NETWORK PROGRAMMING LINUX SOCKET PART 12: SERVER DESIGN

Working program examples if any compiled using gcc, tested using the public IPs, run on **Linux 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.

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

Example 5 Socket
Example 6 Socket
Example 7 Advanced TCP/IP 1
Advanced TCP/IP 2
Advanced TCP/IP 3
Advanced TCP/IP 4
Advanced TCP/IP 5

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.

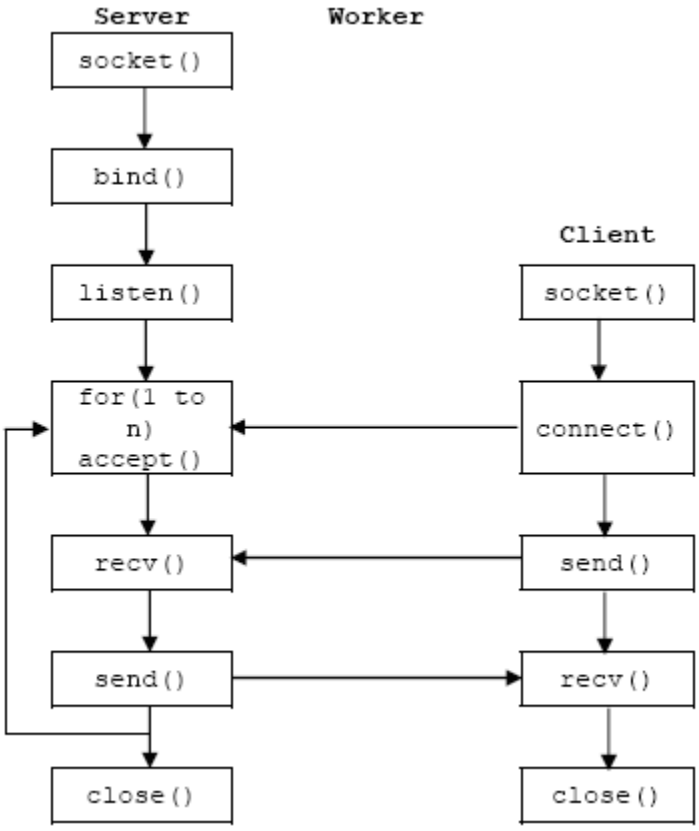

Figure 1: An example of socket APIs used for iterative server design.

- 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.
```

- You may want to change the gets() to the secure version, gets_s(). 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]$
```

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

Further reading and digging:

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.
2. Broadcasting.
3. Telephony.
4. GCC, GDB and other related tools.

| Winsock & .NET | Winsock | < TCP & UDP Working Program Examples | Linux Socket Index | Client-Server Multicast Example > |