COMP20200 Unix Programming

Lecture 15

Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

School of Computer Science, University College Dublin, Ireland

29/03/2023

Lecture overview

- Overview of socket system calls;
- Develop an iterative TCP socket server.

Sockets overview

- Sockets allow data to be exchanged between applications either on the same host or different hosts connected by a network.
- Sockets is the leading API if you require communication between two applications executing on different hosts on a network.
- We use the term TCP socket to refer to an Internet domain stream socket;
- For TCP socket,
  - Socket domain is **AF_INET** or **AF_INET6**.
  - Socket type is **SOCK_STREAM**.
  - protocol argument is 0.
- TCP sockets offer reliability.

TCP sockets: system call sequence diagram

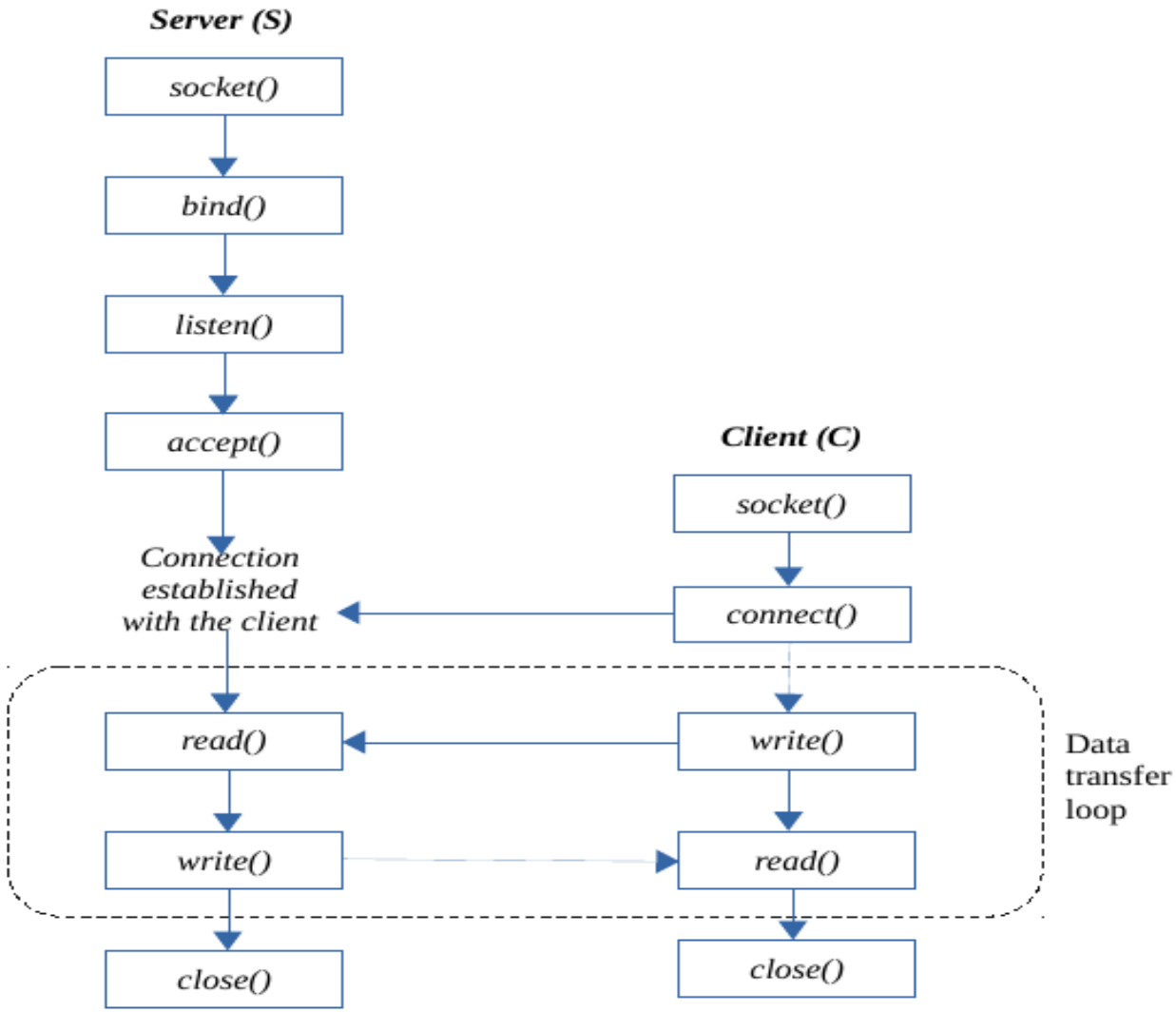

Figure: System calls with TCP sockets.

socket() call

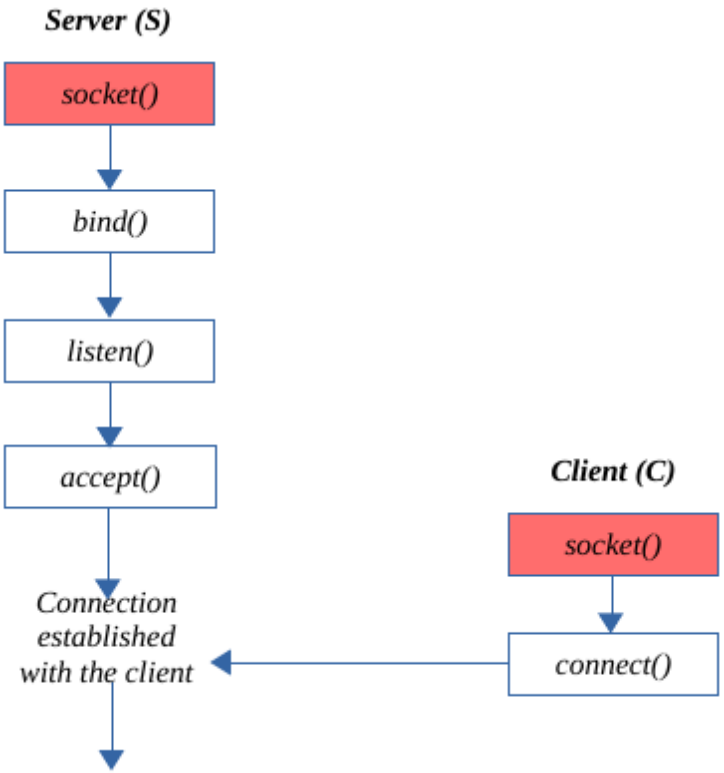

Figure: Both client and server create a new socket using socket() call.

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

- The *socket()* system call creates a new socket.
- On success, socket() returns a file descriptor used to refer to the newly created socket.

bind() call

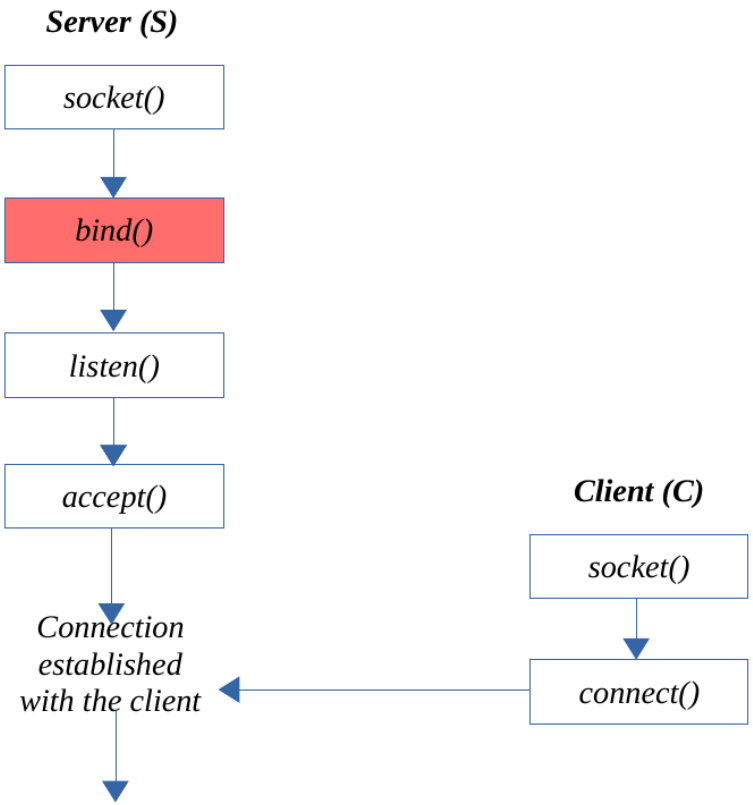

Figure: Socket bind() call.

```
#include <sys/socket.h>
int bind(int fd, const struct sockaddr *
    addr, socklen_t addrlen);
```

- The *bind()* system call binds a socket to an address.
- The **fd** argument is a file descriptor obtained from a previous call to socket().
- The **addr** argument is a pointer to a structure specifying the address to which this socket is to be bound.
- The **addrlen** argument specifies the size of the address structure.

bind() call: what is *struct sockaddr*?

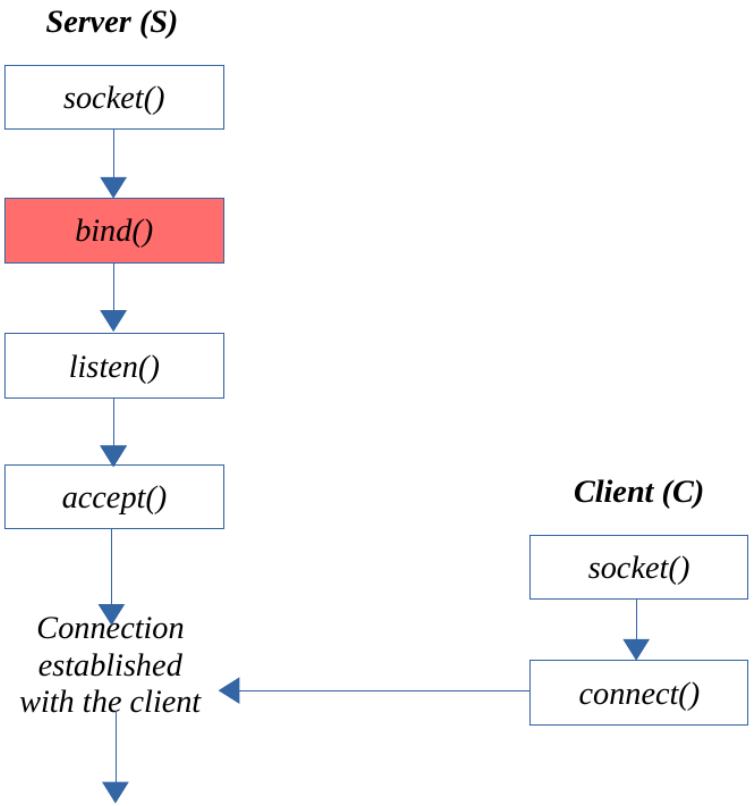

Figure: Socket bind() call.

```
struct sockaddr {
    /* Address family (AF_* constant) */
    sa_family_t sa_family;
    /* Socket address (size varies
       according to socket domain) */
    char sa_data[14];
};
```

- A server's socket is bound to a well-known address that is known in advance to client applications that need to communicate with that server.

bind() call: what is *struct sockaddr*?

- Each socket domain uses a different address format.
- For each socket domain, a different structure type is defined to store a socket address.

Table: sockaddr structures.

Domain	**Address structure**
AF_UNIX	sockaddr_un
AF_INET	sockaddr_in
AF_INET6	sockaddr_in6

bind() call: what is *struct sockaddr*?

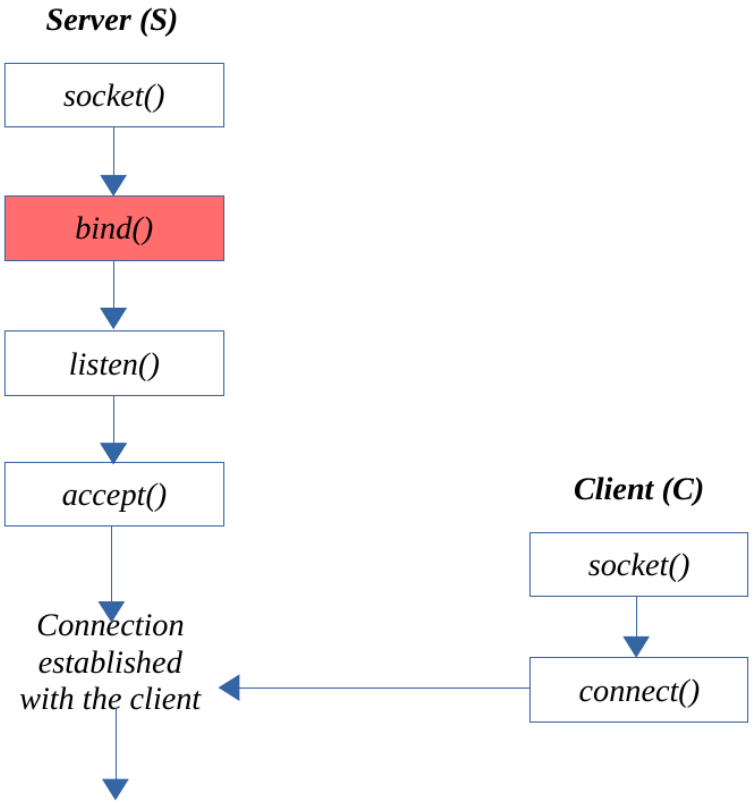

Figure: Socket bind() call.

```
struct sockaddr {
    /* Address family (AF_* constant) */
    sa_family_t sa_family;
    /* Socket address (size varies
       according to socket domain) */
    char sa_data[14];
};
```

- How to design one bind() interface function that caters to all the socket domains?
- bind() must be able to accept address structures of any type.
- To permit this, sockets API defines a generic address structure, **struct sockaddr**.

listen() call

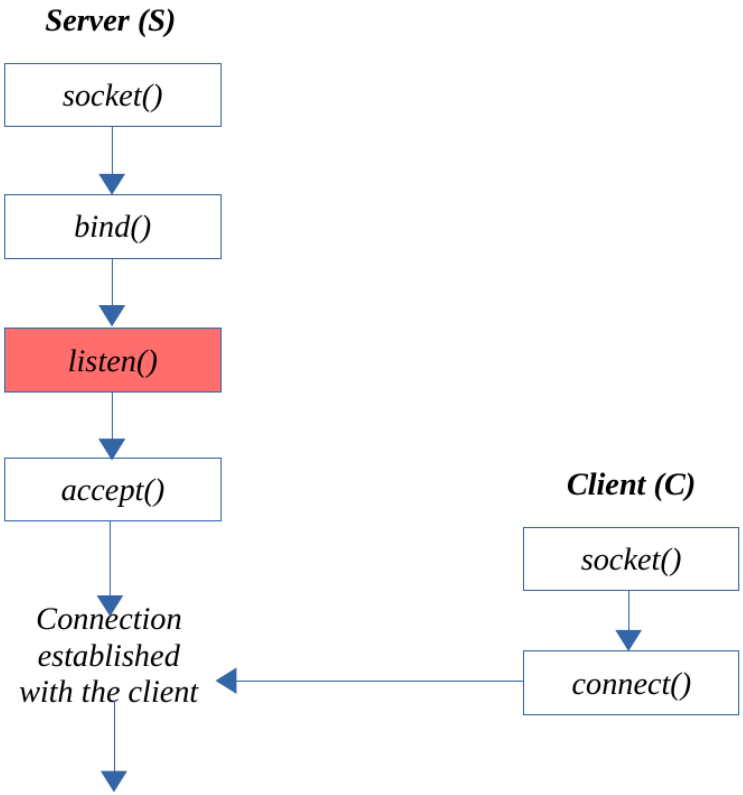

Figure: Socket listen() call.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- The *listen()* system call allows a stream socket to accept incoming connections from other sockets.
- What is the backlog argument?
  - If a client calls *connect()* before the server calls *accept()*, the connection request becomes a pending connection.
  - The backlog argument limits the number of pending connections.
  - The kernel maintains a queue of pending connections.

accept() call

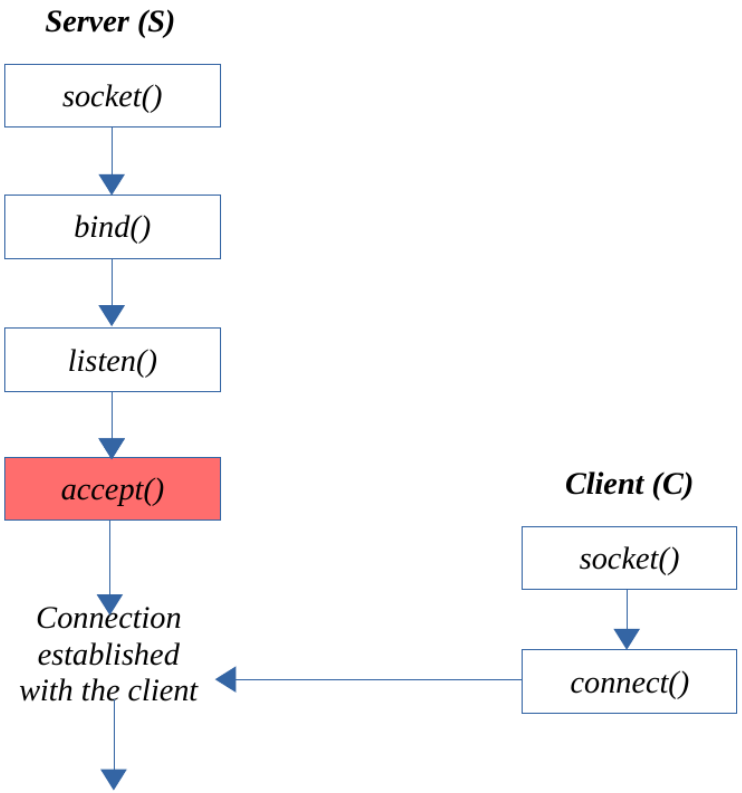

Figure: Socket accept() call.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *
    addr, socklen_t *addrlen);
```

- The *accept()* system call accepts an incoming connection on a listening stream socket.
- *The accept() call creates a new socket.*
- The new socket is connected to the peer socket that performed the connect().
- The file descriptor for the connected socket is the result of the accept() call.

accept() call

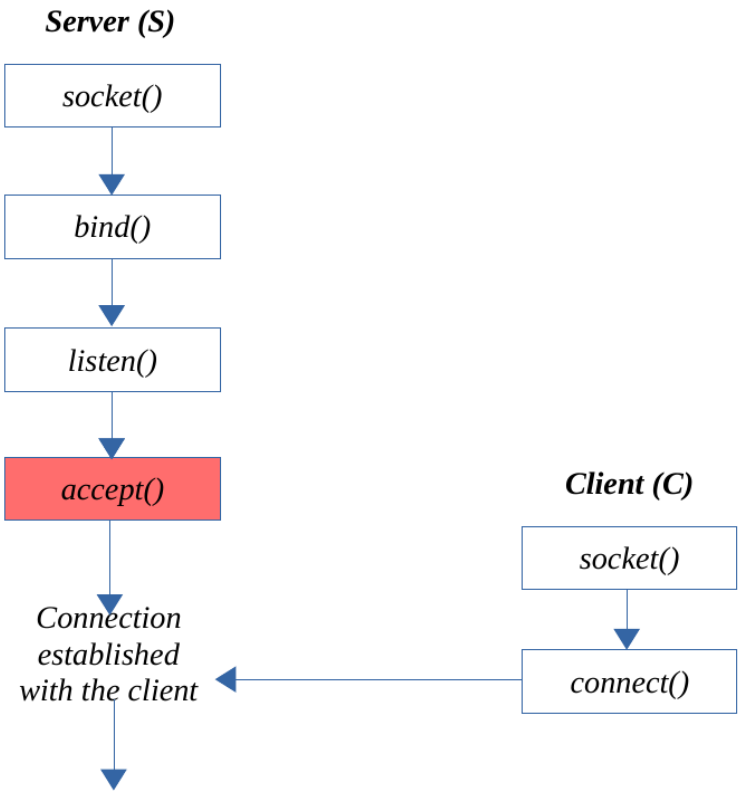

Figure: Socket accept() call.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *
    addr, socklen_t *addrlen);
```

- The listening socket (*sockfd*) remains open, and can be used to accept further connections.
- The *addr* and *addrlen* arguments return the address of the peer socket.
- If we are not interested in the address of the peer socket, then addr and addrlen should be specified as NULL and 0.

connect() call

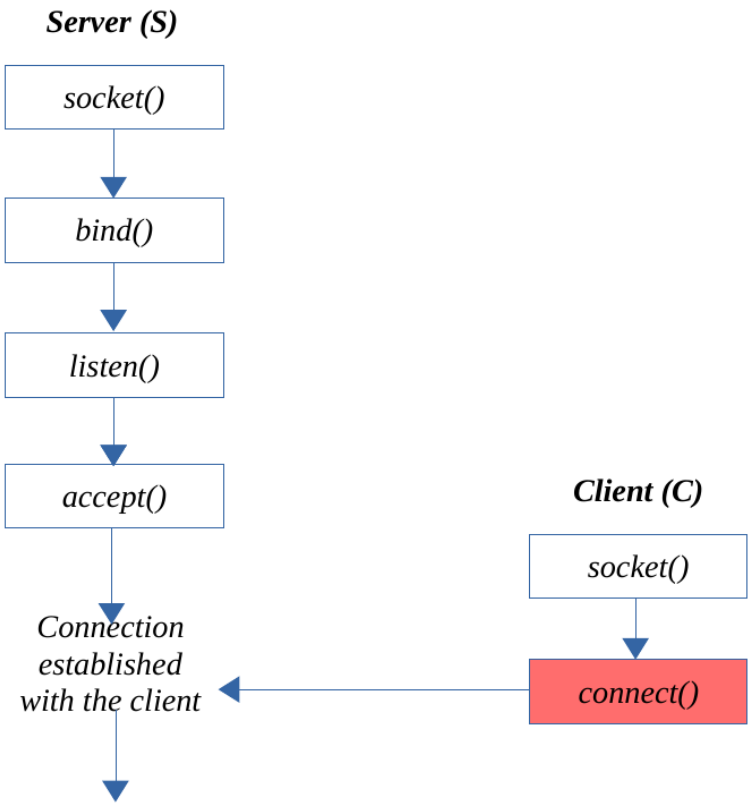

Figure: Socket connect() call.

```
#include <sys/socket.h>
int connect(int sockfd, const struct
    sockaddr *addr, socklen_t addrlen);
```

- The *connect()* system call connects the active socket referred to by *sockfd* to the listening socket.
- The address of the listening socket is specified by *addr* and *addrlen* arguments.

TCP sockets: connection establishment

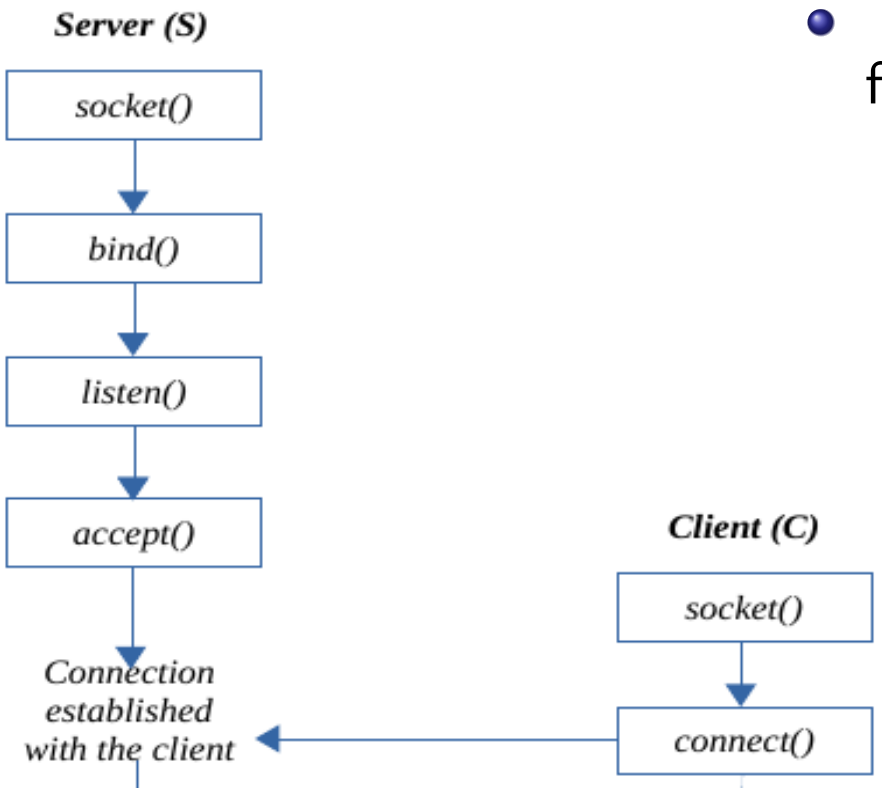

Figure: Connection establishment between client and server.

- **S** and **C** issue a *socket()* call.
- The two sockets are connected as follows:
  - **S** calls *bind()* to bind the socket to a well-known address.
  - **S** calls *listen()* to notify the kernel that it is ready to accept incoming connections.
  - **C** calls *connect()* to establish the connection by specifying the address.
  - **S** that called *listen()* then accepts the connection using *accept()*.

I/O on stream sockets

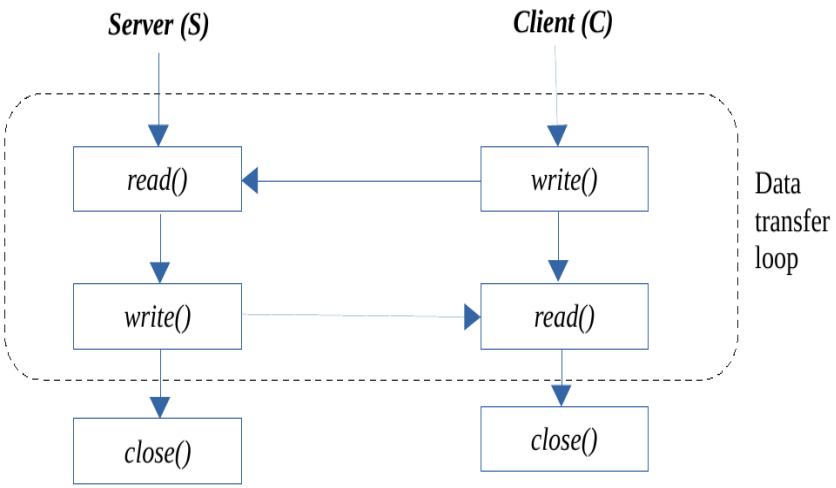

Figure: Sending data and receiving data.

- **S** and **C** can now transfer data between themselves using *read()* and *write()* system calls or the socket-specific *send()/recv()*.
- Until **S** or **C** close the connection using *close()* call.

Developing an iterative TCP Socket Server

Iterative TCP Socket Server

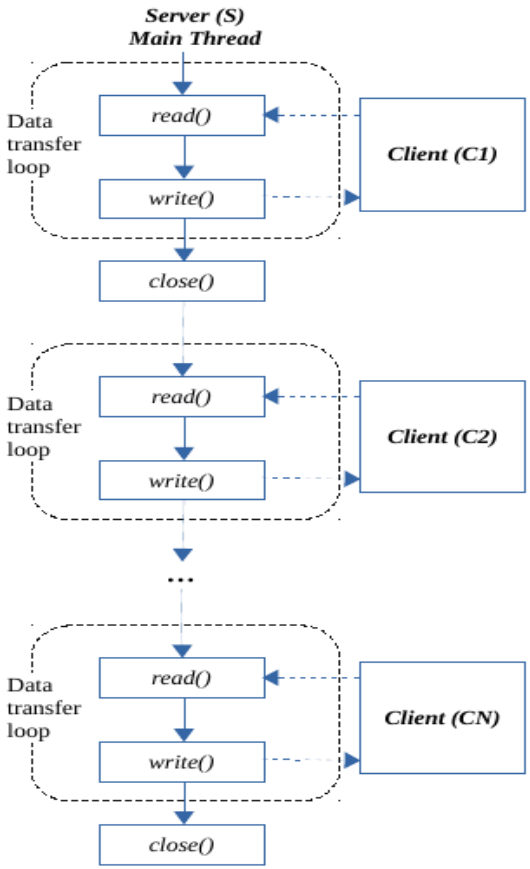

Figure: TCP iterative socket server.

- The server handles one client at a time.
- In a loop,
  - It issues *accept()* call to accept the connection.
  - Data exchange with client using *read()/write()* calls.
  - *close()* call to close the data socket.
  - And then service the next client.

Iterative TCP Socket Server - Headers

Our simple server echoes the message from a client.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

- Common headers and declarations for both the client and server.
- *stdio.h* include for file I/O.
- *string.h* include for *memset()* function.
- *unistd.h* include for socket I/O (*read()*/*write()*).

Iterative TCP Socket Server - Macros

```c
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <errno.h>
#define BUFSIZE 32              /* Message size */
#define SERVERIP "127.0.0.1" /* Loopback address */
```

- *socket.h* include for *socket()* system calls.
- *netdb.h* include for *getnameinfo()* function.
- *arpa/inet.h* include for *inet_addr()* function.
- *errno.h* include for errno.
- Our client and server exchange messages of size 32 bytes.
- “127.0.0.1” is a loopback IP address convenient for testing your client and server codes.

Server code - socket creation

```
int lfd = socket(AF_INET, SOCK_STREAM, 0);
if (lfd == -1) {
    fprintf(stderr, "socket() error.\n");
    exit(-1);
}
```

- On success, *lfd* contains the newly created socket.

Server code - bind and publish endpoint

```
struct sockaddr_in serverAddress;
memset(&serverAddress, 0, sizeof(struct sockaddr_in));
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = inet_addr(SERVERIP);
serverAddress.sin_port = htons(atoi(argv[1]));
int rc = bind(lfd, (struct sockaddr *)&serverAddress,
                    sizeof(struct sockaddr));
if (rc == -1) {
    fprintf(stderr, "bind() error.\n");
    exit(-1);
}
```

- Since we are creating IPv4 socket address, we fill **struct sockaddr_in**.
- The server's listening endpoint has the address, (SERVERIP, port).
- *port* is supplied to the server program as the first argument (argv[1]).

inet_addr() and htons() library functions

- inet_addr() function converts a IPv4 dotted string into binary data in network byte order.
- htons() function takes a numeric port value and converts it to network byte order.
- IP addresses and port numbers are integer values stored in a host in *host byte order*.
- Since they are transmitted between a network of hosts, they must be translated to binary *network byte order*.
- Read up on **little endianness** and **big endianness**.

Server code: Listening for connections

```
#define BACKLOG 10
if (listen(lfd, BACKLOG) == -1)
   exit(EXIT_FAILURE);
/* Cont'd... */
```

- It marks the socket *lfd* as *passive*.
- The socket *lfd* will then be able to accept connections from other (active) sockets.

Server code: Handling clients

```
for (;;) { /* Handle clients iteratively */
    struct sockaddr_storage claddr;
    socklen_t addrlen = sizeof(struct sockaddr_storage);
    int cfd = accept(lfd, (struct sockaddr *)&claddr, &addrlen);
    if (cfd == -1) {
        continue;    /* Print an error message */
    }
/* Cont'd... */
```

- The *for loop* services clients iteratively.
- A new connection is accepted using *accept()* call.
- *accept()* call creates a new socket.
- It is this socket that is connected to the client socket that performed the *connect()*.
- The listening socket *lfd* remains open to accepting new connections.
- The address of the client is obtained in *claddr*.

Server code: Getting the client address

```
#include <sys/socket.h>
#include <netdb.h>
char host[NI_MAXHOST];
char service[NI_MAXSERV];
if (getnameinfo((struct sockaddr *) &claddr, addrlen,
            host, NI_MAXHOST, service, NI_MAXSERV, 0) == 0)
    fprintf(stdout, "Connection from (%s, %s)\n", host, service);
else
    fprintf(stderr, "Connection from (?UNKNOWN?)");
  /* Cont'd... */
```

- *getnameinfo()* is a library function.
- Given a socket address structure (either IPv4 or IPv6), it returns strings containing the corresponding host and service name (or numeric equivalent).
- The server displays the client's address (IP address plus port number) on standard output.

Server code: Receive the client message

```
size_t totRead;
char* bufr = buf;
for (totRead = 0; totRead < BUFSIZE; ) {
    ssize_t numRead = read(cfd, bufr, BUFSIZE - totRead);
    if (numRead == 0)
        break;
    if (numRead == -1) {
        if (errno == EINTR)
            continue;
        else {
            fprintf(stderr, "Read error.\n");
        }
    }
    totRead += numRead;
    bufr += numRead;
}
```

- The loop ensures read of BUFSIZE bytes.
- A *read()* may read fewer bytes than requested.
- Such partial transfers can occur when performing I/O on stream sockets.

Server code: echo the message

```
size_t totWritten;
const char* bufw = buf;
for (totWritten = 0; totWritten < BUFSIZE; ) {
    ssize_t numWritten = write(cfd, bufw, BUFSIZE - totWritten);
    if (numWritten <= 0) {
        if (numWritten == -1 && errno == EINTR)
            continue;
        else {
            fprintf(stderr, "Write error.\n");
            exit(EXIT_FAILURE);
        }
    }
    totWritten += numWritten;
    bufw += numWritten;
}
```

- The loop ensures write of BUFSIZE bytes.
- A *write()* may transfer fewer bytes than requested.
- The *write()* may be interrupted by a signal-handler.

Server code: Termination

```
            /* in the for loop... */
            if (close(cfd) == -1) /* Close connection */ {
                fprintf(stderr, "close error.\n");
                exit(EXIT_FAILURE);
            }
        }
        if (close(lfd) == -1) /* Close the listening socket fd */ {
            fprintf(stderr, "close error.\n");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

- Close the connection established with the client using the *close()* system call.
- Service the next client.
- Close the listening socket fd (*lfd*) before the *exit*.

TCP socket client and server sources

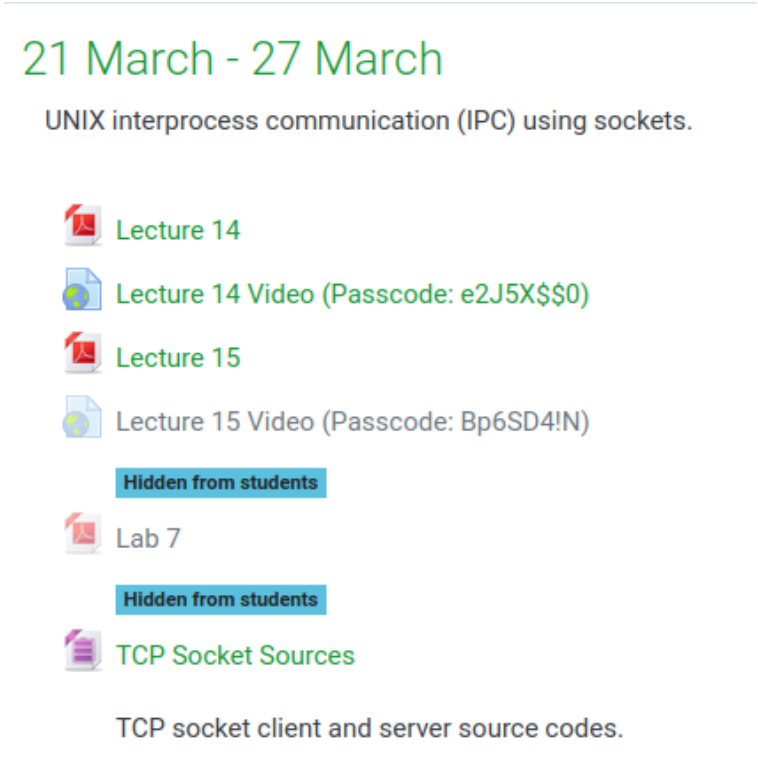

Figure: TCP socket client and server sources.

- iserver.c — TCP socker server that takes a port as an argument.
- iclient.c — A client that connects to a server at the IP address and port given to it as arguments.
- iserver_addrinfo.c — Uses *getaddrinfo()* function to automatically select an address to bind.
- iclient_addrinfo.c — Uses *getaddrinfo()* function to automatically select an address to connect.

Executing the TCP stream socket server

```
$ gcc -o iserver iserver.c
$ gcc -o iclient iclient.c
```

In one terminal,

```
$ ./iserver 49999
Listening on (127.0.0.1, 49999)
<waiting for clients to connect>
<ctrl-C to terminate>
```

In a different terminal,

```
$ ./iclient 127.0.0.1 49999
Sending ORHERDQOPUUUZNXBDRYAJYUFWFWMBJP to localhost:49999
Received ORHERDQOPUUUZNXBDRYAJYUFWFWMBJP
```

Press ctrl-C to send SIGINT to server for termination.

What is the IP address, 127.0.0.1?

- 127.0.0.1 is the loopback address. It is assigned the hostname, **localhost**.
- A datagram sent to this address is never placed on a network.
- But automatically loops back to become an input to the sending host.
- *This address is most used for testing client and server applications on the same host.*

Lookahead: Lecture 16

In the next lecture, we will develop a TCP socket client and a multithreaded server.

Q & A