MODULE 41
NETWORK PROGRAMMING
SOCKET PART III
-Program Examples-

Note:
This is a continuation from Part II, Module40. Working program examples compiled using `gcc`, tested using the public IPs, run on Fedora 3 with several times update, as normal user. The Fedora machine used for the testing having the "**`No Stack Execute`**" disabled and the **SELinux** set to default configuration. All the program example is generic. Beware codes that expand more than one line. Have a nice ride lol!

This Module will cover the following sub-topics:

- Example: `select()` server
- Connecting a TCP server and client:
  - Example: Connecting a TCP server to a client, a server program
  - Example: Connecting a TCP client to a server, a client program
- UDP connectionless client/server
- Connecting a UDP server and client:
  - Example: Connecting a UDP server to a client, a server program
  - Example: Connecting a UDP client to a server, a client program
- Connection-oriented server designs:
  - Iterative server
  - `spawn()` server and `spawn()` worker
  - `sendmsg()` server and `recvmsg()` worker
  - Multiple `accept()` servers and multiple `accept()` workers
  - Example: Writing an iterative server program
  - Example: Connection-oriented common client
  - Example: Sending and receiving a multicast datagram
  - Example: Sending a multicast datagram, a server program
  - Example: Receiving a multicast datagram, a client

Example: `select()` server

- The following program example acts like a simple multi-user chat server. Start running it in one window, then `telnet` to it ("`telnet hostname 2020`") from multiple other windows.
- When you type something in one `telnet` session, it should appear in all the others windows.

```
/*******select.c*********/
/*******Using select() for I/O multiplexing*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
/* port we're listening on */
#define PORT 2020

int main(int argc, char *argv[])
{
/* master file descriptor list */
fd_set master;
/* temp file descriptor list for select() */
fd_set read_fds;
/* server address */
struct sockaddr_in serveraddr;
/* client address */
struct sockaddr_in clientaddr;
/* maximum file descriptor number */
int fdmax;
/* listening socket descriptor */
int listener;
/* newly accept()ed socket descriptor */
int newfd;
/* buffer for client data */
char buf[1024];
```

```
int nbytes;
/* for setsockopt() SO_REUSEADDR, below */
int yes = 1;
int addrlen;
int i, j;
/* clear the master and temp sets */
FD_ZERO(&master);
FD_ZERO(&read_fds);

/* get the listener */
if((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("Server-socket() error lol!");
/*just exit lol!*/
exit(1);
}
printf("Server-socket() is OK...\n");
/*"address already in use" error message */
if(setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
{
perror("Server-setsockopt() error lol!");
exit(1);
}
printf("Server-setsockopt() is OK...\n");

/* bind */
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = INADDR_ANY;
serveraddr.sin_port = htons(PORT);
memset(&(serveraddr.sin_zero), '\0', 8);

if(bind(listener, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) == -1)
{
perror("Server-bind() error lol!");
exit(1);
}
printf("Server-bind() is OK...\n");

/* listen */
if(listen(listener, 10) == -1)
{
perror("Server-listen() error lol!");
exit(1);
}
printf("Server-listen() is OK...\n");

/* add the listener to the master set */
FD_SET(listener, &master);
/* keep track of the biggest file descriptor */
fdmax = listener; /* so far, it's this one*/

/* loop */
for(;;)
{
/* copy it */
read_fds = master;

if(select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1)
{
perror("Server-select() error lol!");
exit(1);
}
printf("Server-select() is OK...\n");

/*run through the existing connections looking for data to be read*/
for(i = 0; i <= fdmax; i++)
{
if(FD_ISSET(i, &read_fds))
{ /* we got one... */
if(i == listener)
{
/* handle new connections */
addrlen = sizeof(clientaddr);
if((newfd = accept(listener, (struct sockaddr *)&clientaddr, &addrlen)) == -1)
{
perror("Server-accept() error lol!");
}
else
{
printf("Server-accept() is OK...\n");
```

```
FD_SET(newfd, &master); /* add to master set */
if(newfd > fdmax)
{ /* keep track of the maximum */
fdmax = newfd;
}
printf("%s: New connection from %s on socket %d\n", argv[0],
inet_ntoa(clientaddr.sin_addr), newfd);
}
}
else
{
/* handle data from a client */
if((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0)
{
/* got error or connection closed by client */
if(nbytes == 0)
 /* connection closed */
 printf("%s: socket %d hung up\n", argv[0], i);

else
perror("recv() error lol!");

/* close it... */
close(i);
/* remove from master set */
FD_CLR(i, &master);
}
else
{
/* we got some data from a client*/
for(j = 0; j <= fdmax; j++)
{
/* send to everyone! */
if(FD_ISSET(j, &master))
{
/*except the listener and ourselves */
       if(j != listener && j != i)
       {
              if(send(j, buf, nbytes, 0) == -1)
                     perror("send() error lol!");
       }
}
}
}
}
}
}
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g select.c -o select
```

- Run the program.

```
[bodo@bakawali testsocket]$ ./select
Server-socket() is OK...
Server-setsockopt() is OK...
Server-bind() is OK...
Server-listen() is OK...
```

- You can leave the program running at the background (`Ctrl + z`).

```
[bodo@bakawali testsocket]$ ./select
Server-socket() is OK...
Server-setsockopt() is OK...
Server-bind() is OK...
Server-listen() is OK...

[1]+  Stopped                 ./select
[bodo@bakawali testsocket]$ bg
[1]+ ./select &
[bodo@bakawali testsocket]$
```

- Do some verification.

```
[bodo@bakawali testsocket]$ ps aux | grep select
bodo     27474  0.0  0.2  1384  292 pts/2    S+   14:32   0:00 ./select
bodo     27507  0.0  0.5  3724  668 pts/3    S+   14:34   0:00 grep select
[bodo@bakawali testsocket]$ netstat -a |grep 2020
tcp        0      0 *:2020     *:*                 LISTEN
[bodo@bakawali testsocket]$
```

- Telnet from other computers or windows using hostname or the IP address. Here we use hostname, bakawali. Use escape character (`Ctrl + ]`) to terminate command. For other telnet command please type `help`.

```
[bodo@bakawali testsocket]$ telnet bakawali 2020
Trying 203.106.93.94...
Connected to bakawali.jmti.gov.my (203.106.93.94).
Escape character is '^]'.
^]
telnet> mode line
testing some text
the most visible one
```

- The last two messages were typed at another two machines that connected through socket 5 and 6 (socket 4 is another window of the server) using telnet. Socket 5 and 6 are from Windows 2000 Server machines.
- The following are messages on the server console. There are another two machine connected to the server and the messages at the server console is shown below.

```
[bodo@bakawali testsocket]$ Server-select() is OK...
Server-accept() is OK...
./select: New connection from 203.106.93.94 on socket 4
Server-select() is OK...
...
Server-accept() is OK...
./select: New connection from 203.106.93.91 on socket 5
Server-select() is OK...
Server-select() is OK...
...
Server-select() is OK...
Server-select() is OK...
Server-accept() is OK...
./select: New connection from 203.106.93.82 on socket 6
```

- When the clients disconnected from the server through socket 4, 5 and 6, the following messages appear on the server console.

```
...
Server-select() is OK...
Server-select() is OK...
./select: socket 5 hung up
Server-select() is OK...
./select: socket 6 hung up
Server-select() is OK...
./select: socket 4 hung up
```

- There are two file descriptor sets in the code: `master` and `read_fds`. The first, `master`, holds all the socket descriptors that are currently connected, as well as the socket descriptor that is listening for new connections.
- The reason we have the `master` set is that `select()` actually changes the set you pass into it to reflect which sockets are ready to read. Since we have to keep track of the connections from one call of `select()` to the next, we must store these safely away somewhere. At the last minute, we copy the `master` into the `read_fds`, and then call `select()`.
- Then every time we get a new connection, we have to add it to the `master` set and also every time a connection closes, we have to remove it from the `master` set.
- Notice that we check to see when the `listener` socket is ready to read. When it is, it means we have a new connection pending, and we `accept()` it and add it to the `master` set. Similarly, when a client

connection is ready to read, and `recv()` returns `0`, we know that the client has closed the connection, and we must remove it from the `master` set.

- If the client `recv()` returns non-zero, though, we know some data has been received. So we get it, and then go through the `master` list and send that data to all the rest of the connected clients.

Connecting a TCP server and client

- The following program examples are connection-oriented where sockets use TCP to connect a server to a client, and a client to a server. This example provides more complete sockets' APIs usage.

Example: Connecting a TCP server to a client, a server program

```
/************tcpserver.c**********************/
/* Header files needed to use the sockets API. */
/* File contain Macro, Data Type and Structure */
/*********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
/* BufferLength is 100 bytes */
#define BufferLength 100
/* Server port number */
#define SERVPORT 3111

int main()
{
/* Variable and structure definitions. */
int sd, sd2, rc, length = sizeof(int);
int totalcnt = 0, on = 1;
char temp;
char buffer[BufferLength];
struct sockaddr_in serveraddr;
struct sockaddr_in their_addr;

fd_set read_fd;
struct timeval timeout;
timeout.tv_sec = 15;
timeout.tv_usec = 0;

/* The socket() function returns a socket descriptor */
/* representing an endpoint. The statement also */
/* identifies that the INET (Internet Protocol) */
/* address family with the TCP transport (SOCK_STREAM) */
/* will be used for this socket. */
/*********************************************/
/* Get a socket descriptor */
if((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
perror("Server-socket() error");
/* Just exit */
exit (-1);
}
else
printf("Server-socket() is OK\n");

/* The setsockopt() function is used to allow */
/* the local address to be reused when the server */
/* is restarted before the required wait time */
/* expires. */
/*********************************************/
/* Allow socket descriptor to be reusable */
if((rc = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on))) < 0)
{
perror("Server-setsockopt() error");
close(sd);
exit (-1);
}
else
printf("Server-setsockopt() is OK\n");

/* bind to an address */
```

```
memset(&serveraddr, 0x00, sizeof(struct sockaddr_in));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVPORT);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

printf("Using %s, listening at %d\n", inet_ntoa(serveraddr.sin_addr), SERVPORT);

/* After the socket descriptor is created, a bind() */
/* function gets a unique name for the socket. */
/* In this example, the user sets the */
/* s_addr to zero, which allows the system to */
/* connect to any client that used port 3005. */
if((rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr))) < 0)
{
perror("Server-bind() error");
/* Close the socket descriptor */
close(sd);
/* and just exit */
exit(-1);
}
else
printf("Server-bind() is OK\n");

/* The listen() function allows the server to accept */
/* incoming client connections. In this example, */
/* the backlog is set to 10. This means that the */
/* system can queue up to 10 connection requests before */
/* the system starts rejecting incoming requests.*/
/************************************************/
/* Up to 10 clients can be queued */
if((rc = listen(sd, 10)) < 0)
{
perror("Server-listen() error");
close(sd);
exit (-1);
}
else
printf("Server-Ready for client connection...\n");

/* The server will accept a connection request */
/* with this accept() function, provided the */
/* connection request does the following: */
/* - Is part of the same address family */
/* - Uses streams sockets (TCP) */
/* - Attempts to connect to the specified port */
/***********************************************/
/* accept() the incoming connection request. */

 int sin_size = sizeof(struct sockaddr_in);
if((sd2 = accept(sd, (struct sockaddr *)&their_addr, &sin_size)) < 0)
{
perror("Server-accept() error");
close(sd);
exit (-1);
}
else
printf("Server-accept() is OK\n");

/*client IP*/
printf("Server-new socket, sd2 is OK...\n");
printf("Got connection from the f***ing client: %s\n", inet_ntoa(their_addr.sin_addr));

/* The select() function allows the process to */
/* wait for an event to occur and to wake up */
/* the process when the event occurs. In this */
/* example, the system notifies the process */
/* only when data is available to read. */
/***********************************************/
/* Wait for up to 15 seconds on */
/* select() for data to be read. */
FD_ZERO(&read_fd);
FD_SET(sd2, &read_fd);
rc = select(sd2+1, &read_fd, NULL, NULL, &timeout);
if((rc == 1) && (FD_ISSET(sd2, &read_fd)))
{
/* Read data from the client. */
totalcnt = 0;

while(totalcnt < BufferLength)
{
```

```
/* When select() indicates that there is data */
/* available, use the read() function to read */
/* 100 bytes of the string that the */
/* client sent. */
/*****************************************/
/* read() from client */
rc = read(sd2, &buffer[totalcnt], (BufferLength - totalcnt));
if(rc < 0)
{
perror("Server-read() error");
close(sd);
close(sd2);
exit (-1);
}
else if (rc == 0)
{
printf("Client program has issued a close()\n");
close(sd);
close(sd2);
exit(-1);
}
else
{
    totalcnt += rc;
    printf("Server-read() is OK\n");
}

}
}
else if (rc < 0)
{
perror("Server-select() error");
close(sd);
close(sd2);
exit(-1);
}
/* rc == 0 */
else
{
printf("Server-select() timed out.\n");
close(sd);
close(sd2);
exit(-1);
}

/*Shows the data*/
printf("Received data from the f***ing client: %s\n", buffer);

/* Echo some bytes of string, back */
/* to the client by using the write() */
/* function. */
/***********************************/
/* write() some bytes of string, */
/* back to the client. */
printf("Server-Echoing back to client...\n");
rc = write(sd2, buffer, totalcnt);
if(rc != totalcnt)
{
perror("Server-write() error");
/* Get the error number. */
rc = getsockopt(sd2, SOL_SOCKET, SO_ERROR, &temp, &length);
if(rc == 0)
{
/* Print out the asynchronously */
/* received error. */
errno = temp;
perror("SO_ERROR was: ");
}
else
printf("Server-write() is OK\n");

close(sd);
close(sd2);
exit(-1);
}

/* When the data has been sent, close() */
/* the socket descriptor that was returned */
/* from the accept() verb and close() the */
/* original socket descriptor. */
```

```
/*****************************************/
/* Close the connection to the client and */
/* close the server listening socket. */
/*****************************************/
close(sd2);
close(sd);
exit(0);
return 0;
}
```

- Compile and link the program. Make sure there is no error.

```
[bodo@bakawali testsocket]$ gcc -g tcpserver.c -o tcpserver
```

- Run the program. In this example we let the program run in the background.

```
[bodo@bakawali testsocket]$ ./tcpserver
Server-socket() is OK
Server-setsockopt() is OK
Using 0.0.0.0, listening at 3111
Server-bind() is OK
Server-Ready for client connection...

[1]+  Stopped                 ./tcpserver
[bodo@bakawali testsocket]$ bg
[1]+ ./tcpserver &
[bodo@bakawali testsocket]$
```

- Do some verification.

```
[bodo@bakawali testsocket]$ ps aux | grep tcpserver
bodo      7914  0.0  0.2  3172  324 pts/3    S    11:59   0:00 ./tcpserver
bodo      7921  0.0  0.5  5540  648 pts/3    S+   12:01   0:00 grep tcpserver
[bodo@bakawali testsocket]$ netstat -a | grep 3111
tcp        0      0 *:3111              *:*                     LISTEN
```

- When the next program example (the TCP client) is run, the following messages should be expected at the server console.

```
[bodo@bakawali testsocket]$ Server-accept() is OK
Server-new socket, sd2 is OK...
Got connection from the f***ing client: 203.106.93.94
Server-read() is OK
Received data from the f***ing client: This is a test string from client lol!!!
Server-Echoing back to client...

[1]+  Done                    ./tcpserver
[bodo@bakawali testsocket]$
```

- If the server program and then the client are run, the following messages should be expected at the server console.

```
[bodo@bakawali testsocket]$ ./tcpserver
Server-socket() is OK
Server-setsockopt() is OK
Using 0.0.0.0, listening at 3111
Server-bind() is OK
Server-Ready for client connection...
Server-accept() is OK
Server-new socket, sd2 is OK...
Got connection from the f***ing client: 203.106.93.94
Server-read() is OK
Received data from the f***ing client: This is a test string from client lol!!!
Server-Echoing back to client...
[bodo@bakawali testsocket]$
```

- Just telneting the server.

```
[bodo@bakawali testsocket]$ telnet 203.106.93.94 3111
```

```
Trying 203.106.93.94...
Connected to bakawali.jmti.gov.my (203.106.93.94).
Escape character is '^]'.
^]
telnet> help
Commands may be abbreviated.  Commands are:

close           close current connection
logout          forcibly logout remote user and close the connection
display         display operating parameters
mode            try to enter line or character mode ('mode ?' for more)
open            connect to a site
quit            exit telnet
send            transmit special characters ('send ?' for more)
set             set operating parameters ('set ?' for more)
unset           unset operating parameters ('unset ?' for more)
status          print status information
toggle          toggle operating parameters ('toggle ?' for more)
slc             change state of special characters ('slc ?' for more)
auth            turn on (off) authentication ('auth ?' for more)
encrypt         turn on (off) encryption ('encrypt ?' for more)
forward         turn on (off) credential forwarding ('forward ?' for more)
z               suspend telnet
!               invoke a subshell
environ         change environment variables ('environ ?' for more)
?               print help information
telnet>quit
```

- Well, it looks that we have had a telnet session with the server.

Example: Connecting a TCP client to a server, a client program

- Well, let try the client program that will connect to the previous server program.
- The following example shows how to connect a socket client program to a connection-oriented server.

```
/************tcpclient.c**********************/
/* Header files needed to use the sockets API. */
/* File contains Macro, Data Type and */
/* Structure definitions along with Function */
/* prototypes. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
/* BufferLength is 100 bytes */
#define BufferLength 100
/* Default host name of server system. Change it to your default */
/* server hostname or IP.  If the user do not supply the hostname */
/* as an argument, the_server_name_or_IP will be used as default*/
#define SERVER "The_server_name_or_IP"
/* Server's port number */
#define SERVPORT 3111

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or */
/* set the server name in the #define SERVER ... */
int main(int argc, char *argv[])
{
/* Variable and structure definitions. */
int sd, rc, length = sizeof(int);
struct sockaddr_in serveraddr;
char buffer[BufferLength];
char server[255];
char temp;
int totalcnt = 0;
struct hostent *hostp;
char data[100] = "This is a test string from client lol!!! ";
```

```
/* The socket() function returns a socket */
/* descriptor representing an endpoint. */
/* The statement also identifies that the */
/* INET (Internet Protocol) address family */
/* with the TCP transport (SOCK_STREAM) */
/* will be used for this socket. */
/*****************************************/
/* get a socket descriptor */
if((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
perror("Client-socket() error");
exit(-1);
}
else
printf("Client-socket() OK\n");
/*If the server hostname is supplied*/
if(argc > 1)
{
/*Use the supplied argument*/
strcpy(server, argv[1]);
printf("Connecting to the f***ing %s, port %d ...\n", server, SERVPORT);
}
else
/*Use the default server name or IP*/
strcpy(server, SERVER);

memset(&serveraddr, 0x00, sizeof(struct sockaddr_in));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVPORT);

if((serveraddr.sin_addr.s_addr = inet_addr(server)) == (unsigned long)INADDR_NONE)
{

/* When passing the host name of the server as a */
/* parameter to this program, use the gethostbyname() */
/* function to retrieve the address of the host server. */
/***************************************************/
/* get host address */
hostp = gethostbyname(server);
if(hostp == (struct hostent *)NULL)
{
printf("HOST NOT FOUND --> ");
/* h_errno is usually defined */
/* in netdb.h */
printf("h_errno = %d\n",h_errno);
printf("---This is a client program---\n");
printf("Command usage: %s <server name or IP>\n", argv[0]);
close(sd);
exit(-1);
}
memcpy(&serveraddr.sin_addr, hostp->h_addr, sizeof(serveraddr.sin_addr));
}

/* After the socket descriptor is received, the */
/* connect() function is used to establish a */
/* connection to the server. */
/*********************************************/
/* connect() to server. */
if((rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr))) < 0)
{
perror("Client-connect() error");
close(sd);
exit(-1);
}
else
printf("Connection established...\n");

/* Send string to the server using */
/* the write() function. */
/***************************************/
/* Write() some string to the server. */
printf("Sending some string to the f***ing %s...\n", server);
rc = write(sd, data, sizeof(data));

if(rc < 0)
{
perror("Client-write() error");
rc = getsockopt(sd, SOL_SOCKET, SO_ERROR, &temp, &length);
if(rc == 0)
{
```

```
/* Print out the asynchronously received error. */
errno = temp;
perror("SO_ERROR was");
}
close(sd);
exit(-1);
}
else
{
printf("Client-write() is OK\n");
printf("String successfully sent lol!\n");
printf("Waiting the %s to echo back...\n", server);
}

totalcnt = 0;
while(totalcnt < BufferLength)
{

/* Wait for the server to echo the */
/* string by using the read() function. */
/**************************************/
/* Read data from the server. */
rc = read(sd, &buffer[totalcnt], BufferLength-totalcnt);
if(rc < 0)
{
perror("Client-read() error");
close(sd);
exit(-1);
}
else if (rc == 0)
{
printf("Server program has issued a close()\n");
close(sd);
exit(-1);
}
else
totalcnt += rc;
}
printf("Client-read() is OK\n");
printf("Echoed data from the f***ing server: %s\n", buffer);

/* When the data has been read, close() */
/* the socket descriptor. */
/**************************************/
/* Close socket descriptor from client side. */
close(sd);
exit(0);
return 0;
}
```

- Compile and link the client program.

```
[bodo@bakawali testsocket]$ gcc -g tcpclient.c -o tcpclient
```

- Run the program. Before that don't forget to run the server program first. The first run is without the server hostname/IP.

```
[bodo@bakawali testsocket]$ ./tcpclient
Client-socket() OK
HOST NOT FOUND --> h_errno = 1
---This is a client program---
Command usage: ./tcpclient <server name or IP>
[bodo@bakawali testsocket]$
```

- Then run with the server hostname or IP.

```
[bodo@bakawali testsocket]$ ./tcpclient 203.106.93.94
Client-socket() OK
Connecting to the f***ing 203.106.93.94, port 3111 ...
Connection established...
Sending some string to the f***ing 203.106.93.94...
Client-write() is OK
String successfully sent lol!
Waiting the 203.106.93.94 to echo back...
Client-read() is OK
```

```
Echoed data from the f***ing server: This is a test string from client lol!!!
[bodo@bakawali testsocket]$
```

- And at the server console.

```
[bodo@bakawali testsocket]$ ./tcpserver
Server-socket() is OK
Server-setsockopt() is OK
Using 0.0.0.0, listening at 3111
Server-bind() is OK
Server-Ready for client connection...
Server-accept() is OK
Server-new socket, sd2 is OK...
Got connection from the f***ing client: 203.106.93.94
Server-read() is OK
Received data from the f***ing client: This is a test string from client lol!!!
Server-Echoing back to client...
[bodo@bakawali testsocket]$
```

- Well, it works!

UDP connectionless client/server

- The connectionless protocol server and client examples illustrate the socket APIs that are written for User Datagram Protocol (UDP). The server and client examples use the following sequence of function calls:
  - `socket()`
  - `bind()`
- The following figure illustrates the client/server relationship of the socket APIs for a connectionless protocol.

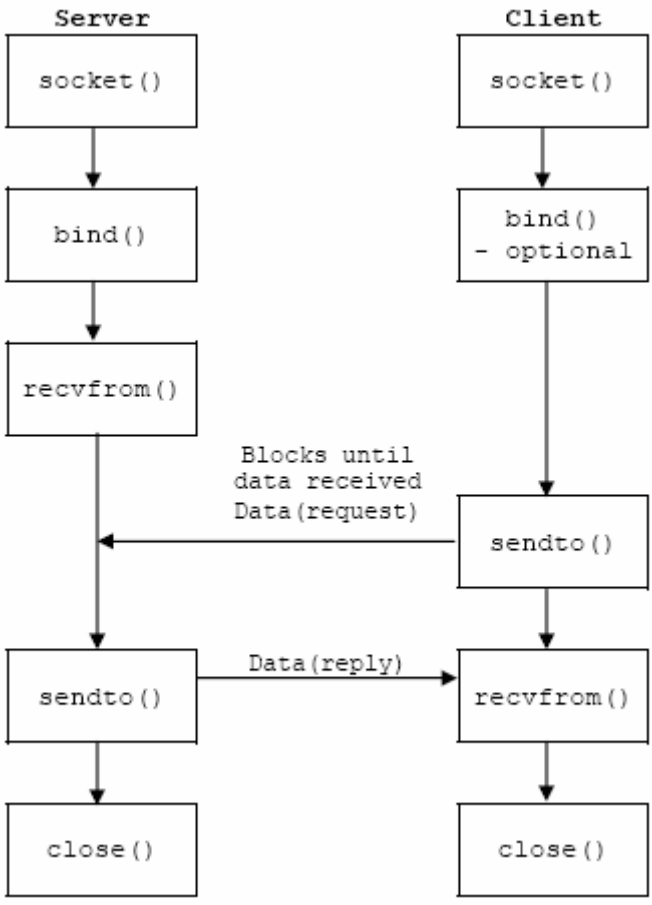

Figure 1: UDP connectionless APIs relationship.

Connecting a UDP server and client

- The following examples show how to use UDP to connect a server to a connectionless client, and a connectionless client to a server.

Example: Connecting a UDP server to a client, a server program

- The first example shows how to use UDP to connect a connectionless socket server program to a client.

```
/******************udpserver.c****************/
/* Header files needed to use the sockets API. */
/* File contain Macro, Data Type and Structure */
/* definitions along with Function prototypes. */
/* header files */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
/* Server's port number, listen at 3333 */
#define SERVPORT 3333

/* Run the server without argument */
int main(int argc, char *argv[])
{
/* Variable and structure definitions. */
int sd, rc;
struct sockaddr_in serveraddr, clientaddr;
int clientaddrlen = sizeof(clientaddr);
int serveraddrlen = sizeof(serveraddr);
char buffer[100];
char *bufptr = buffer;
int buflen = sizeof(buffer);

/* The socket() function returns a socket */
/* descriptor representing an endpoint. */
/* The statement also identifies that the */
/* INET (Internet Protocol) address family */
/* with the UDP transport (SOCK_DGRAM) will */
/* be used for this socket. */
/*****************************************/
/* get a socket descriptor */
if((sd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
{
perror("UDP server - socket() error");
exit(-1);
}
else
printf("UDP server - socket() is OK\n");

printf("UDP server - try to bind...\n");

/* After the socket descriptor is received, */
/* a bind() is done to assign a unique name */
/* to the socket. In this example, the user */
/* set the s_addr to zero. This allows the */
/* system to connect to any client that uses */
/* port 3333. */
/*****************************************/
/* bind to address */
memset(&serveraddr, 0x00, serveraddrlen);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVPORT);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
if((rc = bind(sd, (struct sockaddr *)&serveraddr, serveraddrlen)) < 0)
{
perror("UDP server - bind() error");
close(sd);
/* If something wrong with socket(), just exit lol */
exit(-1);
}
else
printf("UDP server - bind() is OK\n");
```

```
printf("Using IP %s and port %d\n", inet_ntoa(serveraddr.sin_addr), SERVPORT);
printf("UDP server - Listening...\n");

/* Use the recvfrom() function to receive the */
/* data. The recvfrom() function waits */
/* indefinitely for data to arrive. */
/*********************************************/
/* This example does not use flags that control */
/* the reception of the data. */
/*********************************************/
/* Wait on client requests. */
rc = recvfrom(sd, bufptr, buflen, 0, (struct sockaddr *)&clientaddr, &clientaddrlen);
if(rc < 0)
{
perror("UDP Server - recvfrom() error");
close(sd);
exit(-1);
}
else
printf("UDP Server - recvfrom() is OK...\n");

printf("UDP Server received the following:\n \"%s\" message\n", bufptr);
printf("from port %d and address %s.\n", ntohs(clientaddr.sin_port),
inet_ntoa(clientaddr.sin_addr));

/* Send a reply by using the sendto() function. */
/* In this example, the system echoes the received */
/* data back to the client. */
/*********************************************/
/* This example does not use flags that control */
/* the transmission of the data */
/*********************************************/
/* Send a reply, just echo the request */
printf("UDP Server replying to the stupid UDP client...\n");
rc = sendto(sd, bufptr, buflen, 0, (struct sockaddr *)&clientaddr, clientaddrlen);
if(rc < 0)
{
perror("UDP server - sendto() error");
close(sd);
exit(-1);
}
else
printf("UDP Server - sendto() is OK...\n");

/* When the data has been sent, close() the */
/* socket descriptor. */
/*******************************************/
/* close() the socket descriptor. */
close(sd);
exit(0);
}
```

- Compile and link the udp server program.

```
[bodo@bakawali testsocket]$ gcc -g udpserver.c -o udpserver
```

- Run the program and let it run in the background.

```
[bodo@bakawali testsocket]$ ./udpserver
UDP server - socket() is OK
UDP server - try to bind...
UDP server - bind() is OK
Using IP 0.0.0.0 and port 3333
UDP server - Listening...

[1]+  Stopped                 ./udpserver
[bodo@bakawali testsocket]$ bg
[1]+ ./udpserver &
[bodo@bakawali testsocket]$
```

- Verify the program running.

```
[bodo@bakawali testsocket]$ ps aux | grep udpserver
bodo      7963  0.0  0.2  2240  324 pts/2    S    12:22   0:00 ./udpserver
bodo      7965  0.0  0.5  4324  648 pts/2    S+   12:24   0:00 grep udpserver
```

- Verify that the udp server is listening at port 3333 waiting for the client connection.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3333
udp        0      0 *:3333                      *:*
[bodo@bakawali testsocket]$
```

- Without the client program (next example) you can try telneting the server using port 3333 for testing. For this program example the following telnet session cannot be established for UDP/connectionless.

```
[bodo@bakawali testsocket]$ telnet 203.106.93.94 3333
Trying 203.106.93.94...
telnet: connect to address 203.106.93.94: Connection refused
telnet: Unable to connect to remote host: Connection refused
```

Example: Connecting a UDP client to a server, a client program

- The following example shows how to use UDP to connect a connectionless socket client program to a server. This program will be used to connect to the previous UDP server.

```
/***************udpclient.c*******************/
/* Header files needed to use the sockets API. */
/* File contain Macro, Data Type and Structure */
/* definitions along with Function prototypes. */
/*********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/* Host name of my system, change accordingly */
/* Put the server hostname that run the UDP server program */
/* This will be used as default UDP server for client connection */
#define SERVER "bakawali"
/* Server's port number */
#define SERVPORT 3333
/* Pass in 1 argument (argv[1]) which is either the */
/* address or host name of the server, or */
/* set the server name in the #define SERVER above. */

int main(int argc, char *argv[])
{
/* Variable and structure definitions. */
int sd, rc;
struct sockaddr_in serveraddr, clientaddr;
int serveraddrlen = sizeof(serveraddr);
char server[255];
char buffer[100];
char *bufptr = buffer;
int buflen = sizeof(buffer);
struct hostent *hostp;
memset(buffer, 0x00, sizeof(buffer));
/* 36 characters + terminating NULL */
memcpy(buffer, "Hello! A client request message lol!", 37);

/* The socket() function returns a socket */
/* descriptor representing an endpoint. */
/* The statement also identifies that the */
/* INET (Internet Protocol) address family */
/* with the UDP transport (SOCK_DGRAM) will */
/* be used for this socket. */
/*****************************************/
/* get a socket descriptor */
if((sd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
{
perror("UDP Client - socket() error");
/* Just exit lol! */
exit(-1);
}
else
printf("UDP Client - socket() is OK!\n");
```

```
/* If the hostname/IP of the server is supplied */
/* Or if(argc = 2) */
if(argc > 1)
strcpy(server, argv[1]);
else
{
/*Use default hostname or IP*/
printf("UDP Client - Usage %s <Server hostname or IP>\n", argv[0]);
printf("UDP Client - Using default hostname/IP!\n");

strcpy(server, SERVER);
}

memset(&serveraddr, 0x00, sizeof(struct sockaddr_in));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVPORT);

if((serveraddr.sin_addr.s_addr = inet_addr(server)) == (unsigned long)INADDR_NONE)
{
/* Use the gethostbyname() function to retrieve */
/* the address of the host server if the system */
/* passed the host name of the server as a parameter. */
/*********************************************/
/* get server address */
hostp = gethostbyname(server);
if(hostp == (struct hostent *)NULL)
{
printf("HOST NOT FOUND --> ");
/* h_errno is usually defined */
/* in netdb.h */
printf("h_errno = %d\n", h_errno);
exit(-1);
}
else
{
printf("UDP Client - gethostname() of the server is OK... \n");
printf("Connected to UDP server %s on port %d.\n", server, SERVPORT);
}
memcpy(&serveraddr.sin_addr, hostp->h_addr, sizeof(serveraddr.sin_addr));
}

/* Use the sendto() function to send the data */
/* to the server. */
/*********************************************/
/* This example does not use flags that control */
/* the transmission of the data. */
/*********************************************/
/* send request to server */
rc = sendto(sd, bufptr, buflen, 0, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
if(rc < 0)
{
perror("UDP Client - sendto() error");
close(sd);
exit(-1);
}
else
printf("UDP Client - sendto() is OK!\n");

printf("Waiting a reply from UDP server...\n");

/* Use the recvfrom() function to receive the */
/* data back from the server. */
/*********************************************/
/* This example does not use flags that control */
/* the reception of the data. */
/*********************************************/
/* Read server reply. */
/* Note: serveraddr is reset on the recvfrom() function. */
rc = recvfrom(sd, bufptr, buflen, 0, (struct sockaddr *)&serveraddr, &serveraddrlen);

if(rc < 0)
{
perror("UDP Client - recvfrom() error");
close(sd);
exit(-1);
}
else
{
printf("UDP client received the following: \"%s\" message\n", bufptr);
```

```
printf(" from port %d, address %s\n", ntohs(serveraddr.sin_port),
inet_ntoa(serveraddr.sin_addr));
}

/* When the data has been received, close() */
/* the socket descriptor. */
/*****************************************/
/* close() the socket descriptor. */
close(sd);
exit(0);
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g udpclient.c -o udpclient
```

- Run the program. Before that make sure the previous program example (the UDP server) is running.

```
[bodo@bakawali testsocket]$ ./udpclient
UDP Client - socket() is OK!
UDP Client - Usage ./udpclient <Server hostname or IP>
UDP Client - Using default hostname/IP!
UDP Client - gethostname() of the server is OK...
Connected to UDP server bakawali on port 3333.
UDP Client - sendto() is OK!
Waiting a reply from UDP server...
UDP client received the following: "Hello! A client request message lol!"
message
 from port 3333, address 203.106.93.94
[bodo@bakawali testsocket]$
```

- Well, our udp server and client communicated successfully. The following are the expected messages at server console.

```
[bodo@bakawali testsocket]$ ./udpserver
UDP server - socket() is OK
UDP server - try to bind...
UDP server - bind() is OK
Using IP 0.0.0.0 and port 3333
UDP server - Listening...
UDP Server - recvfrom() is OK...
UDP Server received the following:
 "Hello! A client request message lol!" message
from port 32824 and address 203.106.93.94.
UDP Server replying to the stupid UDP client...
UDP Server - sendto() is OK...
[bodo@bakawali testsocket]$
```

Connection-oriented server designs

There are a number of ways that you can design a connection-oriented socket server. While additional socket server designs are possible, the designs provided in the examples below are the most common:

Note: A worker job or a worker thread refers to a process or sub-process (thread) that does data processing by using the socket descriptor. For example, a worker process accesses a database file to extract and format information for sending to the remote peer through the socket descriptor and its associated connection. It could then receive a response or set of data from the remote peer and update the database accordingly.
Depending on the design of the server, the worker usually does not perform the connection "bring-up" or initiation. This is usually done by the listening or server job or thread. The listening or server job usually passes the descriptor to the worker job or thread.

Iterative server

In the iterative server example, a single server job handles all incoming connections and all data flows with the client jobs. When the `accept()` API completes, the server handles the entire transaction. This is the easiest server to develop, but it does have a few problems. While the server is handling the request from a given client, additional clients could be trying to get to the server. These requests fill the `listen()` backlog and some of them will be rejected eventually.

All of the remaining examples are concurrent server designs. In these designs, the system uses multiple jobs and threads to handle the incoming connection requests. With a concurrent server there are usually multiple clients that connect to the server at the same time.

`spawn()` server and `spawn()` worker

The `spawn()` server and `spawn()` worker example uses the `spawn()` API to create a new job (often called a "child job") to handle each incoming request. After `spawn()` completes, the server can then wait on the `accept()` API for the next incoming connection to be received. The only problem with this server design is the performance overhead of creating a new job each time a connection is received. You can avoid the performance overhead of the `spawn()` server example by using pre-started jobs. Instead of creating a new job each time a connection is received, the incoming connection is given to a job that is already active. If the child job is already active, the `sendmsg()` and `recvmsg()` APIs.

`sendmsg()` server and `recvmsg()` worker

Servers that use `sendmsg()` and `recvmsg()` APIs to pass descriptors remain unhindered during heavy activity. They do not need to know which worker job is going to handle each incoming connection. When a server calls `sendmsg()`, the descriptor for the incoming connection and any control data are put in an internal queue for the `AF_UNIX` socket. When a worker job becomes available, it calls `recvmsg()` and receives the first descriptor and the control data that was in the queue.
An example of how you can use the `sendmsg()` API to pass a descriptor to a job that does not exist, a server can do the following:

1. Use the `socketpair()` API to create a pair of `AF_UNIX` sockets.
2. Use the `sendmsg()` API to send a descriptor over one of the `AF_UNIX` sockets created by `socketpair()`.
3. Call `spawn()` to create a child job that inherits the other end of the socket pair.

The child job calls `recvmsg()` to receive the descriptor that the server passed. The child job was not active when the server called `sendmsg()`. The `sendmsg()` and `recvmsg()` APIs are extremely flexible. You can use these APIs to send data buffers, descriptors, or both.

Multiple `accept()` servers and multiple `accept()` workers

In the previous examples, the worker job did not get involved until after the server received the incoming connection request. The multiple `accept()` servers and multiple `accept()` workers example of the system turns each of the worker jobs into an iterative server. The server job still calls the `socket()`, `bind()`, and `listen()` APIs. When the `listen()` call completes, the server creates each of the worker jobs and gives a listening socket to each one of them. All of the worker jobs then call the `accept()` API. When a client tries to connect to the server, only one `accept()` call completes, and that worker handles the connection. This type of design removes the need to give the incoming connection to a worker job, and saves the performance overhead that is associated with that operation. As a result, this design has the best performance. A worker job or a worker thread refers to a process or sub-process (thread) that does data processing by using the socket descriptor. For example, a worker process accesses a database file to extract and format information for sending to the remote peer through the socket descriptor and its associated connection. It could then receive a response or set of data from the remote peer and update the database accordingly.
Depending on the design of the server, the worker usually does not perform the connection "bring-up" or initiation. This is usually done by the listening or server job or thread. The listening or server job usually passes the descriptor to the worker job or thread.

Example: Writing an iterative server program

- This example shows how you can write an iterative server program. The following simple figure illustrates how the server and client jobs interact when the system used the iterative server design.

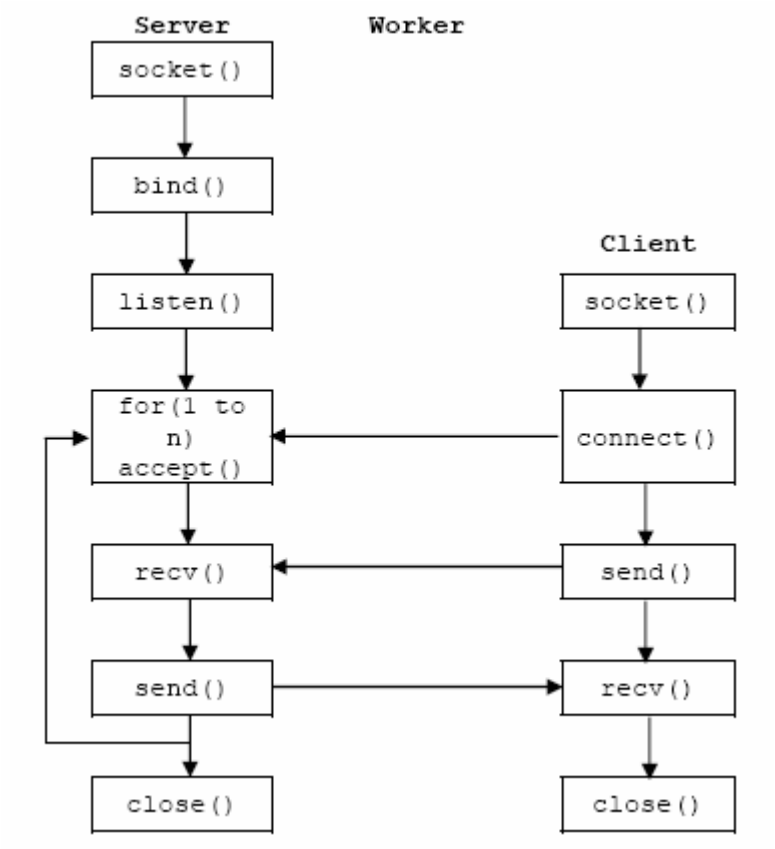

Figure 2: An example of socket APIs used for iterative server design.

- In the following example of the server program, the number of incoming connections that the server allows depends on the first parameter that is passed to the server. The default is for the server to allow only one connection.

```
/**** iserver.c ****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT 12345

/* Run with a number of incoming connection as argument */
int main(int argc, char *argv[])
{
int i, len, num, rc;
int listen_sd, accept_sd;
/* Buffer for data */
char buffer[100];
struct sockaddr_in addr;

/* If an argument was specified, use it to */
/* control the number of incoming connections */
if(argc >= 2)
num = atoi(argv[1]);
/* Prompt some message */
else
{
printf("Usage: %s <The_number_of_client_connection else 1 will be used>\n", argv[0]);
num = 1;
}

/* Create an AF_INET stream socket to receive */
/* incoming connections on */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if(listen_sd < 0)
{
perror("Iserver - socket() error");
exit(-1);
}
else
```

```
printf("Iserver - socket() is OK\n");

printf("Binding the socket...\n");
/* Bind the socket */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(SERVER_PORT);
rc = bind(listen_sd, (struct sockaddr *)&addr, sizeof(addr));
if(rc < 0)
{
perror("Iserver - bind() error");
close(listen_sd);
exit(-1);
}
else
printf("Iserver - bind() is OK\n");

/* Set the listen backlog */
rc = listen(listen_sd, 5);
if(rc < 0)
{
perror("Iserver - listen() error");
close(listen_sd);
exit(-1);
}
else
printf("Iserver - listen() is OK\n");

/* Inform the user that the server is ready */
printf("The Iserver is ready!\n");
/* Go through the loop once for each connection */
for(i=0; i < num; i++)
{
/* Wait for an incoming connection */
printf("Iteration: #%d\n", i+1);
printf(" waiting on accept()\n");
accept_sd = accept(listen_sd, NULL, NULL);
if(accept_sd < 0)
{
perror("Iserver - accept() error");
close(listen_sd);
exit(-1);
}
else
printf("accept() is OK and completed successfully!\n");

/* Receive a message from the client */
printf("I am waiting client(s) to send message(s) to me...\n");
rc = recv(accept_sd, buffer, sizeof(buffer), 0);
if(rc <= 0)
{
perror("Iserver - recv() error");
close(listen_sd);
close(accept_sd);
exit(-1);
}
else
printf("The message from client: \"%s\"\n", buffer);
/* Echo the data back to the client */
printf("Echoing it back to client...\n");
len = rc;
rc = send(accept_sd, buffer, len, 0);
if(rc <= 0)
{
perror("Iserver - send() error");
close(listen_sd);
close(accept_sd);
exit(-1);
}
else
printf("Iserver - send() is OK.\n");
/* Close the incoming connection */
close(accept_sd);
}
/* Close the listen socket */
close(listen_sd);
return 0;
}
```

- Compile and link

```
[bodo@bakawali testsocket]$ gcc -g iserver.c -o iserver
```

- Run the server program.

```
[bodo@bakawali testsocket]$ ./iserver
Usage: ./iserver <The_number_of_client_connection else 1 will be used>
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
```

- The server is waiting the connections from clients. The following program example is a client program.

Example: Connection-oriented common client

- This example provides the code for the client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()`.
- The client job is not aware that the data buffer it sent and received is going to a worker job rather than to the server.
- This client job program can also be used to work with other previous connection-oriented server program examples.

```
/****** comclient.c ******/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
/*Our server port as in the previous program */
#define SERVER_PORT 12345

main (int argc, char *argv[])
{
int len, rc;
int sockfd;
char send_buf[100];
char recv_buf[100];
struct sockaddr_in addr;

if(argc !=2)
{
printf("Usage: %s <Server_name or Server_IP_address>\n", argv[0]);
exit (-1);
}
/* Create an AF_INET stream socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd < 0)
{
perror("client - socket() error");
exit(-1);
}
else
printf("client - socket() is OK.\n");
/* Initialize the socket address structure */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(SERVER_PORT);
/* Connect to the server */
rc = connect(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
if(rc < 0)
{
perror("client - connect() error");
close(sockfd);
exit(-1);
}
else
{
printf("client - connect() is OK.\n");
printf("connect() completed successfully.\n");
```

```
printf("Connection with %s using port %d established!\n", argv[1], SERVER_PORT);
}

/* Enter data buffer that is to be sent */
printf("Enter message to be sent to server:\n");
gets(send_buf);
/* Send data buffer to the worker job */
len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
if(len != strlen(send_buf) + 1)
{
perror("client - send() error");
close(sockfd);
exit(-1);
}
else
printf("client - send() is OK.\n");
printf("%d bytes sent.\n", len);
/* Receive data buffer from the worker job */
len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
if(len != strlen(send_buf) + 1)
{
perror("client - recv() error");
close(sockfd);
exit(-1);
}
else
{
printf("client - recv() is OK.\n");
printf("The sent message: \"%s\" successfully received by server and echoed back to
client!\n", recv_buf);
printf("%d bytes received.\n", len);
}
/* Close the socket */
close(sockfd);
return 0;
}
```

- Compile and link

```
[bodo@bakawali testsocket]$ gcc -g comclient.c -o comclient
/tmp/ccG1hQSw.o(.text+0x171): In function `main':
/home/bodo/testsocket/comclient.c:53: warning: the `gets' function is dangerous
and should not be used.
```

- Run the program and make sure you run the server program as in the previous program example.

```
[bodo@bakawali testsocket]$ ./comclient
Usage: ./comclient <Server_name or Server_IP_address>
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
This is a test message from a stupid client lol!
client - send() is OK.
49 bytes sent.
client - recv() is OK.
The sent message: "This is a test message from a stupid client lol!"
successfully received by server and echoed back to client!
49 bytes received.
[bodo@bakawali testsocket]$
```

- And the message at the server console.

```
[bodo@bakawali testsocket]$ ./iserver
Usage: ./iserver <The_number_of_client_connection else 1 will be used>
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
```

```
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "This is a test message from a stupid client lol!"
Echoing it back to client...
Iserver - send() is OK.
[bodo@bakawali testsocket]$
```

- Let try more than 1 connection. Firstly, run the server.

```
[bodo@bakawali testsocket]$ ./iserver 2
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
```

- Then run the client twice.

```
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
Test message #1
client - send() is OK.
16 bytes sent.
client - recv() is OK.
The sent message: "Test message #1" successfully received by server and echoed
back to client!
16 bytes received.
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
Test message #2
client - send() is OK.
16 bytes sent.
client - recv() is OK.
The sent message: "Test message #2" successfully received by server and echoed
back to client!
16 bytes received.
[bodo@bakawali testsocket]$
```

- The message on the server console.

```
[bodo@bakawali testsocket]$ ./iserver 2
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "Test message #1"
Echoing it back to client...
Iserver - send() is OK.
Iteration: #2
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "Test message #2"
Echoing it back to client...
Iserver - send() is OK.
```

```
[bodo@bakawali testsocket]$
```

Example: Sending and receiving a multicast datagram

- IP multicasting provides the capability for an application to send a single IP datagram that a group of hosts in a network can receive. The hosts that are in the **group** may reside on a single subnet or may be on different subnets that have been connected by multicast capable routers.
- Hosts may join and leave groups at any time. There are no restrictions on the location or number of members in a host group. A **class D** Internet address in the range `224.0.0.1` to `239.255.255.255` identifies a host group.
- An application program can send or receive multicast datagrams by using the `socket()` API and connectionless `SOCK_DGRAM` type sockets. Each multicast transmission is sent from a single network interface, even if the host has more than one multicasting-capable interface.
- It is a one-to-many transmission method. You cannot use connection-oriented sockets of type `SOCK_STREAM` for multicasting.
- When a socket of type `SOCK_DGRAM` is created, an application can use the `setsockopt()` function to control the multicast characteristics associated with that socket. The `setsockopt()` function accepts the following `IPPROTO_IP` level flags:

    1. `IP_ADD_MEMBERSHIP`: Joins the multicast group specified.
    2. `IP_DROP_MEMBERSHIP`: Leaves the multicast group specified.
    3. `IP_MULTICAST_IF`: Sets the interface over which outgoing multicast datagrams are sent.
    4. `IP_MULTICAST_TTL`: Sets the `Time To Live` (`TTL`) in the `IP` header for outgoing multicast datagrams. By default it is set to 1. TTL of 0 are not transmitted on any sub-network. Multicast datagrams with a TTL of greater than 1 may be delivered to more than one sub-network, if there are one or more multicast routers attached to the first sub-network.
    5. `IP_MULTICAST_LOOP`: Specifies whether or not a copy of an outgoing multicast datagram is delivered to the sending host as long as it is a member of the multicast group.

- The following examples enable a socket to send and receive multicast datagrams. The steps needed to send a multicast datagram differ from the steps needed to receive a multicast datagram.

Example: Sending a multicast datagram, a server program

- The following example enables a socket to perform the steps listed below and to send multicast datagrams:

    1. Create an `AF_INET`, `SOCK_DGRAM` type socket.
    2. Initialize a `sockaddr_in` structure with the destination group IP address and port number.
    3. Set the `IP_MULTICAST_LOOP` socket option according to whether the sending system should receive a copy of the multicast datagrams that are transmitted.
    4. Set the `IP_MULTICAST_IF` socket option to define the local interface over which you want to send the multicast datagrams.
    5. Send the datagram.

```
[bodo@bakawali testsocket]$ cat mcastserver.c
/* Send Multicast Datagram code example. */
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

struct in_addr localInterface;
struct sockaddr_in groupSock;
int sd;
char databuf[1024] = "Multicast test message lol!";
int datalen = sizeof(databuf);

int main (int argc, char *argv[])
{
/* Create a datagram socket on which to send. */
sd = socket(AF_INET, SOCK_DGRAM, 0);
if(sd < 0)
{
perror("Opening datagram socket error");
```

```
exit(1);
}
else
printf("Opening the datagram socket...OK.\n");

/* Initialize the group sockaddr structure with a */
/* group address of 225.1.1.1 and port 5555. */
memset((char *) &groupSock, 0, sizeof(groupSock));
groupSock.sin_family = AF_INET;
groupSock.sin_addr.s_addr = inet_addr("226.1.1.1");
groupSock.sin_port = htons(4321);

/* Disable loopback so you do not receive your own datagrams.
{
char loopch = 0;
if(setsockopt(sd, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&loopch, sizeof(loopch)) < 0)
{
perror("Setting IP_MULTICAST_LOOP error");
close(sd);
exit(1);
}
else
printf("Disabling the loopback...OK.\n");
}
*/

/* Set local interface for outbound multicast datagrams. */
/* The IP address specified must be associated with a local, */
/* multicast capable interface. */
localInterface.s_addr = inet_addr("203.106.93.94");
if(setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF, (char *)&localInterface,
sizeof(localInterface)) < 0)
{
perror("Setting local interface error");
exit(1);
}
else
printf("Setting the local interface...OK\n");
/* Send a message to the multicast group specified by the*/
/* groupSock sockaddr structure. */
/*int datalen = 1024;*/
if(sendto(sd, databuf, datalen, 0, (struct sockaddr*)&groupSock, sizeof(groupSock)) < 0)
{perror("Sending datagram message error");}
else
printf("Sending datagram message...OK\n");

/* Try the re-read from the socket if the loopback is not disable
if(read(sd, databuf, datalen) < 0)
{
perror("Reading datagram message error\n");
close(sd);
exit(1);
}
else
{
printf("Reading datagram message from client...OK\n");
printf("The message is: %s\n", databuf);
}
*/
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g mcastserver.c -o mcastserver
```

- Before running this multicaster program, you have to run the client program first as in the following.

Example: Receiving a multicast datagram, a client

- The following example enables a socket to perform the steps listed below and to receive multicast datagrams:

    1. Create an `AF_INET`, `SOCK_DGRAM` type socket.
    2. Set the `SO_REUSEADDR` option to allow multiple applications to receive datagrams that are destined to the same local port number.

3. Use the `bind()` verb to specify the local port number. Specify the IP address as `INADDR_ANY` in order to receive datagrams that are addressed to a multicast group.
4. Use the `IP_ADD_MEMBERSHIP` socket option to join the multicast group that receives the datagrams. When joining a group, specify the **class D** group address along with the IP address of a local interface. The system must call the `IP_ADD_MEMBERSHIP` socket option for each local interface receiving the multicast datagrams.
5. Receive the datagram.

```
/* Receiver/client multicast Datagram example. */
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

struct sockaddr_in localSock;
struct ip_mreq group;
int sd;
int datalen;
char databuf[1024];

int main(int argc, char *argv[])
{
/* Create a datagram socket on which to receive. */
sd = socket(AF_INET, SOCK_DGRAM, 0);
if(sd < 0)
{
perror("Opening datagram socket error");
exit(1);
}
else
printf("Opening datagram socket....OK.\n");

/* Enable SO_REUSEADDR to allow multiple instances of this */
/* application to receive copies of the multicast datagrams. */
{
int reuse = 1;
if(setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&reuse, sizeof(reuse)) < 0)
{
perror("Setting SO_REUSEADDR error");
close(sd);
exit(1);
}
else
printf("Setting SO_REUSEADDR...OK.\n");
}

/* Bind to the proper port number with the IP address */
/* specified as INADDR_ANY. */
memset((char *) &localSock, 0, sizeof(localSock));
localSock.sin_family = AF_INET;
localSock.sin_port = htons(4321);
localSock.sin_addr.s_addr = INADDR_ANY;
if(bind(sd, (struct sockaddr*)&localSock, sizeof(localSock)))
{
perror("Binding datagram socket error");
close(sd);
exit(1);
}
else
printf("Binding datagram socket...OK.\n");

/* Join the multicast group 226.1.1.1 on the local 203.106.93.94 */
/* interface. Note that this IP_ADD_MEMBERSHIP option must be */
/* called for each local interface over which the multicast */
/* datagrams are to be received. */
group.imr_multiaddr.s_addr = inet_addr("226.1.1.1");
group.imr_interface.s_addr = inet_addr("203.106.93.94");
if(setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&group, sizeof(group)) < 0)
{
perror("Adding multicast group error");
close(sd);
exit(1);
}
else
printf("Adding multicast group...OK.\n");
```

```
/* Read from the socket. */
datalen = sizeof(databuf);
if(read(sd, databuf, datalen) < 0)
{
perror("Reading datagram message error");
close(sd);
exit(1);
}
else
{
printf("Reading datagram message...OK.\n");
printf("The message from multicast server is: \"%s\"\n", databuf);
}
return 0;
}
```

- Compile and link.

```
[bodo@bakawali testsocket]$ gcc -g  mcastclient.c -o mcastclient
```

- Run the client program.

```
[bodo@bakawali testsocket]$ ./mcastclient
Opening datagram socket....OK.
Setting SO_REUSEADDR...OK.
Binding datagram socket...OK.
Adding multicast group...OK.
```

- Then run the server program.

```
[bodo@bakawali testsocket]$ ./mcastserver
Opening the datagram socket...OK.
Setting the local interface...OK
Sending datagram message...OK
[bodo@bakawali testsocket]$
```

- The messages on the client console are shown below.

```
[bodo@bakawali testsocket]$ ./mcastclient
Opening datagram socket....OK.
Setting SO_REUSEADDR...OK.
Binding datagram socket...OK.
Adding multicast group...OK.
Reading datagram message...OK.
The message from multicast server is: "Multicast test message lol!"
[bodo@bakawali testsocket]$
```

Continue on next Module…TCP/IP and RAW socket, more program examples.

----------End Part III----------

---www.tenouk.com---

Further reading and digging:

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.
2. Broadcasting.
3. Telephony.