

Project Part 2: Background Information

Overview

Read each section below before you start the Part 2 of your project. You will need this information to complete Part 2 of the project. Consult your Open Learning Faculty Member if you have any questions about the project.

Objectives

In completing Project Part 2, you will achieve the following:

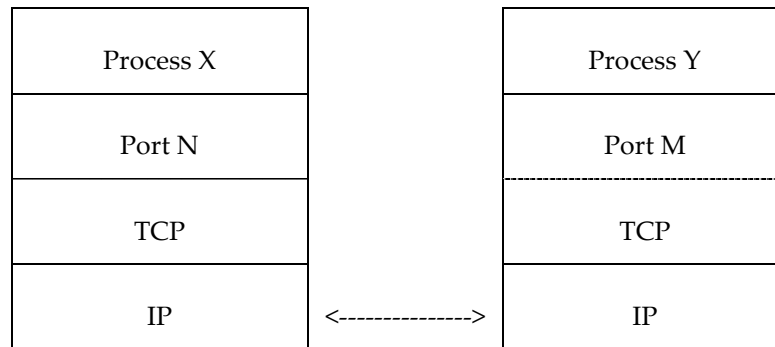
- Gain a better understanding of Transmission Control Protocol (TCP) / Internet Protocol (IP);
- Experience sockets, a communication mechanism, in programming; and
- Use TCP and User Datagram Protocol (UDP) with sockets.

Brief review of TCP/IP protocol suite

Application	FTP, TELNET, HTTP, applications
Transport	TCP, UDP
Network	IP, ICMP, ARP
Data link	Ethernet, ...
Physical	

- Transmission Control Protocol (TCP)
 - Connection oriented
 - Byte stream
 - Reliable delivery, error and flow control
 - Full duplex
- User Datagram Protocol (UDP)
 - Connectionless
 - Block transfer
 - Best effort delivery

- Full duplex
- Internet Protocol (IP)
 - Best effort delivery



Information needed to exchange data between two application peers over the Internet

- Client/server service model
- Source and destination port addresses
 - Source (or called local) and **destination** (or called remote) **port**:
 - Destination ports should be fixed and known to clients.
 - Source ports are allocated by system kernels.
 - **Protocol**: TCP or UDP
- Source and **destination addresses**
 - IP addresses
 - Destination addresses should be given from the user in the form of dotted decimals or domain names.
 - Source addresses are allocated by system kernels.

Introduction to sockets

A **socket** is a communication mechanism and is identified by an integer that is called the **socket descriptor**.

A socket includes an internal data structure consisting of the following five data items.

- Protocol id;
- The destination host address;
- The destination port number if applicable;
- The local host address;
- The local port number if applicable.

Server applications run first with specific port numbers, and client applications contact the server applications with the server port numbers and the server system address. For popular server applications, the fixed well known port numbers are assigned, so that users can use the applications with the same port numbers. There are a number of functions related to socket to support client/server communication paradigm. These are:

socket()	create a socket
bind()	associate a socket with a server port number
connect()	connect a socket to a server system with its address
listen()	wait for a connection request from a client system
accept()	accept the connection request

In addition, the functions `setsockopt()`, `getsockopt()`, `fcntl()` and `ioctl()` may be used to manipulate the properties of a socket, the function `select()` may be used to identify sockets with particular communication statuses. The function `close()` may be used to close a socket.

Data can be written to a socket using any of the functions `write()`, `send()`, `sendto()` and `sendmsg()`. Data can be read from a socket using any of the functions `read()`, `recv()`, `recvfrom()` and `recvmsg()`.

IPv4 socket address structure

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>           // required header files

struct in_addr {                  // data structure for an
Internet address
    in_addr_t    s_addr;          // 32-bit IPv4 address
                                   // network byte ordered
};

struct sockaddr_in {              // data structure for a
socket address
    sa_family_t    sin_family;    // AF_INET for IPv4
    in_port_t      sin_port;      // 16-bit TCP or UDP port in
network byte order
    struct in_addr  sin_addr;      // 32-bit IPv4 address
    char            sin_zero[8];   // unused
};

-----

int8_t      signed 8-bit integer
<sys/types.h>
uint8_t     unsigned 8-bit integer
int16_t     signed 16-bit integer
uint16_t    unsigned 16-bit integer
int32_t     signed 32-bit integer
uint32_t    unsigned 32-bit integer
```

```

sa_family_t address family of socket address structure
<sys/socket.h>

socklen_t    length of socket address structure, normally uint32_t

in_addr_t    IPv4 address, normally uint32_t
<netinet/in.h>

in_port_t    TCP or UDP port, normally uint16_t
              0 - 1023: well known port
              1024 - 49151: registered port
              49152 - 65535: dynamic and client port

```

Example:

```

#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

struct sockaddr_in    server;

server.sin_family = AF_INET;           // Address Family -
Internet

server.sin_port    = htons(9876);      // host byte
order to network byte order for short

server.sin_addr.s_addr    = inet_addr("198.162.21.132");

```

Network byte order

- Different microprocessors use different models of main memory with respect to addressing for non-one byte data type, like int, short, float, and double.
- Little-endian byte order
 - High-address byte → high-order byte
 - Intel
- Big-endian byte order
 - Low-address byte → high-order byte
 - Motorola

- The numbers used in an Intel system are interpreted in a Motorola system as different numbers. For example, 1 in an Intel system is a very large number in a Motorola system.
- Therefore, we need a common mechanism to handle the above problem. Before senders send data, they change the data from **host byte order** to **network byte order**, and when receivers receive data, they change the data from network byte order back to host byte order. Here are related functions:

```
#include <netinet/in.h>

uint16_t    htons(uint16_t); // host to network short integer,
e.g., port numbers
uint32_t    htonl(uint32_t); // host to network long integer,
e.g., IP addresses
uint16_t    ntohs(uint16_t); // network to host short
uint32_t    ntohl(uint32_t); // network to host long
```

Address transformation functions

```
// "206.62.226.35" <-> read address of integer value

#include <arpa/inet.h>

int_addr_t  inet_addr(const char *strptr); // returns
network byte order
char*       inet_ntoa(struct in_addr inaddr); // from network
byte order

//-- sample code -- convert dotted address to integer

struct sockaddr_in server_addr;
server_addr.sin_addr.s_addr = inet_addr(argv[1]); // argv[1]
contains a dotted address
```

```
    printf("%x\n", server_addr.sin_addr.s_addr);           // %x means
hexadecimal values
    printf("%s\n", inet_ntoa(server_addr.sin_addr));
```

Information about remote hosts, using domain names

```
#include <netdb.h>

    struct hostent    *gethostbyname(const char *name); // domain
name or dotted

    struct hostent    *gethostbyaddr(const char *addr, size_t len,
int family);

    //-- sample code -- convert a domain name to address

    struct hostent *host;
    struct in_addr addr;

    host = gethostbyname(argv[1]);           // argv[1] contains a domain
name of a host
    if (host != 0) {
        addr.s_addr = *((u_long *) (host->h_addr_list[0]));
        printf("address for host: %s\n", inet_ntoa(addr));
    }

    addr.s_addr = inet_addr(argv[2]);        // contains a dotted address

    host = gethostbyaddr((char *) (&addr.s_addr),
sizeof(addr.s_addr), AF_INET);

    if (host != 0)
        printf("domain name: %s\n", host->h_name);
```

Note: We will not use those functions in this course.

Byte manipulation functions

```
#include <strings.h>
```

```
void    bzero(void *dest, size_t nbytes);
```

```
void    bcopy(const void *src, void *dest, size_t ntypes);
```

Example:

```
struct sockaddr_in server_addr;

char    buf[128];

bzero(buf, 128);                                // there is no '&' before buf

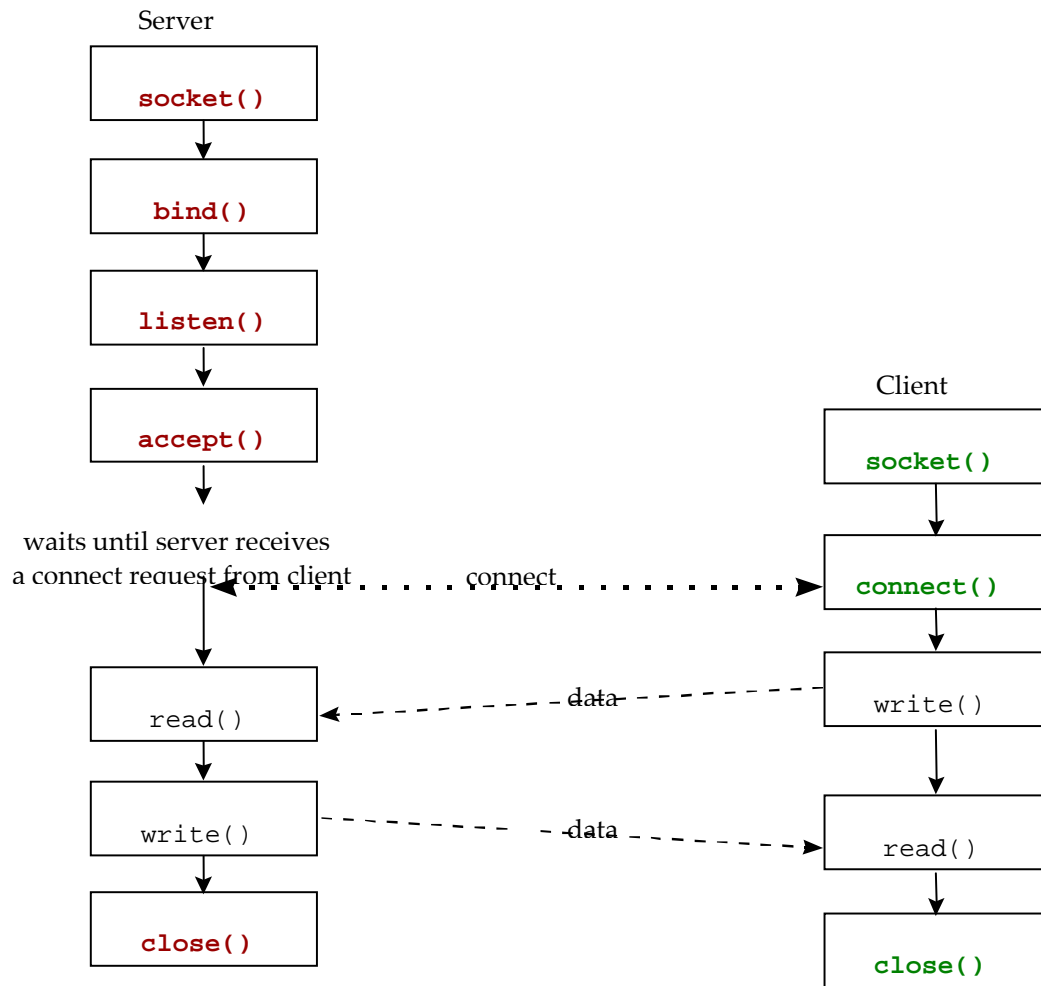
bzero(&server_addr, sizeof(server_addr));        // sizeof() returns the
number bytes allocated

...;

bcopy(&server_addr, buf, sizeof(server_addr));    // important example; there
is no '&' before buf
```


Elementary TCP Sockets

Introduction



Socket(2) function

To perform network i/o, the first thing a process must do is to call the `socket()` function, specifying the type of communication protocols desired.

```
#include <sys/types.h>
#include <sys/socket.h>
int  socket(int domain, int type, int protocol);
```

<i>domain</i>	PF_INET	IPv4 protocols	
	PF_INET6	IPv6 protocols	
	PF_LOCAL	Unix domain protocols	
	PF_PACKET	Low level packet interface, e.g., Ethernet	
	...		
<i>type</i>	SOCK_STREAM	stream socket	TCP
	SOCK_DGRAM	datagram socket	UDP
	SOCK_RAW	raw socket	IP
	SOCK_PACKET	data link socket (obsolete; replaced by	
packet(7))			
<i>protocol</i>	0	usually 0 for SOCK_STREAM and	
SOCK_DGRAM			

Example:

```
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);    // for TCP connections
```

bind(2) function

The `bind()` function assigns a local protocol address, i.e., port number and address to permit, to a socket.

```
#include <sys/types.h>

#include <sys/socket.h>

int  bind(int sockfd, struct sockaddr *my_addr, socklen_t
addrlen);
```

Comment: `struct sockaddr_in*` (not `struct sockaddr*`) should be used for `my_addr`. Similarly, it is also applied to `accept()`, `connect()`, `recvfrom()` and `sendto()`.

Example:

```
int sd, r,
struct sockaddr_in server_addr;
sd = socket(PF_INET, SOCK_STREAM, 0);
bzero(&server_addr, sizeof(server_addr)); //
clear
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = inet_addr(argv[1]); // argv[1]
contains a dotted address
server_addr.sin_port = htons(atoi(argv[2])); // argv[2]
contains a server port #
r = bind(sd, (struct sockaddr*)(&server_addr),
sizeof(server_addr)); // see the type conversion
```

You really need to check all the return values from the system calls. If `r` is negative in the above example, your server port number is probably used by another program.

listen(2) function

The `listen()` function converts an unconnected socket into a **passive socket**, indicating that the kernel should accept incoming connection requests.

```
#include <sys/socket.h>

int  listen(int sockfd, int backlog);
```

backlog the maximum number of connections that the socket can allow

accept(2) function

It is called by a TCP server to return the next completed connection, i.e., a new socket fd for the connection.

```
#include <sys/types.h>
#include <sys/socket.h>

int  accept(int sockfd, struct sockaddr *cliaddr,
            socklen_t *addrlen); // cliaddr contains the
                                client's info
```

Example:

```
int client_socket, addr_length;
struct sockaddr_in client_addr;
addr_length = sizeof(client_addr);

client_socket = accept (sd, (struct sockaddr*)&client_addr,
&addr_length);
```

connect(2) function

The `connect()` function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/types.h>
#include <sys/socket.h>

int    connect(int sockfd, struct sockaddr *servaddr,
               socklen_t addrlen); // servaddr contains the
                                   server's info
```

Example:

```
struct sockaddr_in server_addr;

r = connect (sd, (struct sockaddr *)&server_addr,
            sizeof(server_addr));
```

Concurrent servers

```
pid_t pid;
int    listenfd, connfd;

listenfd = socket(...);
bind(listenfd, ...);
listen(listenfd, ...);
for ( ; ; ) {
    connfd = accept(listenfd, ...);
    if ((pid = fork()) == 0) {
        close(listenfd);
        doit(connfd); // do something for the purpose
        close(connfd);
        exit(0);
    }
    close(connfd);
}
```

read() and write() functions

The normal UNIX `read()` and `write()` functions are used to read data from a TCP socket and to write data to a TCP socket respectively. **`read()` returns a negative number when a peer is closed by calling `close()`.**

close() functions

The normal UNIX `close()` function is used to close a socket and terminate a TCP connection.

Exercise Program: TCP_echo_server.c

```
// argv[1] : port number

#include ...
#include ...
...

int sock_server, sock_client, r, len;
char buf[512];
struct sockaddr_in my_addr, client_addr; // not struct sockaddr
type

sock_server = socket (PF_INET, ..., 0); // create a socket for
TCP
if (sock_server < 0)
    ...

bzero(&my_addr, sizeof(my_addr)); // clear

my_addr.sin_family = ...; //Address Family INET
my_addr.sin_port = htons(atoi(argv[...])); // server port number
```

```
my_addr.sin_addr.s_addr = htonl(INADDR_ANY); // accept from
anywhere

r = bind(sock_server, (struct sockaddr *)&my_addr,
sizeof(my_addr));
if (r < 0)
    ...

r = ...; // listen requests
if (r < 0)
    ...

... { // infinite loop
    len = sizeof(client_addr);
    sock_client = accept(sock_server, (struct sockaddr
*)&client_addr, &len);

    n = read (sock_client, buf, ...); // read/write with
client socket
    while (n > 0) { // read() returns the # of
bytes read
        write (...); // read() returns
negative value for broken connection
        ... = read (...);
    }

    close(...); // close the client socket
}

...; // close the server socket
```

Exercise Program: TCP_echo_client.c

```
// argv[1] : dotted address of server
// argv[2] : port number; should be the one used by the TCP echo
server
// argv[3] : message to send
// ...

#include ...

...

int    s, n, port_no, r;
struct sockaddr_in server_addr;
char   *haddr, *message;
char   buf[BUF_LEN+1];

s = socket(... // create a socket for TCP

bzero((char *)&server_addr, sizeof(server_addr)); // clear

server_addr.sin_family = ...; // IPv4 Internet family
server_addr.sin_addr.s_addr = inet_addr(...); // server address
server_addr.sin_port = htons(...); // server port number

r = connect(s, (struct sockaddr *)&server_addr,
sizeof(server_addr)); // connect to the server

write(s, ..., ...); // write a message

n = read(s, ..., ...); // read
if (n < 0)
    ...
else {
```

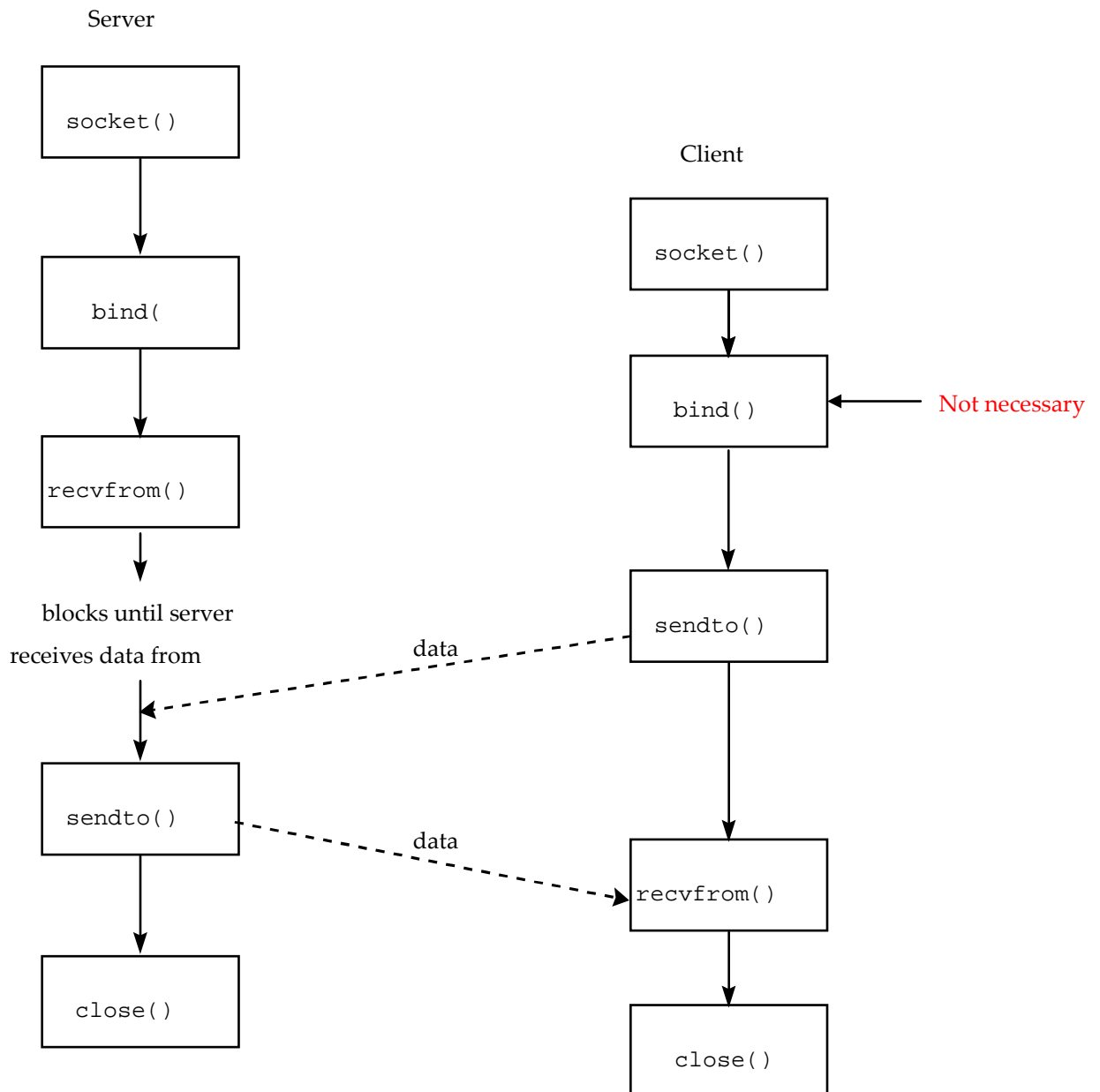


```
        buf[n] = '\\0'; // in order to put the terminating byte of
a string
        ... // print
    }

    close(s);
```

Elementary UDP Sockets

Introduction



recvfrom(2) and sendto(2) functions

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buf, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, void *buf, size_t nbytes, int flags,
               struct sockaddr *to, socklen_t addrlen);

//-- sample code - server
int sockfd, n, len;
char msg[MAXLINE];
struct sockaddr_in cliaddr;
sockfd = ... // create a UDP socket
... // bind

n = recvfrom (sockfd, msg, MAXLINE, 0, (struct sockaddr
*)&cliaddr, &len);

        sendto(sockfd, msg, n,          0, (struct sockaddr
*)&cliaddr, len);
```