| Winsock & .NET | Winsock | < TCP, UDP Program Examples |
Linux Socket Index | More TCP, UDP, Client-Server Examples > |

NETWORK PROGRAMMING SOCKET PART 9 - CLIENT & SERVER PROGRAM EXAMPLES

Menu

Network Story 1
Network Story 2
Network Story 3
Network Story 4
Network Story 5
Network Story 6
Socket Example 1
Socket Example 2
Socket Example 3
Socket Example 4
Socket Example 5
Socket Example 6
Socket

Working program examples if any compiled using gcc, tested using the public IPs, run on **Linux/Fedora Core 3**, with several times of update, as normal user. The Fedora machine used for the testing having the "No Stack Execute" disabled and the SELinux set to default configuration.

A Simple Stream Client Program Example

- This client will connect to the host that you specify in the command line, with port 3490. It will get the string that the previous server sends. The following is the source code.

```
/*** clientprog.c ****/
/*** a stream socket client demo ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

// the port client will be connecting to
#define PORT 3490
// max number of bytes we can get at once
```

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

```
#define MAXDATASIZE 300

int main(int argc, char *argv[])
{
int sockfd, numbytes;
char buf[MAXDATASIZE];
struct hostent *he;
// connector's address information
struct sockaddr_in their_addr;

// if no command line argument supplied
if(argc != 2)
{
    fprintf(stderr, "Client-Usage: %s
the_client_hostname\n", argv[0]);
    // just exit
    exit(1);
}

// get the host info
if((he=gethostbyname(argv[1])) == NULL)
{
    perror("gethostbyname()");
    exit(1);
}
else
    printf("Client-The remote host is:
%s\n", argv[1]);

if((sockfd = socket(AF_INET, SOCK_STREAM,
0)) == -1)
{
    perror("socket()");
    exit(1);
}


else
    printf("Client-The socket() sockfd is OK...\n");

// host byte order
their_addr.sin_family = AF_INET;
// short, network byte order
printf("Server-Using %s and port %d...\n", argv[1],
PORT);
their_addr.sin_port = htons(PORT);
their_addr.sin_addr = *((struct in_addr
*)he->h_addr);
```

```
// zero the rest of the struct
memset(&(their_addr.sin_zero), '\0', 8);

if(connect(sockfd, (struct sockaddr *)&their_addr,
sizeof(struct sockaddr)) == -1)
{
    perror("connect()");
    exit(1);
}
else
    printf("Client-The connect() is OK...\n");

if((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0))
== -1)
{
    perror("recv()");
    exit(1);
}
else
    printf("Client-The recv() is OK...\n");

buf[numbytes] = '\0';
printf("Client-Received: %s", buf);

printf("Client-Closing sockfd\n");
close(sockfd);
return 0;
}
```

- Compile and link the program.

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

- Run the program without argument.

```
[bodo@bakawali testsocket]$ ./clientprog
Client-Usage: ./clientprog the_client_hostname
```

- Run the program with server IP address or name as an argument. Here we use IP address.
- Make sure your previous serverprog program is running. We will connect using the same server. You can try running the server and client program at different machines.

```
[bodo@bakawali testsocket]$ ./clientprog
203.106.93.94
...
```

```
[bodo@bakawali testsocket]$ ./clientprog bakawali
Client-The remote host is: bakawali
Client-The socket() sockfd is OK...
Server-Using bakawali and port 3490...
Client-The connect() is OK...
Client-The recv() is OK...
Client-Received: This is the test string from server!
Client-Closing sockfd
```

- Verify the connection.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3490
tcp        0      0 *:3490
*:*                   LISTEN
tcp        0      0 bakawali.jmti.gov.my:3490
bakawali.jmti.gov.my:1358   TIME_WAIT
[bodo@bakawali testsocket]$
```

- At server's console, we have the following messages.

```
[bodo@bakawali testsocket]$ ./serverprog
Server-socket() sockfd is OK...
Server-setsockopt() is OK...
Server-Using 0.0.0.0 and port 3490...
Server-bind() is OK...
Server-listen() is OK...Listening...
Server-sigaction() is OK...
Server-accept() is OK...
Server-new socket, new_fd is OK...
Server: Got connection from 203.106.93.94
Server-send() is OK...!
Server-new socket, new_fd closed successfully...
```

- Well, our server and client programs work! Here we run the server program and let it listens for connection. Then we run the client program. They got connected!
- Notice that if you don't run the server before you run the client, connect() returns "Connection refused" message as shown below.

```
[bodo@bakawali testsocket]$ ./clientprog bakawali
Client-The remote host is: bakawali
Client-The socket() sockfd is OK...
Server-Using bakawali and port 3490...
connect: Connection refused
```

Datagram Sockets: The Connectionless

- The following program examples use the UDP, the connectionless datagram.

The senderprog.c (client) is sending a message to receiverprog.c (server) that acts as listener.

```
/*receiverprog.c - a server, datagram sockets*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
/* the port users will be connecting to */
#define MYPORT 4950
#define MAXBUFLEN 500

int main(int argc, char *argv[])
{
int sockfd;
/* my address information */
struct sockaddr_in my_addr;
/* connector's address information */
struct sockaddr_in their_addr;
int addr_len, numbytes;
char buf[MAXBUFLEN];

if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
    perror("Server-socket() sockfd error lol!");
    exit(1);
}
else
    printf("Server-socket() sockfd is OK...\n");

/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
/* automatically fill with my IP */
my_addr.sin_addr.s_addr = INADDR_ANY;
/* zero the rest of the struct */
memset(&(my_addr.sin_zero), '\0', 8);

if(bind(sockfd, (struct sockaddr *)&my_addr,
sizeof(struct sockaddr)) == -1)
{
```

```
    perror("Server-bind() error lol!");
    exit(1);
}
else
    printf("Server-bind() is OK...\n");

addr_len = sizeof(struct sockaddr);

if((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1, 0,
(struct sockaddr *)&their_addr, &addr_len)) == -1)
{
    perror("Server-recvfrom() error lol!");
    /*If something wrong, just exit lol...*/
    exit(1);
}
else
{
    printf("Server-Waiting and listening...\n");
    printf("Server-recvfrom() is OK...\n");
}

printf("Server-Got packet from %s\n",
inet_ntoa(their_addr.sin_addr));
printf("Server-Packet is %d bytes long\n", numbytes);
buf[numbytes] = '\0';
printf("Server-Packet contains \"%s\"\n", buf);

if(close(sockfd) != 0)
    printf("Server-sockfd closing failed!\n");
else
    printf("Server-sockfd successfully closed!\n");
return 0;
}
```

- Compile and link the program.

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

- Run the program, and then verify that it is running in background, start listening, waiting for connection.

```
[bodo@bakawali testsocket]$ ./receiverprog
Server-socket() sockfd is OK...
Server-bind() is OK...

[1]+  Stopped                 ./receiverprog
```

```
[bodo@bakawali testsocket]$ bg
[1]+ ./receiverprog &
[bodo@bakawali testsocket]$ netstat -a | grep 4950
udp        0      0 *:4950                  *:*
[bodo@bakawali testsocket]$
```

- This is UDP server, trying telnet to this server will fail because telnet uses TCP instead of UDP.

```
[bodo@bakawali testsocket]$ telnet   203.106.93.94
4950
Trying 203.106.93.94...
telnet: connect to address 203.106.93.94: Connection
refused
telnet: Unable to connect to remote host: Connection
refused
[bodo@bakawali testsocket]$
```

- Notice that in our call to socket() we're using SOCK_DGRAM. Also, note that there's no need to listen() or accept(). The following is the source code for senderprog.c (the client).

```
/*senderprog.c - a client, datagram*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
/* the port users will be connecting to */
#define MYPORT 4950

int main(int argc, char *argv[ ])
{
int sockfd;
/* connector's address information */
struct sockaddr_in their_addr;
struct hostent *he;
int numbytes;

if (argc != 3)
{
```

```
fprintf(stderr, "Client-Usage: %s <hostname>
<message>\n", argv[0]);
exit(1);
}
/* get the host info */
if ((he = gethostbyname(argv[1])) == NULL)
{
perror("Client-gethostbyname() error lol!");
exit(1);
}
else
printf("Client-gethostname() is OK...\n");

if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
perror("Client-socket() error lol!");
exit(1);
}
else
printf("Client-socket() sockfd is OK...\n");

/* host byte order */
their_addr.sin_family = AF_INET;
/* short, network byte order */
printf("Using port: 4950\n");
their_addr.sin_port = htons(MYPORT);
their_addr.sin_addr = *((struct in_addr
*)he->h_addr);
/* zero the rest of the struct */
memset(&(their_addr.sin_zero), '\0', 8);

if((numbytes = sendto(sockfd, argv[2],
strlen(argv[2]), 0, (struct sockaddr *)&their_addr,
sizeof(struct sockaddr))) == -1)
{
perror("Client-sendto() error lol!");
exit(1);
}
else
printf("Client-sendto() is OK...\n");

printf("sent %d bytes to %s\n", numbytes,
inet_ntoa(their_addr.sin_addr));

if (close(sockfd) != 0)
printf("Client-sockfd closing is failed!\n");
else
```

```
printf("Client-sockfd successfully closed!\n");
return 0;
}
```

- Compile and link the program.

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

- Run the program without arguments.

```
[bodo@bakawali testsocket]$ ./senderprog
Client-Usage: ./senderprog <hostname> <message>
```

- Run the program with arguments.

```
[bodo@bakawali testsocket]$ ./senderprog
203.106.93.94 "Testing UDP datagram message from
client"
Client-gethostname() is OK...
Client-socket() sockfd is OK...
Using port: 4950
Server-Waiting and listening...
Server-recvfrom() is OK...
Server-Got packet from 203.106.93.94
Server-Packet is 42 bytes long
Server-Packet contains "Testing UDP datagram message
from client"
Server-sockfd successfully closed!
Client-sendto() is OK...
sent 42 bytes to 203.106.93.94
Client-sockfd successfully closed!
[1]+  Done                    ./receiverprog
[bodo@bakawali testsocket]$
```

- Here, we test the UDP server and the client using the same machine. Make sure there is no restriction such as permission etc. for the user that run the programs.
- To make it really real, may be you can test these programs by running receiverprog on some machine, and then run senderprog on another. If there is no error, they should communicate.
- If senderprog calls connect() and specifies the receiverprog's address then the senderprog may only sent to and receive from the address specified by connect().
- For this reason, you don't have to use sendto() and recvfrom(); you can simply use send() and recv().

Blocking

- In a simple word 'block' means sleep but in a standby mode. You probably noticed that when you run **receiverprog**, previously, it just sits there until a packet arrives.
- What happened is that it called recvfrom(), there was no data, and so recvfrom() is said to "block" (that is, sleep there) until some data arrives. The socket functions which can block are:

1. `accept()`
2. `read()`
3. `readv()`
4. `recv()`
5. `recvfrom()`
6. `recvmsg()`
7. `send()`
8. `sendmsg()`
9. `sendto()`
10. `write()`
11. `writev()`

- The reason they can do this is because they're allowed to. When you first create the socket descriptor with socket(), the kernel sets it to blocking.
- If you don't want a socket to be blocking, you have to make a call to fcntl() something like the following:

```
#include <unistd.h>
#include <fcntl.h>
...
...
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
...
...
```

- By setting a socket to non-blocking, you can effectively 'poll' the socket for information. If you try to read from a non-blocking socket and there's no data there, it's not allowed to block, it will return -1 and errno will be set to EWOULDBLOCK.
- Generally speaking, however, this type of polling is a bad idea. If you put your program in a busy-wait looking for data on the socket, you'll suck up CPU time.
- A more elegant solution for checking to see if there's data waiting to be read comes in the following section on select().

Using select() for I/O multiplexing

- One traditional way to write network servers is to have the main server block on accept(), waiting for a connection. Once a connection comes in, the server forks, then the child process handles the connection and the main server is able to service new incoming requests.
- With select(), instead of having a process for each request, there is usually only one process that multiplexes all requests, servicing each request as much as it can.
- So one main advantage of using select() is that your server will only require a single process to handle all requests. Thus, your server will not need shared memory or synchronization primitives for different tasks to communicate.
- As discussed before we can use the non-blocking sockets' functions but it is CPU intensive.
- One major disadvantage of using select(), is that your server cannot act like there's only one client, like with a forking solution. For example, with a forking solution, after the server forks, the child process works with the client as if there was only one client in the universe, the child does not have to worry about new incoming connections or the existence of other sockets.
- With select(), the programming isn't as transparent. The prototype is as the following:

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

int select(int numfds, fd_set *readfds,
fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

- The function monitors "sets" of file descriptors; in particular readfds, writefds, and exceptfds. If you want to see if you can read from standard input and some socket descriptor, sockfd, just add the file descriptors 0 and sockfd to the set readfds.
- The parameter numfds should be set to the values of the highest file descriptor plus one. In this example, it should be set to sockfd+1, since it is assuredly higher than standard input that is 0.

- When select() returns, readfds will be modified to reflect which of the file descriptors you have selected which is ready for reading. You can test them with the macro FD_ISSET() listed below.
- Let see how to manipulate these sets. Each set is of the type fd_set. The following macros operate on this type:

1. `FD_ZERO(fd_set *set)` – clears a file descriptor set.
2. `FD_SET(int fd, fd_set *set)` – adds fd to the set.
3. `FD_CLR(int fd, fd_set *set)` – removes fd from the set.
4. `FD_ISSET(int fd, fd_set *set)` – tests to see if fd is in the set.

- select() works by blocking until something happens on a file descriptor/socket. The 'something' is the data coming in or being able to write to a file descriptor, you tell select() what you want to be woken up by. How do you tell it? You fill up an fd_set structure with some macros.
- Most select() based servers look quite similar:

1. Fill up an fd_set structure with the file descriptors you want to know when data comes in on.
2. Fill up an fd_set structure with the file descriptors you want to know when you can write on.
3. Call select() and block until something happens.
4. Once select() returns, check to see if any of your file descriptors was the reason you woke up. If so, 'service' that file descriptor in whatever particular way your server needs to (i.e. read in a request for a Web page).
5. Repeat this process forever.

- Sometimes you don't want to wait forever for someone to send you some data. Maybe every 60 seconds you want to print something like "Processing..." to the terminal even though nothing has happened.
- The timeval structure allows you to specify a timeout period. If the time is exceeded and select() still hasn't found any ready file descriptors, it'll return, so you can continue processing.
- The struct timeval has the following fields:

```
struct timeval
{
    int tv_sec;   /* seconds */
    int tv_usec; /* microseconds */
};
```

- Just set tv_sec to the number of seconds to wait, and set tv_usec to the number of microseconds to wait. There are 1,000,000 microseconds in a second. Also, when the function returns, timeout might be updated to show the time still remaining.

- Standard UNIX time slice is around 100 milliseconds, so you might have to wait that long no matter how small you set your struct timeval.
- If you set the fields in your struct timeval to 0, select() will timeout immediately, effectively polling all the file descriptors in your sets. If you set the parameter timeout to NULL, it will never timeout, and will wait until the first file descriptor is ready.
- Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to select().
- The following code snippet waits 5.8 seconds for something to appear on standard input.

```
/*selectcp.c - a select() demo*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
/* file descriptor for standard input */
#define STDIN 0

int main(int argc, char *argv[ ])
{
struct timeval tval;
fd_set readfds;
tval.tv_sec = 5;
tval.tv_usec = 800000;

FD_ZERO(&readfds);
FD_SET(STDIN, &readfds);
/* don't care about writefds and exceptfds: */
select(STDIN+1, &readfds, NULL, NULL, &tval);
if (FD_ISSET(STDIN, &readfds))
  printf("A key was pressed lor!\n");
else
  printf("Timed out lor!...\n");
return 0;
}
```

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

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

- Run the program and then press k.

```
[bodo@bakawali testsocket]$ ./selectcp
k
A key was pressed lor!
```

- Run the program and just leave it.

```
[bodo@bakawali testsocket]$ ./selectcp
Timed out lor!...
```

- If you're on a line buffered terminal, the key you hit should be RETURN or it will time out anyway.
- Now, some of you might think this is a great way to wait for data on a datagram socket and you are right: it might be. Some Unices can use select() in this manner, and some can't. You should see what your local man page says on the matter if you want to attempt it.
- Some Unices update the time in your struct timeval to reflect the amount of time still remaining before a timeout. But others do not. Don't rely on that occurring if you want to be portable. Use gettimeofday() if you need to track time elapsed.
- When a socket in the read set closes the connection, select() returns with that socket descriptor set as "ready to read". When you actually do recv() from it, recv() will return 0. That's how you know the client has closed the connection.
- If you have a socket that is listen()ing, you can check to see if there is a new connection by putting that socket's file descriptor in the readfds set.

Continue on next Module…More…

Further reading and digging:

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

| Winsock & .NET | Winsock | < TCP, UDP Program Examples |
Linux Socket Index | More TCP, UDP, Client-Server Examples > |