Session NM056

Programming TCP/IP with Sockets

Geoff Bryant
Process software

Course Roadmap

- NM055 (11:00-12:00) Important Terms and Concepts
  - TCP/IP and Client/Server Model
  - Sockets and TLI
  - Client/Server in TCP/IP
- NM056 (1:00-2:00) Socket Routines
- NM057 (2:00-3:00) Library Routines
- NM058 (3:00-4:00) Sample Client/Server
- NM059 (4:00-5:00) VMS specifics (QIOs)
- NM067 (6:00-7:00) Clinic - Q&A

TCP/IP Programming

Slides and Source Code available via anonymous FTP:

Host:: ftp.process.com
Directory: [pub.decus]
Slides: DECUS_F96_PROG.PS
Examples: DECUS_F96_PROG_EXAMPLES.TXT

Host: ftp.opus1.com
Slides: DECUS_F96_PROG.PS
Examples: DECUS_F96_PROG_EXAMPLES.TXT

Programming with Sockets Roadmap

- Berkeley Socket History
- Overview of the Socket Paradigm
- Socket addresses
- Other programming models

Berkeley Socket History

- Documented in such books as
  - Stevens's *Unix Network Programming*
  - Comer's *Internetworking with TCP/IP, vol III*
- First provided with Berkeley Software Distribution (BSD 4.1c) Unix for the VAX
- Popularized in the 1986 4.3BSD release

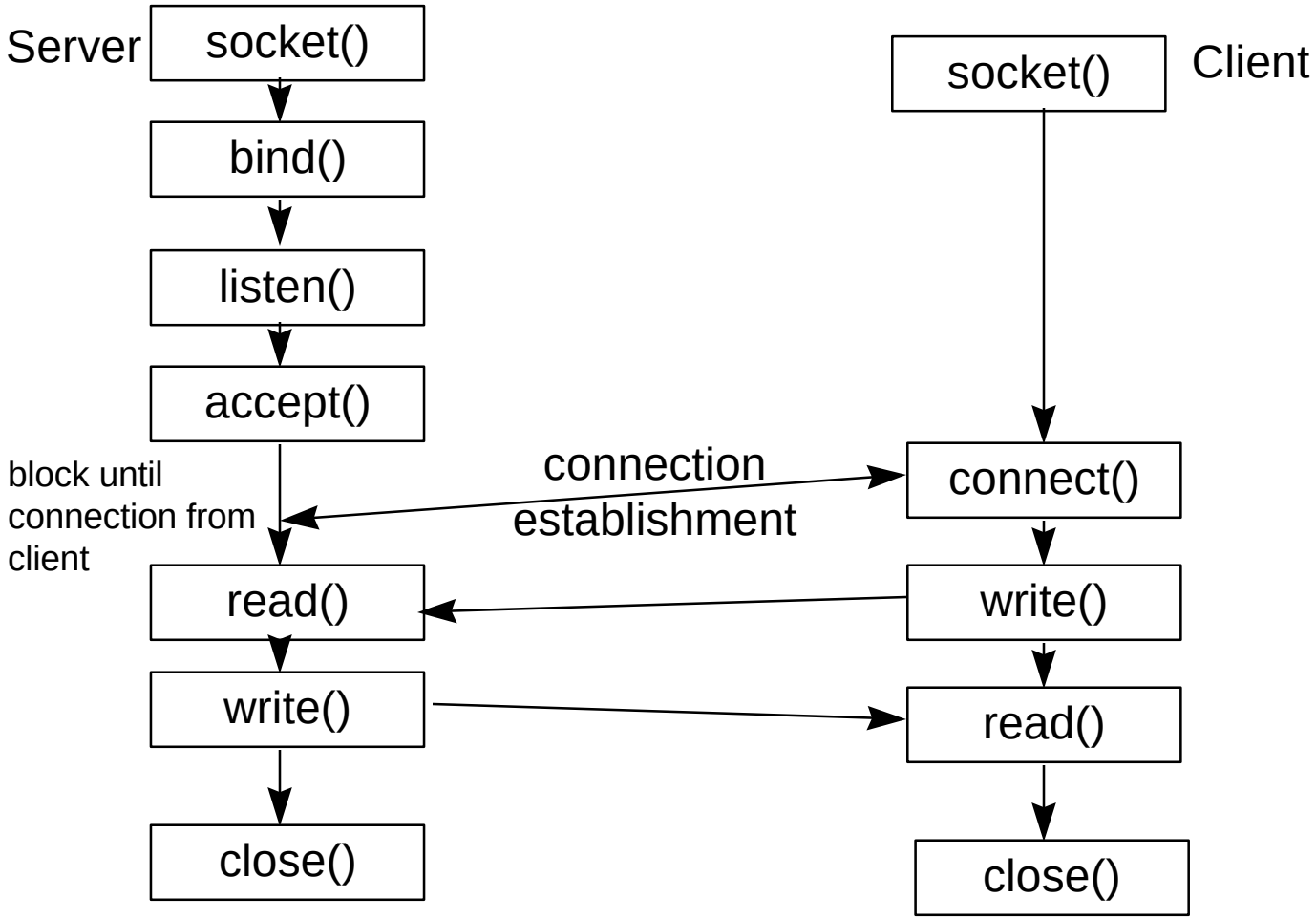
Other Programming Models

- Socket model is widely used
  - mainly due to wide implementation of BSD networking kernel
  - Avaible for Unix, Windows, VMS, ...
- Other models may be used in different environments
  - STREAMS model (UNIX System V)
  - Others

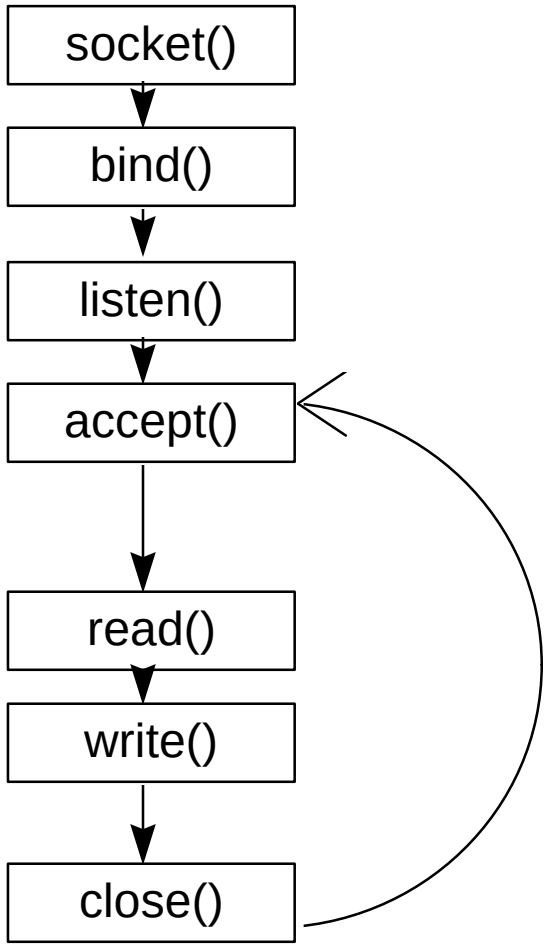
Socket Paradigm Overview

- A *socket* is a communications endpoint
- In BSD networking, it is a data structure within the kernel
- A socket is “named” by its *socket address*
- A *connection* is represented by two communicating sockets

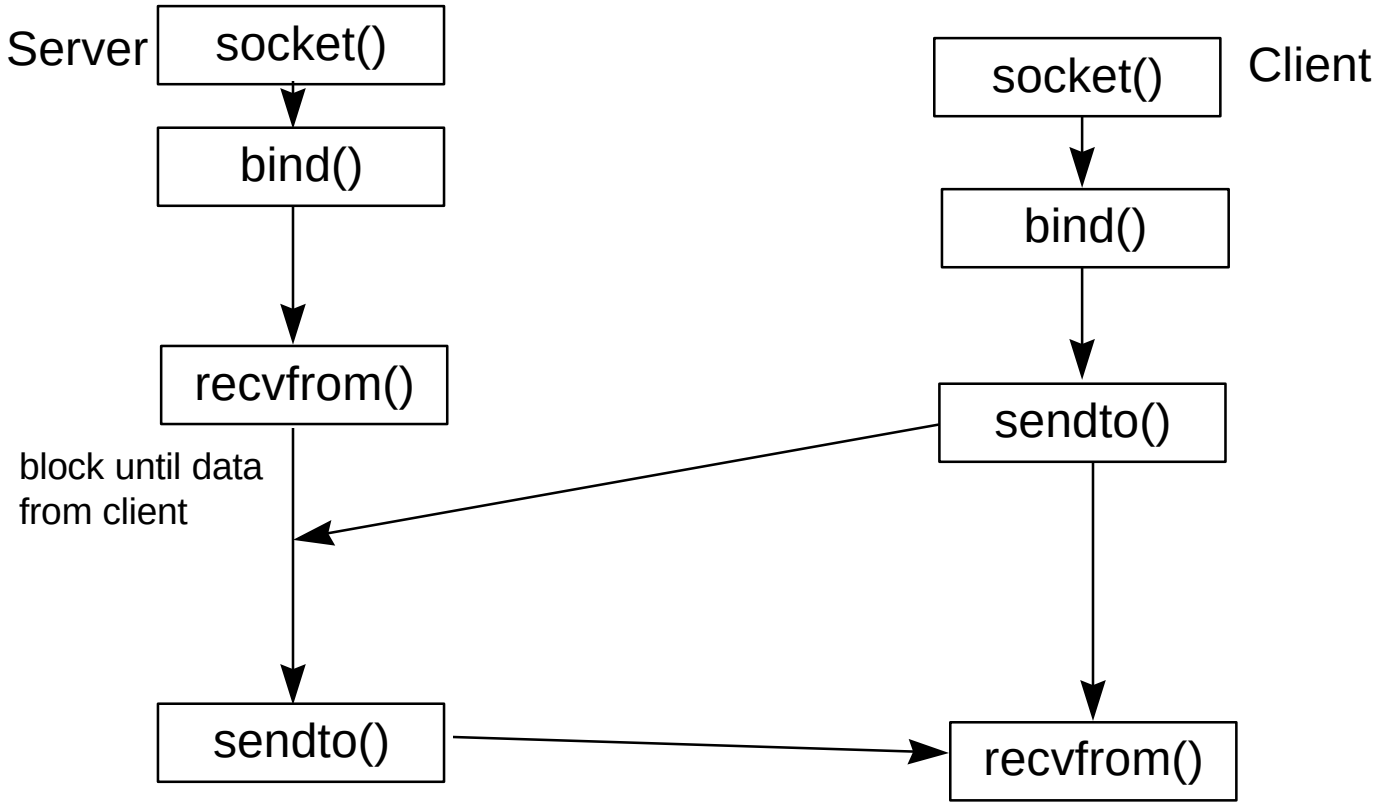
Using sockets is like using the file system



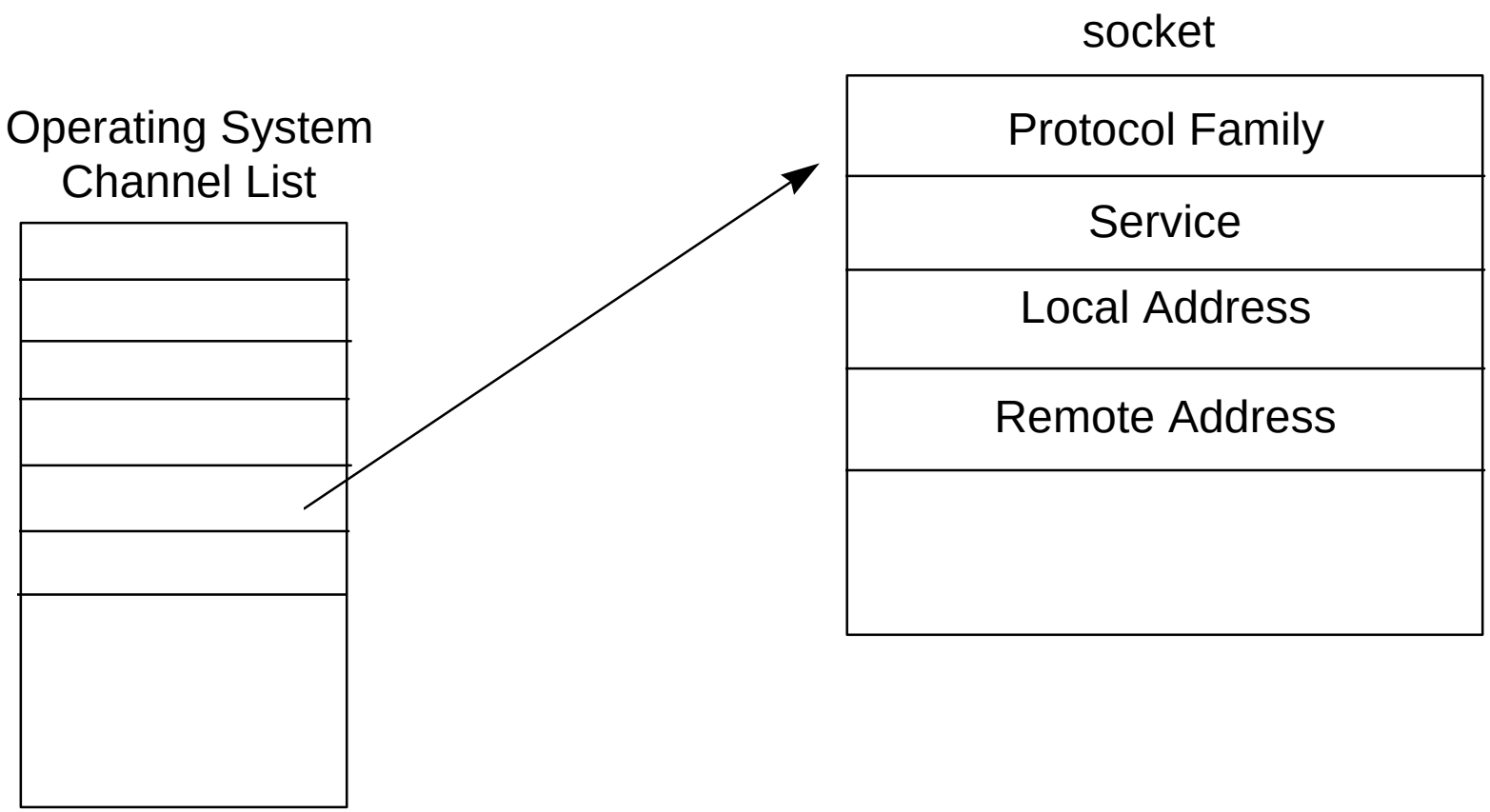
Servers sit in a tight loop



Connectionless Client/Server is simpler



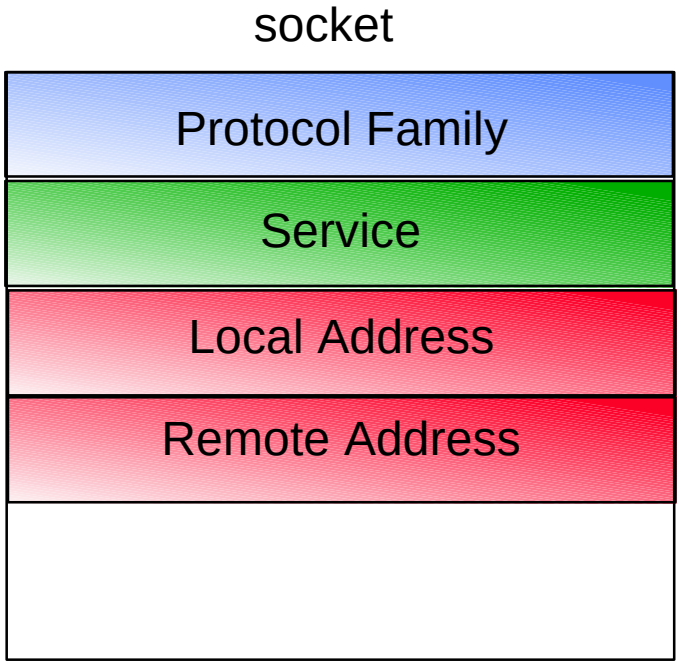
A socket is just a data structure



In C, a socket doesn't have much

integer: always set to be PF_INET for TCP/IP programming

sockaddr: a 2-octet address family identifier and then 14 octets of address-specific information

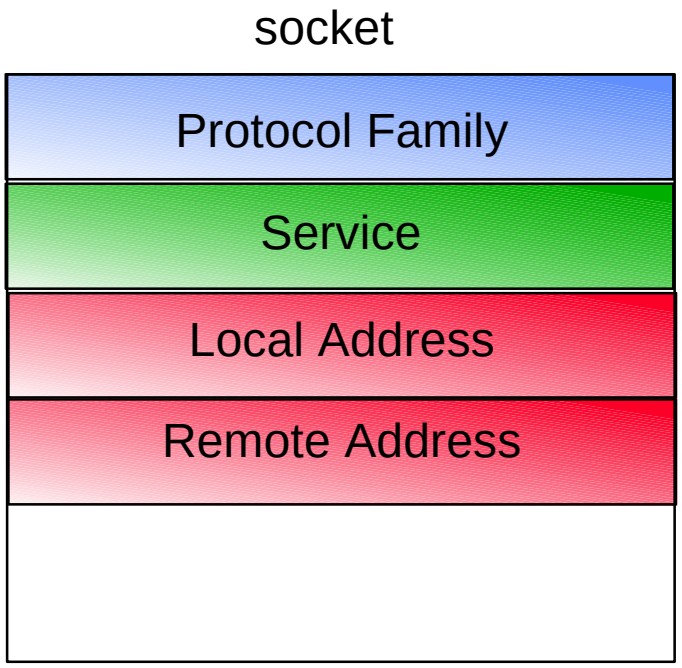


integer: set to be SOCK_STREAM (TCP), SOCK_DGRAM (UDP), or SOCK_RAW (raw IP)

Families and Types

- Address Families (or Protocol Families)
  - AF_UNIX: Unix domain sockets
  - AF_INET: Internet Protocols
  - AF_NS: Xerox NS Protocols
  - AF_IMPLINK: IMP link layer (obsolete)
- Types:
  - SOCK_STREAM: Stream socket (TCP)
  - SOCK_DGRAM: Datagram socket (UDP)
  - SOCK_RAW: Raw socket (IP)

TCP/IP addresses are special cases of sockaddr




```
struct sockaddr {
    u_short sa_family;
    char sa_data[16];
}
```

```
struct sockaddr_in {
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
}
```

sockaddr: a 2-octet address family identifier and then 14 octets of address-specific information

sockaddr unveiled

```
struct sockaddr_in {
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
}
```

sin_family: AF_INET for all TCP/IP addresses

sin_port: 16-bit port number (as used in UDP & TCP headers)

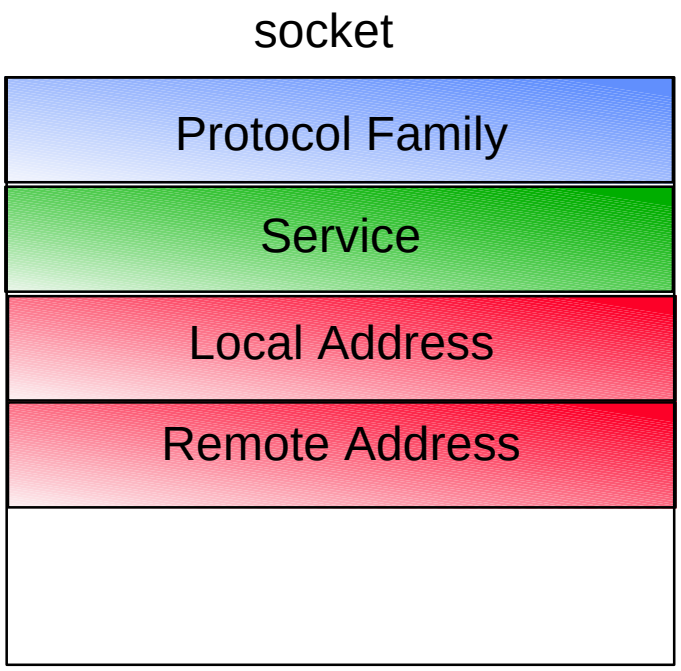
sin_addr: 32-bit IP address

```
/*
 * Internet address (a structure for historical reasons)
 */
struct in_addr {
    union {
       struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
       struct { u_short s_w1,s_w2; } S_un_w;
       u_long S_addr;
    } S_un;
#define s_addr  S_un.S_addr
```

Remember our goal: open()

- TCP/IP socket programming is mostly `read()` and `write()` subroutine calls
- All of the socket routines are there to do the equivalent of an `open()` in the file system.

Five routines are used to replace the open() call



`socket()` : allocate socket in memory, fill in protocol family and service fields

`bind()` : fill in the local address part (local sockaddr). OPTIONAL for most clients

`listen()` : tell network kernel that you want to receive connects (“passive open”)

`accept()` : ask network kernel to hand you the next incoming connect (“passive open”)

`connect()` : tell network kernel to connect to the other side (“active open”)

Overview of Sockets (one more time)

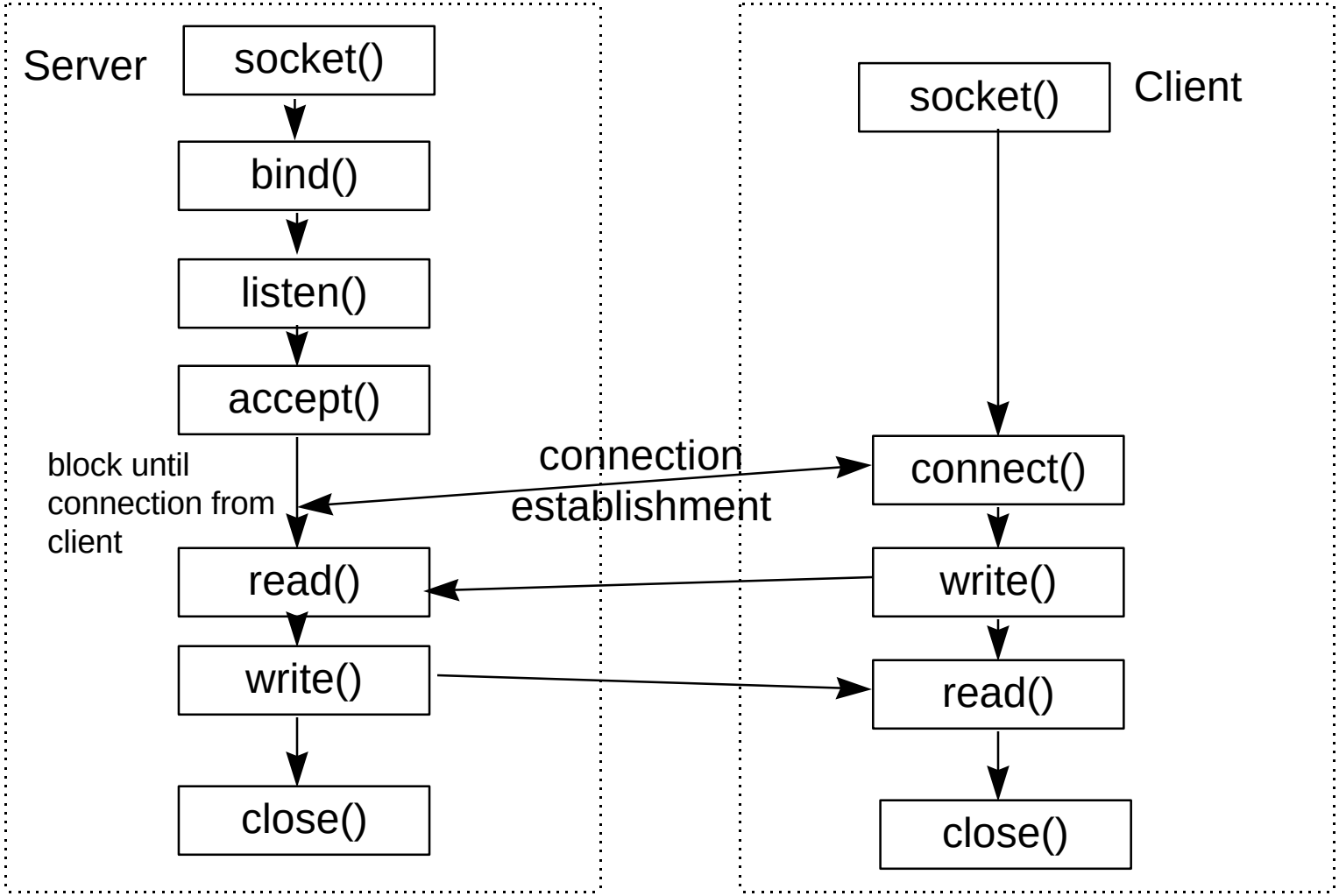
- Get the socket open somehow
  - socket(), bind(), listen(), accept(), connect()
- Read and write from it
  - read(), write()
- When done, close the socket
  - close()

Programming with Sockets Key Concepts

- A socket is a data structure representing a connection
- To open a connection
  - Fill in the data structure
  - Link it to the network kernel
  - Link it to the operating system
- Reading and writing uses the same calls as the file system

Coding with Sockets

General Flow for Servers/Clients is different



Let's start with clients because they're easier

Clients

Step 1: Get Socket

socket()

connect()

write()

read()

close()

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

domain = AF_INET;     /* always the same */
type = SOCK_STREAM;   /* STREAM for TCP */
protocol = 0;
s = socket(domain, type, protocol);
if (s < 0) ERROR("Cannot create socket");
```

Step 2: Fill in remote address fields

socket()

connect()

write()

read()

close()

```
#define SMTP_PORT 25
#define IP_ADDRESS 0xC0F50C02 /*192.245.12.2*/
struct sockaddr_in sin;

sin.sin_family = AF_INET;
sin.sin_port = htons(SMTP_PORT);
sin.sa_addr = htonl(IP_ADDRESS);
```

Step 3: Connect to other side

socket()

connect()

write()

read()

close()

```
int connect(int s, struct sockaddr*name, int
namelen);

int status;


status = connect(s, (struct sockaddr*)sin,
sizeof(sin));

if (status != 0)

    ERROR("Cannot connect to other side");
```

Step 4: Use the socket

socket()

connect()

write()

read()

close()

```
char buf[LINELEN+1];

/*
 *  Now go into a loop, reading data from the network, writing
 *  it to the terminal and reading data from the terminal,
 *  writing it to the network...
 */

while ( (n = read(s, buf, LINELEN) ) > 0) {
    fwrite(buf, n, 1, stdout);

    if (!fgets(buf, LINELEN, stdin)) break;
    write(s, buf, strlen(buf));
 }
 if (n < 0) {
     ERROR("Cannot read from socket!");
}
```

Step 5: Shut it down when done

socket()

connect()

write()

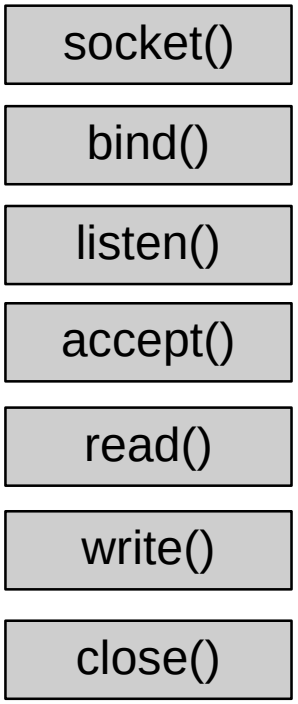
read()

close()

```
status = close(s);
if (status != 0)
    ERROR(“Can not close socket”);
```

Servers

Server Review and Overview



`socket()` : allocate socket in memory, fill in protocol family and service fields

`bind()` : fill in the local address part (local sockaddr). OPTIONAL for most clients

`listen()` : tell network kernel that you want to receive connects (“passive open”)

`accept()` : ask network kernel to hand you the next incoming connect (“passive open”)

Step 1: Get Socket

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

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

domain = PF_INET;     /* always the same */
type = SOCK_STREAM;   /* STREAM for TCP */
protocol = 0;
s = socket(domain, type, protocol);
if (s < 0) ERROR("Cannot create socket");
```

This should look vaguely familiar...

Step 2: Fill in local address fields

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

```
#define SMTP_PORT 25
#define IP_ADDRESS 0xC0F50C02 /*192.245.12.2*/
struct sockaddr_in sin;

sin.sin_family = AF_INET;
sin.sin_port = htons(SMTP_PORT);
sin.sa_addr = INADDR_ANY;
```

INADDR_ANY is a shorthand way of saying “I don’t care what the local address is”...

Step 3: “Bind” address to the socket

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

```
int bind(int s, struct sockaddr*name, int
namelen);

int status;


status = bind(s, (struct sockaddr*)sin,
sizeof(sin));

if (status != 0)

    ERROR(“Cannot bind to local address”);
```

INADDR_ANY is a shorthand way of saying “I don’t care what the local address is”...

Step 4: Tell the kernel to listen

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

```
#define MAX_BACKLOG 5
int listen(int s, int backlog);
int status;

status = listen(s, MAX_BACKLOG);
if (status != 0)
    ERROR("Cannot ask kernel to listen");
```

Step 5: Block waiting for connect

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

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

int status, addrlen, vs;

struct sockaddr_in sin2;


addrlen = sizeof(sin2);

vs = accept(s, (struct sockaddr*)&sin2,
&addrlen);

if (vs < 0)

    ERROR("Cannot accept a connection.");
```

Step 6: Read and Write

socket()

bind()

listen()

accept()

read()

write()

close()

```
/* Get the line from the client. */
        read ( vs, buf, 256 );

/* Translate the logical name. */
        log_name = getenv ( buf );

/* Get the definition string and add a new line to it. */
        sprintf ( buf, "%s\n", log_name );

/* Write the translation to the client. */
        write ( vs, buf, strlen ( buf ) );
```

Yes, this code has lots of holes in it. But you get the picture of what we're trying to do.

Step 7: Tear down when done

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

```
status = close(s);
if (status != 0)
    ERROR(“Can not close socket”);
```

Of course, there are more routines than those

socket	Create a descriptor for use in network communication
connect	Connect to a remote peer (client)
write	Send outgoing data across a connection
read	Acquire incoming data from a connection
close	Terminate communication and deallocate a descriptor
bind	Bind a local IP address and protocol port to a socket
listen	Place the socket in passive mode and set backlog
accept	Accept the next incoming connection (server)
recv, recvmsg	Receive the next incoming datagram
recvfrom	Receive the next incoming datagram and record source addr.
send, sendmsg	Send an outgoing datagram
sendto	Send an outgoing datagram to a particular dest. addr.
shutdown	Terminate a TCP connection in one or both directions
getpeername	After a connection arrives, obtain remote machine's address
getsockopt	Obtain the current options for a socket
setsockopt	Change the options for a socket

Coding with Sockets Key Concepts

- All clients and servers basically look the same
- Follow a template, but make sure you understand what you're doing
  - Lots of templates are wrong
  - Lots of templates don't do what you think they do