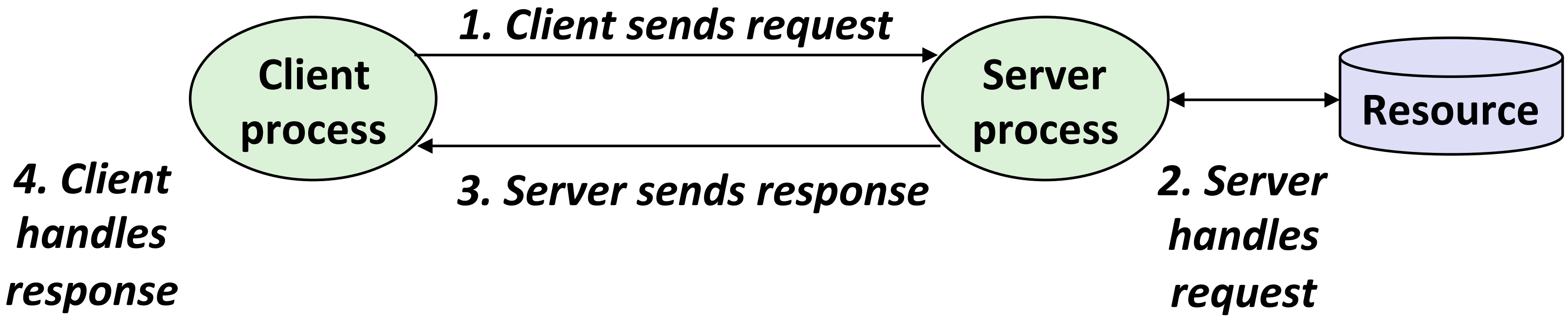
Client host address
128.2.194.242

Server host address
208.216.181.15

51213 is an ephemeral port
allocated by the kernel

80 is a well-known port
associated with Web servers

A Client-Server Transaction



Note: clients and servers are processes running on hosts (can be the same or different hosts)

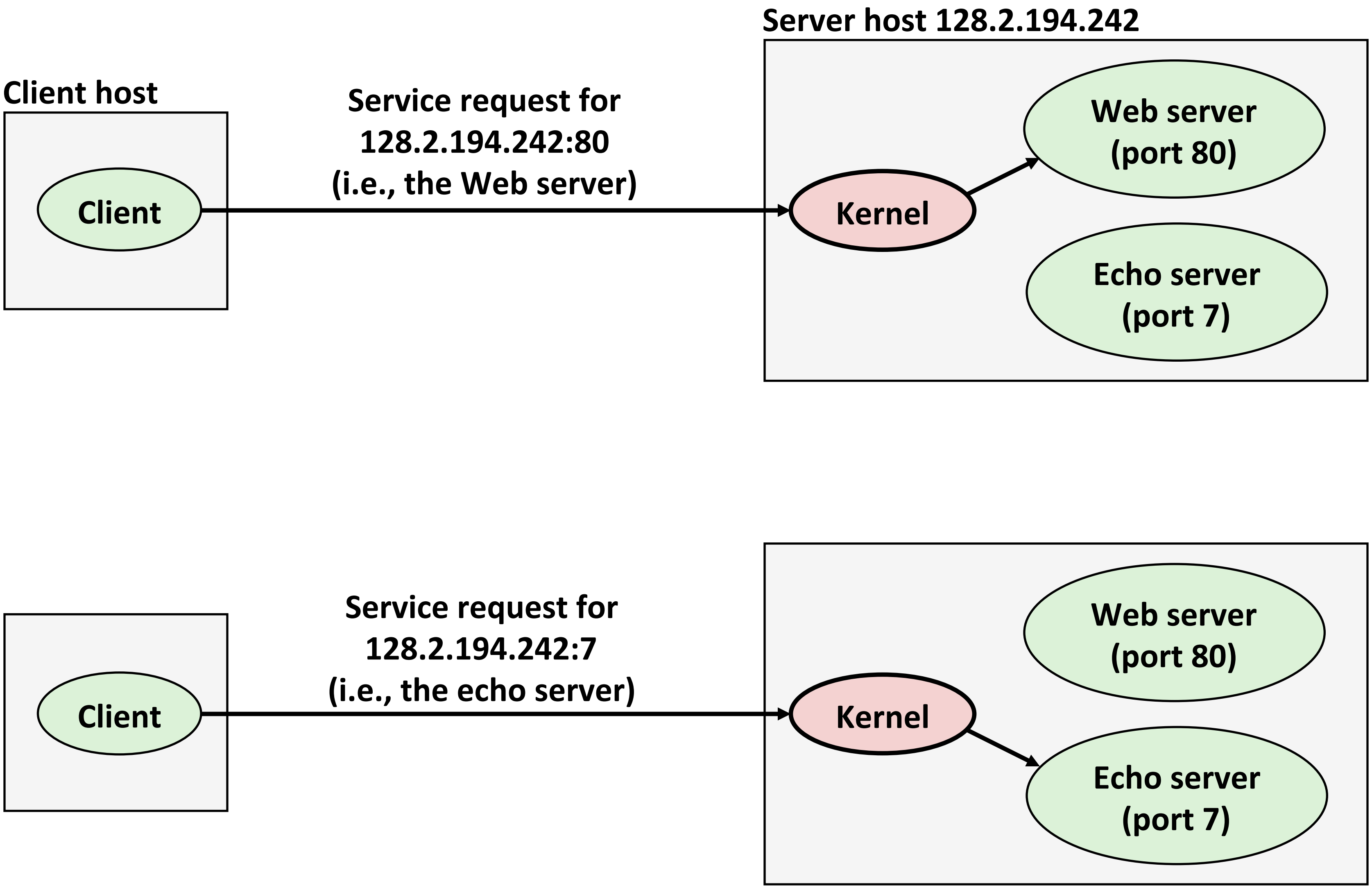
- **Most network applications are based on the client-server model:**
  - A ***server*** process and one or more ***client*** processes
  - Server manages some ***resource***
  - Server provides ***service*** by manipulating resource for clients
  - Server activated by request from client (vending machine analogy)

Clients

- **Examples of client programs**
  - Web browsers, **`ftp`**, **`telnet`**, **`ssh`**

- **How does a client find the server?**
  - The IP address in the server socket address identifies the host (more precisely, an adapter on the host)
  - The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
  - Examples of well known ports
    - Port 7: Echo server
    - Port 22: ssh server
    - Port 25: Mail server
    - Port 80: http server
    - Port 443: https server

Using Ports to Identify Services



Server host 128.2.194.242

Client host

Client

Service request for
128.2.194.242:80
(i.e., the Web server)

Kernel

Web server
(port 80)

Echo server
(port 7)

Client

Service request for
128.2.194.242:7
(i.e., the echo server)

Kernel

Web server
(port 80)

Echo server
(port 7)

Servers

- **Servers are long-running processes (daemons)**
  - Created at boot-time (typically) by the init process (process 1)
  - Run continuously until the machine is turned off

- **Each server waits for requests to arrive on a well-known port associated with a particular service**
  - Port 7: echo server
  - Port 22: ssh server
  - Port 25: mail server
  - Port 80: HTTP server

- **A machine that runs a server process is also often referred to as a "server"**

Server Examples

- **Web server (port 80)**
  - Resource: files/compute cycles (CGI programs)
  - Service: retrieves files and runs CGI programs on behalf of the client
- **ssh server (22)**
  - Resource: terminal
  - Service: proxies a terminal on the server machine and transfers files
- **Mail server (25)**
  - Resource: email “spool” file
  - Service: stores mail messages in spool file

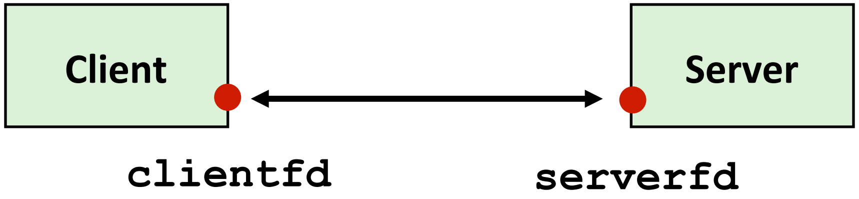
See `/etc/services` for a comprehensive list of the port mappings on a Linux machine

Sockets Interface

- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols
- Provides a user-level interface to the network
- Underlying basis for all Internet applications
- Based on client/server programming model

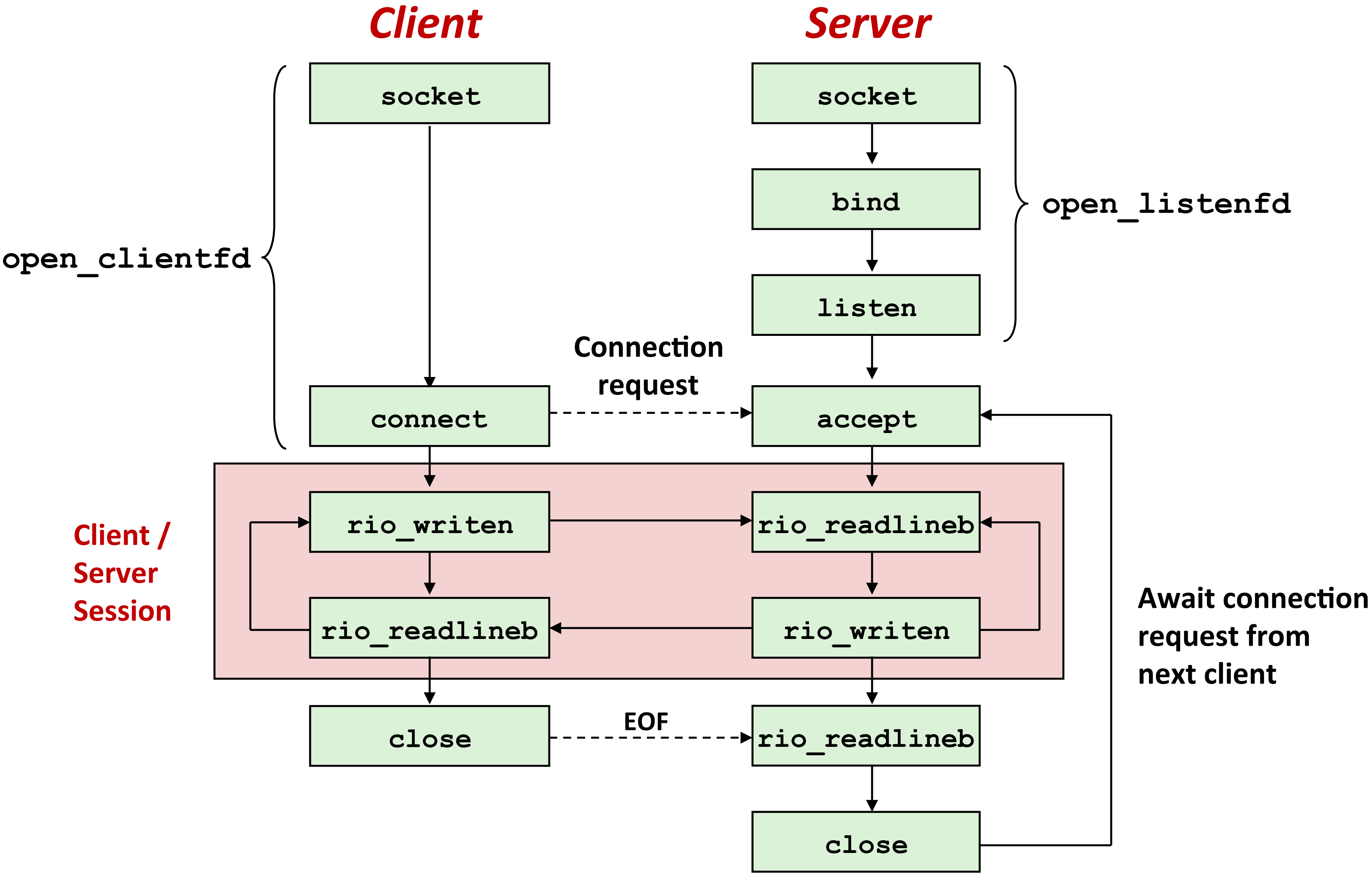
Sockets

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - ***Remember:*** All Unix I/O devices, including networks, are modeled as files
- **Clients and servers communicate with each other by reading from and writing to socket descriptors**



- **The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors**

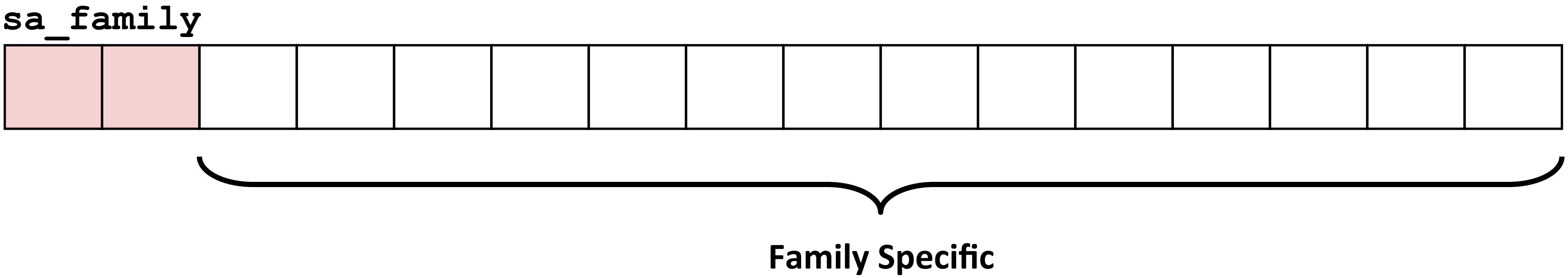
Overview of the Sockets Interface



Socket Address Structures

- **Generic socket address:**
  - For address arguments to **`connect`**, **`bind`**, and **`accept`**
  - Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed

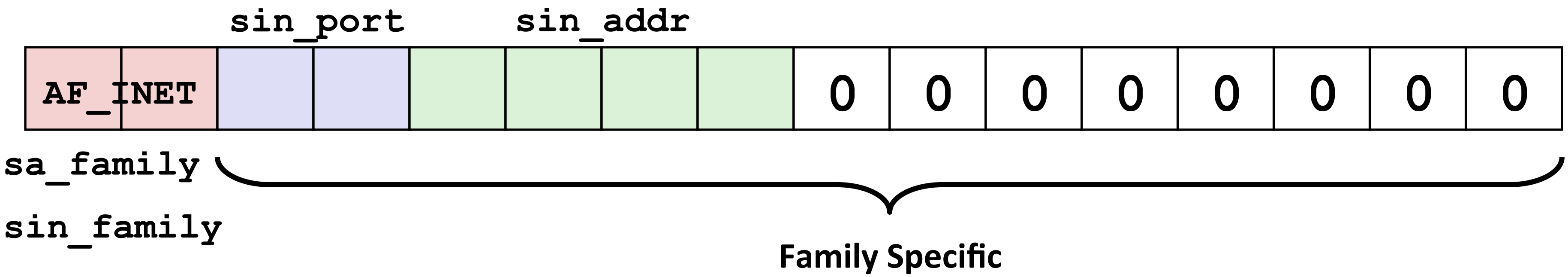
```
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data.  */
};
```



Socket Address Structures

- **Internet-specific socket address:**
  - Must cast (**`sockaddr_in *`**) to (**`sockaddr *`**) for **`connect`**, **`bind`**, and **`accept`**

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
  unsigned short  sin_port;    /* port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```



sin_port

sin_addr

AF_INET

0 0 0 0 0 0 0 0

sa_family

sin_family

Family Specific

Example: Echo Client and Server

On Client

On Server

```
greatwhite> ./echoserveri 15213
```

```
linux> echoclient greatwhite.ics.cs.cmu.edu 15213
```

```
server connected to BRYANT-TP4.VLSI.CS.CMU.EDU
(128.2.213.29), port 64690
```

```
type: hello there
```

```
server received 12 bytes
```

```
echo: HELLO THERE
type: ^D
```

```
Connection closed
```

Echo Client Main Routine

```
#include "csapp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;
    host = argv[1];  port = atoi(argv[2]);
    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);
    printf("type:"); fflush(stdout);
    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        printf("echo:");
        Fputs(buf, stdout);
        printf("type:"); fflush(stdout);
    }
    Close(clientfd);
    exit(0);
}
```

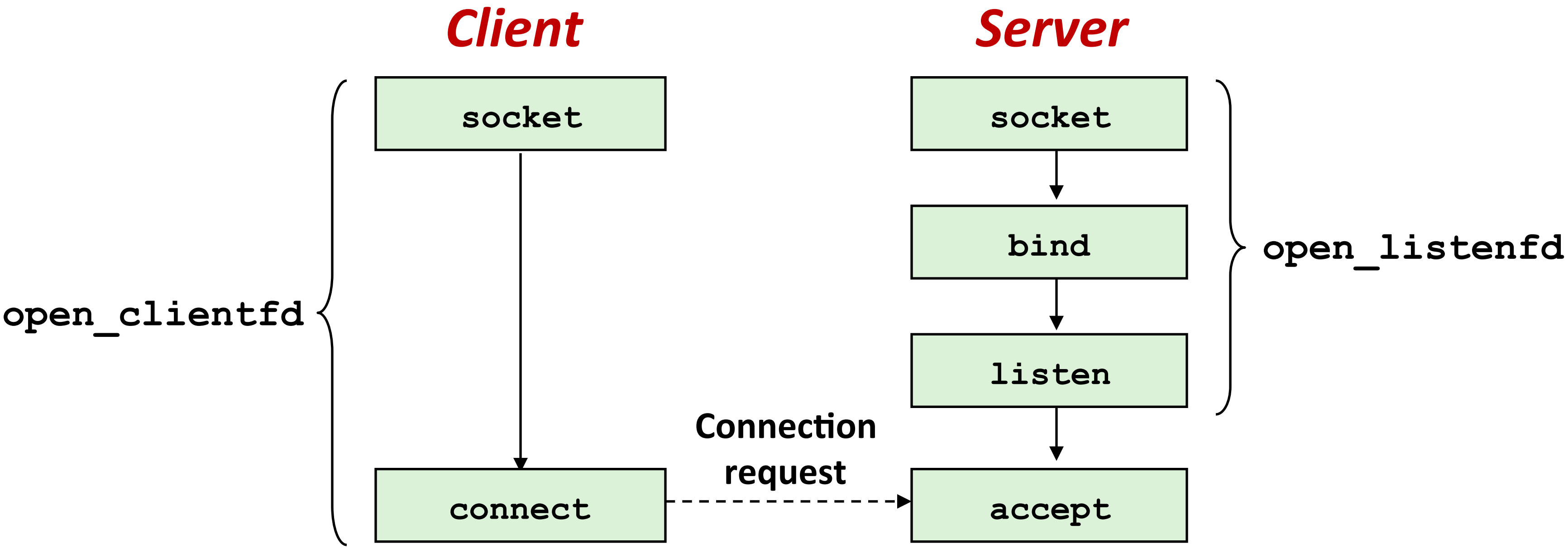
Read input line

Send line to server

Receive line from server

Print server response

Overview of the Sockets Interface



Echo Client: open_clientfd

```
int open_clientfd(char *hostname, int port) {
  int clientfd;
  struct hostent *hp;
  struct sockaddr_in serveraddr;

  if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

  /* Fill in the server's IP address and port */
  if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  bcopy((char *)hp->h_addr_list[0],
        (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
  serveraddr.sin_port = htons(port);

  /* Establish a connection with the server */
  if (connect(clientfd, (SA *) &serveraddr,
      sizeof(serveraddr)) < 0)
    return -1;
  return clientfd;
}
```

This function opens a connection from the client to the server at `hostname:port`

Create socket

Create address

Establish connection

Echo Client: open_clientfd (socket)

- **socket creates a socket descriptor on the client**
  - Just allocates & initializes some internal data structures
  - **AF_INET**: indicates that the socket is associated with Internet protocols
  - **SOCK_STREAM**: selects a reliable byte stream connection
    - provided by TCP

```
int clientfd;  /* socket descriptor */

if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

... <more>
```

Echo Client: open_clientfd (gethostbyname)

- The client then builds the server's Internet address

```
int clientfd;                  /* socket descriptor */
struct hostent *hp;            /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */

...

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
bcopy((char *)hp->h_addr_list[0],
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
```

Check this out!

A Careful Look at bcopy Arguments

```
/* DNS host entry structure */
struct hostent {
   . . .
   int    h_length;       /* length of an address, in bytes */
   char   **h_addr_list; /* null-terminated array of in_addr structs */
};
```

```
struct sockaddr_in  {
  . . .
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  . . .
};
```

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

```
struct hostent *hp;              /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */
...
bcopy((char *)hp->h_addr_list[0], /* src, dest */
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
```

Echo Client: open_clientfd (connect)

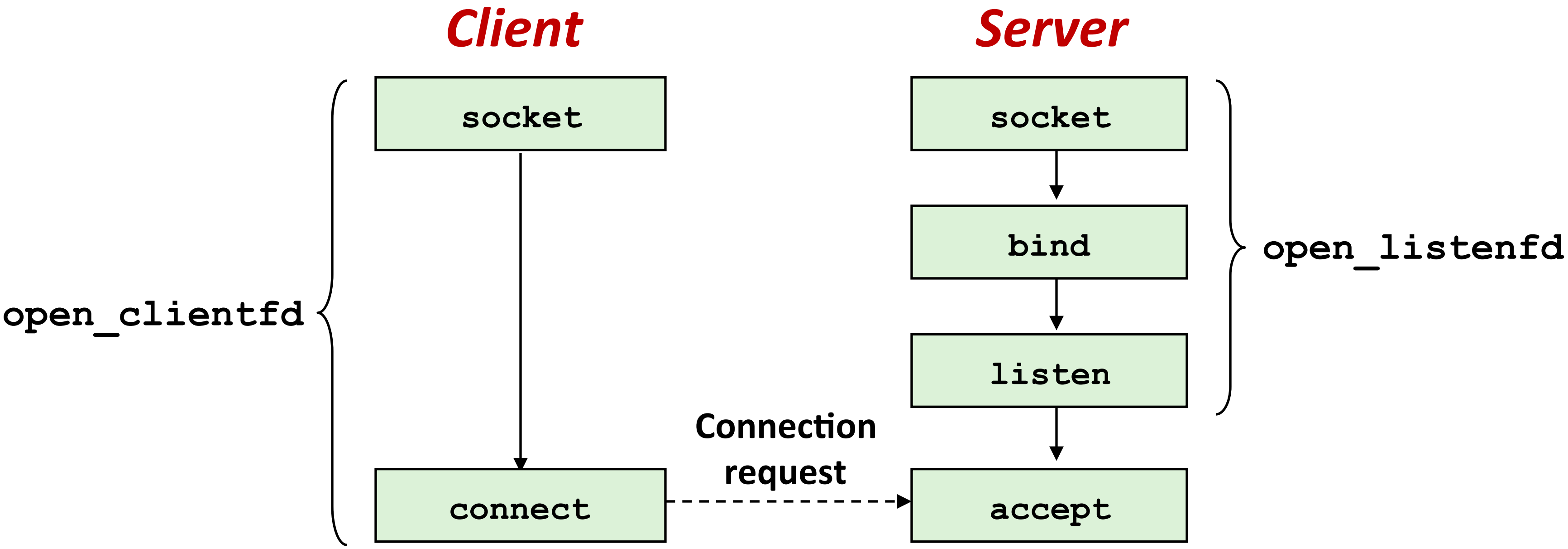
- **Finally the client creates a connection with the server**
  - Client process suspends (blocks) until the connection is created
  - After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls on descriptor **clientfd**

```
  int clientfd;                    /* socket descriptor */
  struct sockaddr_in serveraddr;   /* server address */
  typedef struct sockaddr SA;      /* generic sockaddr */
...
  /* Establish a connection with the server */
  if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
  return clientfd;
}
```

Echo Server: Main Routine

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;
    unsigned short client_port;

    port = atoi(argv[1]); /* the server listens on a port passed
                             on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                        sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        client_port = ntohs(clientaddr.sin_port);
        printf("server connected to %s (%s), port %u\n",
                hp->h_name, haddrp, client_port);
        echo(connfd);
        Close(connfd);
    }
}
```

Overview of the Sockets Interface



Echo Server: open_listenfd

```
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int)) < 0)
        return -1;

... <more>
```

Echo Server: open_listenfd (cont.)

```
...

  /* Listenfd will be an endpoint for all requests to port
       on any IP address for this host */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short)port);
    if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
        return -1;

    /* Make it a listening socket ready to accept
       connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;

   return listenfd;
}
```

Echo Server: open_listenfd (socket)

- **socket creates a socket descriptor on the server**
  - **AF_INET**: indicates that the socket is associated with Internet protocols
  - **SOCK_STREAM**: selects a reliable byte stream connection (TCP)

```
int listenfd; /* listening socket descriptor */

/* Create a socket descriptor */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
```

Echo Server: open_listenfd (setsockopt)

- **The socket can be given some attributes**

```
...
/* Eliminates "Address already in use" error from bind(). */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval , sizeof(int)) < 0)
    return -1;
```

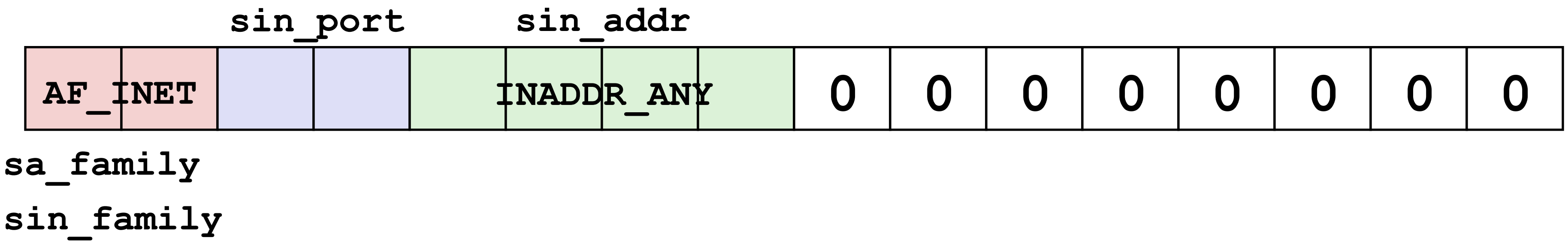
- **Handy trick that allows us to rerun the server immediately after we kill it**
  - Otherwise we would have to wait about 15 seconds
  - Eliminates "Address already in use" error from `bind()`
- **Strongly suggest you do this for all your servers to simplify debugging**

Echo Server: open_listenfd
(initialize socket address)

- Initialize socket with server port number
- Accept connection from any IP address

```
  struct sockaddr_in serveraddr; /* server's socket addr */
...
  /* listenfd will be an endpoint for all requests to port
     on any IP address for this host */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_port = htons((unsigned short)port);
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- IP addr and port stored in network (big-endian) byte order



Echo Server: open_listenfd (bind)

- bind associates the socket with the socket address we just created

```
int listenfd;                  /* listening socket */
struct sockaddr_in serveraddr; /* server's socket addr */

...
  /* listenfd will be an endpoint for all requests to port
     on any IP address for this host */
  if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
      return -1;
```

Echo Server: open_listenfd (listen)

- listen indicates that this socket will accept connection (connect) requests from clients
- LISTENQ is constant indicating how many pending requests allowed

```
int listenfd; /* listening socket */

...
 /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;
    return listenfd;
}
```

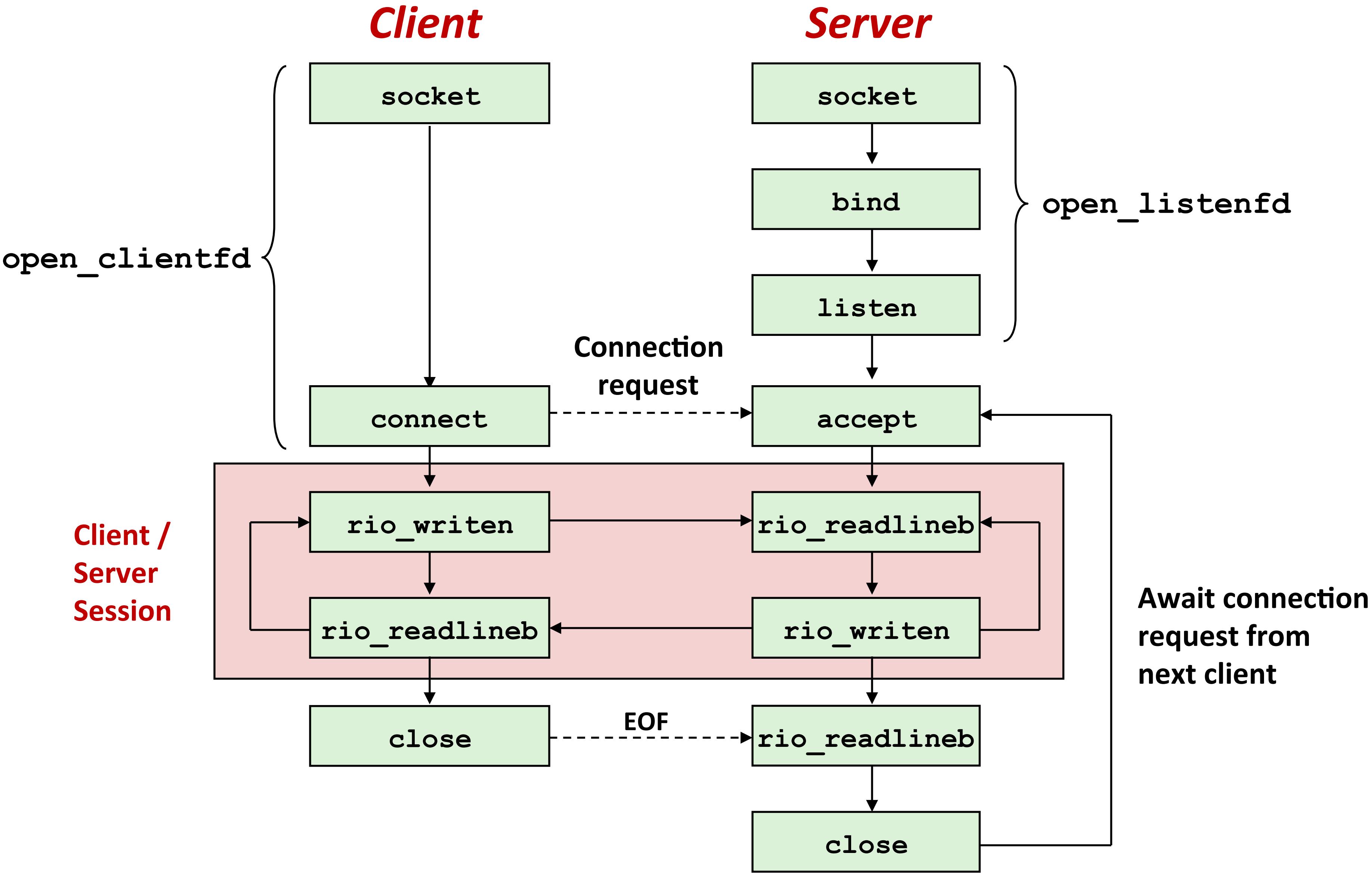
- We're finally ready to enter the main server loop that accepts and processes client connection requests.

Echo Server: Main Loop

- The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```
main() {

   /* create and configure the listening socket */

   while(1) {
      /* Accept(): wait for a connection request */
      /* echo(): read and echo input lines from client til EOF */
      /* Close(): close the connection */
   }
}
```

Overview of the Sockets Interface



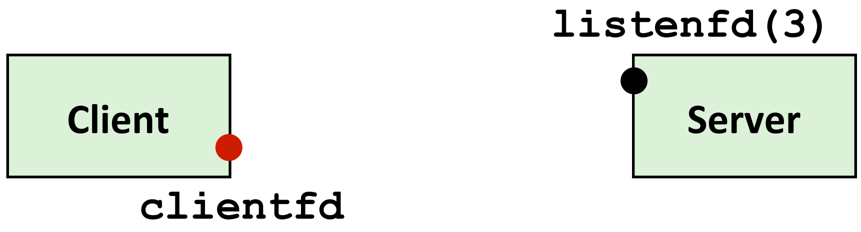
Echo Server: `accept`

- `accept()` blocks waiting for a connection request

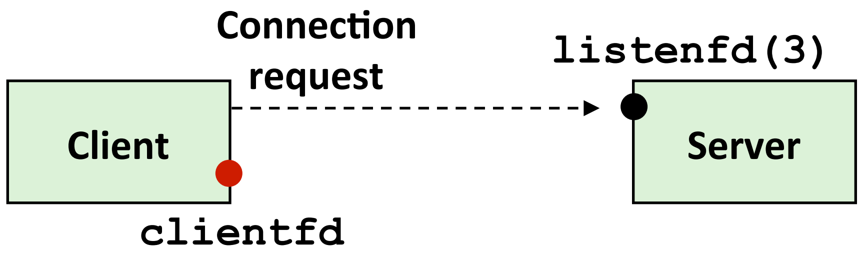
```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

- `accept` returns a *connected descriptor* (`connfd`) with the same properties as the *listening descriptor* (`listenfd`)
  - Returns when the connection between client and server is created and ready for I/O transfers
  - All I/O with the client will be done via the connected socket
- `accept` also fills in client's IP address

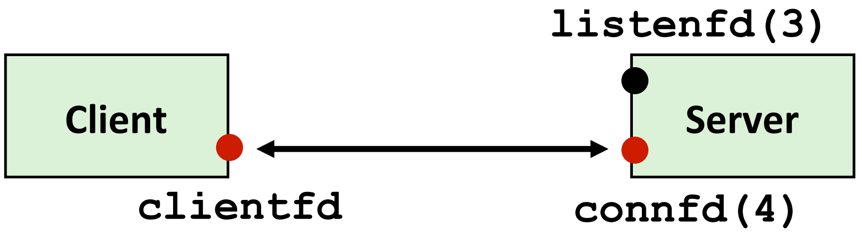
Echo Server: `accept` Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2. Client makes connection request by calling and blocking in `connect`



3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

Echo Server: Identifying the Client

- The server can determine the domain name, IP address, and port of the client

```
struct hostent *hp;   /* pointer to DNS host entry */
char *haddrp;         /* pointer to dotted decimal string */
unsigned short client_port;
hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                        sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
client_port = ntohs(clientaddr.sin_port);
printf("server connected to %s (%s), port %u\n",
       hp->h_name, haddrp, client_port);
```

Echo Server: `echo`

- **The server uses RIO to read and echo text lines until EOF (end-of-file) is encountered.**
  - EOF notification caused by client calling **`close(clientfd)`**

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        upper_case(buf);
        Rio_writen(connfd, buf, n);
        printf("server received %d bytes\n", n);
    }
}
```

Testing Servers Using `telnet`

- **The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections**
  - Our simple echo server
  - Web servers
  - Mail servers

- **Usage:**
  - **`unix>`** ***`telnet <host> <portnumber>`***
  - Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

Testing the Echo Server With `telnet`

```
greatwhite> echoserver 15213

linux> telnet greatwhite.ics.cs.cmu.edu 15213
Trying 128.2.220.10...
Connected to greatwhite.ics.cs.cmu.edu.
Escape character is '^]'.
hi there
HI THERE
```

For More Information

- **W. Richard Stevens, “Unix Network Programming: Networking APIs: Sockets and XTI”, Volume 1, Second Edition, Prentice Hall, 1998**
  - THE network programming bible
- **Unix Man Pages**
  - Good for detailed information about specific functions
- **Complete versions of the echo client and server are developed in the text**
  - Updated versions linked to course website
  - Feel free to use this code in your assignments