| Winsock & .NET | Winsock | < More Socket APIs & C Code Snippets | Linux Socket Index | Client-Server Program Examples > |

NETWORK PROGRAMMING LINUX SOCKET PART 8 - TCP & UDP PROGRAM EXAMPLES

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.

Some Idea In Managing Server Concurrency

Concurrency vs Iteration

Making the decision

- Program design is vastly different.
- Programmer needs to decide early.
- Network and computer speeds also keep changing.
- Optimality is a moving target.
- Programmer must use insight based on experience to decide which is better.

Level of Concurrency

- Number of concurrent clients.
- Iterative means 1 client at a time.
- Unbounded concurrency allows flexibility.
- TCP software limits the number of connections.
- OS limits each process to a fixed number of open files.
- OS limits the number of processes.

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

Problems with Unbounded Concurrency

- OS can run out of resources such as memory, processes, sockets, buffers causing blocking, thrashing, crashing...
- Demand for one service can inhibit others e.g. web server may prevent other use.
- Over-use can limit performance e.g. ftp server could be so slow that clients cancel requests wasting time spent doing a partial transfer.

Cost of Concurrency

- Assuming a forking concurrent server, each connection requires time for a process creation (c).
- Each connection also requires some time for processing requests (p).
- Consider 2 requests arriving at the same time.
- Iterative server completes both at time 2p.
- Concurrent server completes both perhaps at time 2c+p.
- If p < 2c the iterative server is faster.
- The situation can get worse with more requests.
- The number of active processes can exceed the CPU capacity.
- Servers with heavy loads generally try to dodge the process creation cost.

Process Pre-allocation to Limit Delays

- Master server process forks n times.
- The n slaves handle up to n clients.
- Operates like n iterative servers.
- Due to child processes inheriting the parent's passive socket, the slaves can all wait in accept on the same socket.
- For UDP, the slaves can all call recvfrom on the same socket.
- To avoid problems like memory leaks, the slaves can be periodically replaced.
- For UDP, bursts can overflow buffers causing data loss. Pre-allocation can limit this problem.

Dynamic Pre-allocation

- Pre-allocation can cause extra processing time if many slaves are all waiting on the same socket.
- If the server is busy, it can be better to have many slaves pre-allocated.
- If the server is idle, it can be better to have very few slaves pre-allocated.
- Some servers (Apache) adjust the level of concurrency according to service demand.

Delayed Allocation

- Rather than immediately forking, the master can quickly examine a request.
- It may be faster for some requests to handle them in the master rather than forking.
- Longer requests may be more appropriate to handle in a child process.
- If it is hard to quickly estimate processing time, the server can set a timer to expire after a small time and then fork to let a child finish the request.

Client Concurrency

- Shorter Response Time.
- Increased Throughput.
- Concurrency Allows Better Control.
- Communicating with Multiple Servers.
- Achieving Concurrency with a Single Client Process.

--The Linux Socket Program Examples Sections--

DNS

- DNS stands for "Domain Name System" (for Windows implementation it is called Domain Name Service). For socket it has three major components:

    1. Domain name space and resource records: Specifications for a tree-structured name space and the data associated with the names.
    2. Name servers: Server programs that hold information about the domain tree structure and that set information.
    3. Resolvers: Programs that extract information from name servers in response to client requests.

- DNS used to translate the IP address to domain name and vice versa. This way, when someone enters:

```
telnet      serv.google.com
```

- telnet can find out that it needs to connect() to let say, "198.137.240.92". To get these information we can use gethostbyname():

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

- As you see, it returns a pointer to a struct hostent, and struct hostent is shown below:

```
struct hostent
{
    char  *h_name;
    char  **h_aliases;
    int   h_addrtype;
    int   h_length;
    char  **h_addr_list;
};

#define h_addr h_addr_list[0]
```

- And the descriptions:

Member	Description
h_name	Official name of the host.
h_aliases	A NULL-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; usually AF_INET.
h_length	The length of the address in bytes.
h_addr_list	A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order.
h_addr	The first address in h_addr_list.

Table 40.1

- gethostbyname() returns a pointer to the filled struct hostent, or NULL on error but errno is not set, h_errno is set instead.
- As said before in implementation we use Domain Name Service in Windows and BIND in Unix/Linux. Here, we configure the Forward Lookup Zone for name to IP resolution and Reverse Lookup Zone for the reverse.
- The following is a program example using the gethostname().

```
/*****getipaddr.c ******/
/****a hostname lookup program example******/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
```

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

int main(int argc, char *argv[ ])
{
struct hostent *h;

/* error check the command line */
if(argc != 2)
{
    fprintf(stderr, "Usage: %s <domain_name>\n",
argv[0]);
    exit(1);
}

/* get the host info */
if((h=gethostbyname(argv[1])) == NULL)
{
    herror("gethostbyname(): ");
    exit(1);
}
else
    printf("gethostbyname() is OK.\n");

printf("The host name is: %s\n", h->h_name);
printf("The IP Address is: %s\n", inet_ntoa(*((struct
in_addr *)h->h_addr)));
printf("The address length is: %d\n", h->h_length);

printf("Sniffing other
names...sniff...sniff...sniff...\n");
int j = 0;
do
{
    printf("An alias #%d is: %s\n", j,
h->h_aliases[j]);
    j++;
}
while(h->h_aliases[j] != NULL);

printf("Sniffing other
IPs...sniff....sniff...sniff...\n");
int i = 0;
do
{
    printf("Address #%i is: %s\n", i, inet_ntoa(*
```

```
((struct in_addr *)(h->h_addr_list[i])));
    i++;
}
while(h->h_addr_list[i] != NULL);
return 0;
}
```

- Compile and link the program.

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

- Run the program. Because of the server used in this testing is using public IP address, we can test it querying the public domain such as www.yahoo.com.

```
[bodo@bakawali testsocket]$ ./getipaddr   www.yahoo.com
The host name is: www.yahoo.akadns.net
The IP Address is: 66.94.230.50
The address length is: 4
Sniffing other names...sniff...sniff...sniff...
An alias #0 is: www.yahoo.com
Sniffing other IPs...sniff....sniff...sniff...
Address #0 is: 66.94.230.50
Address #1 is: 66.94.230.36
Address #2 is: 66.94.230.41
Address #3 is: 66.94.230.34
Address #4 is: 66.94.230.47
Address #5 is: 66.94.230.32
Address #6 is: 66.94.230.35
Address #7 is: 66.94.230.45
```

- Again, running the program testing another domain.

```
[bodo@bakawali testsocket]$ ./getipaddr
www.google.com
The host name is: www.l.google.com
The IP Address is: 66.102.7.104
The address length is: 4
Sniffing other names...sniff...sniff...sniff...
An alias #0 is: www.google.com
Sniffing other IPs...sniff....sniff...sniff...
Address #0 is: 66.102.7.104
Address #1 is: 66.102.7.99
Address #2 is: 66.102.7.147
[bodo@bakawali testsocket]$
```

- With gethostbyname(), you can't use perror() to print error message since errno is not used instead, call herror().

- You simply pass the string that contains the machine name ("www.google.com") to gethostbyname(), and then grab the information out of the returned struct hostent.
- The only possible weirdness might be in the printing of the IP address. Here, h->h_addr is a char*, but inet_ntoa() wants a struct in_addr passed to it. So we need to cast h->h_addr to a struct in_addr*, then dereference it to get the data.

Some Client-Server Background

- Just about everything on the network deals with client processes talking to server processes and vice-versa. For example take a telnet.
- When you telnet to a remote host on port 23 at client, a program on that server normally called telnetd (telnet daemon), will respond. It handles the incoming telnet connection, sets you up with a login prompt, etc. In Windows this daemon normally called a service. The daemon or service must be running in order to do the communication.
- Note that the client-server pair can communicate using SOCK_STREAM, SOCK_DGRAM, or anything else (as long as they're using the same protocol). Some good examples of client-server pairs are telnet/telnetd, ftp/ftpd, or bootp/bootpd. Every time you use ftp, there's a remote program, ftpd that will serve you.
- Often, there will only be one server, and that server will handle multiple clients using fork() etc. The basic routine is: server will wait for a connection, accept() it and fork() a child process to handle it. The following program example is what our sample server does.

A Simple Stream Server Program Example

- What this server does is send the string "This is a test string from server!" out over a stream connection.
- To test this server, run it in one window and telnet to it from another window or run it in a server and telnet to it from another machine with the following command.

```
telnet     the_remote_hostname     3490
```

- Where the_remote_hostname is the name of the machine you're running it on. The following is the server source code:

```
/* serverprog.c - a stream socket server demo */
#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 <sys/wait.h>
#include <signal.h>

/* the port users will be connecting to */
#define MYPORT 3490
/* how many pending connections queue will hold */
#define BACKLOG 10

void sigchld_handler(int s)
{
    while(wait(NULL) > 0);
}

int main(int argc, char *argv[ ])
{
/* listen on sock_fd, new connection on new_fd */
int sockfd, new_fd;
/* my address information */
struct sockaddr_in my_addr;
/* connector's address information */
struct sockaddr_in their_addr;
int sin_size;
struct sigaction sa;
int yes = 1;

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

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)) == -1)
{
    perror("Server-setsockopt() error lol!");
    exit(1);
}
else
    printf("Server-setsockopt 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;
```

```
printf("Server-Using %s and port %d...\n",
inet_ntoa(my_addr.sin_addr), MYPORT);

/* 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");
    exit(1);
}
else
    printf("Server-bind() is OK...\n");

if(listen(sockfd, BACKLOG) == -1)
{
    perror("Server-listen() error");
    exit(1);
}
printf("Server-listen() is OK...Listening...\n");

/* clean all the dead processes */
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;

if(sigaction(SIGCHLD, &sa, NULL) == -1)
{
    perror("Server-sigaction() error");
    exit(1);
}
else
    printf("Server-sigaction() is OK...\n");

/* accept() loop */
while(1)
{
sin_size = sizeof(struct sockaddr_in);
if((new_fd = accept(sockfd, (struct sockaddr
*)&their_addr, &sin_size)) == -1)
{
    perror("Server-accept() error");
    continue;
}
else
    printf("Server-accept() is OK...\n");
printf("Server-new socket, new_fd is OK...\n");
```

```
printf("Server: Got connection from %s\n",
inet_ntoa(their_addr.sin_addr));

/* this is the child process */
if(!fork())
{
    /* child doesn't need the listener */
   close(sockfd);

   if(send(new_fd, "This is a test string from
server!\n", 37, 0) == -1)
        perror("Server-send() error lol!");
   close(new_fd);
   exit(0);
}
else
    printf("Server-send is OK...!\n");

/* parent doesn't need this*/
close(new_fd);
printf("Server-new socket, new_fd closed
successfully...\n");
}
return 0;
}
```

- Compile and link the program.

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

- Run the program.

```
[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...

[1]+  Stopped                 ./serverprog
[bodo@bakawali testsocket]$
```

- Verify that the program is running in the background. You may do this from another terminal.

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

- Verify that the program/process is listening on the specified port, waiting for connection.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3490
tcp        0      0 *:3490       *:*
LISTEN
```

- Again, verify that the program/process is listening, waiting for connection.

```
[bodo@bakawali testsocket]$ ps aux | grep serverprog
bodo      2586  0.0  0.2  2940  296 pts/3    S
14:04   0:00 ./serverprog
bodo      2590  0.0  0.5  5432  660 pts/3    R+
14:04   0:00 grep serverprog
```

- Then, trying the telnet. Open another terminal, telnet itself with the specified port number. Here we use the server name, bakawali. When the string is displayed press the Escape character Ctrl+] (^]). Then we have a real telnet session.

```
[bodo@bakawali testsocket]$ telnet  bakawali  3490
Trying 203.106.93.94...
Connected to bakawali.jmti.gov.my (203.106.93.94).
Escape character is '^]'.
This is the test string from server!
^]
telnet> ?
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 charaters
('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>
```

- Type quit to exit the session.

```
...
telnet> quit
Connection closed.
[bodo@bakawali ~]$
```

- If we do not stop the server program/process (Ctrl+Z), at the server terminal the following messages should be displayed. Press Enter (Carriage Return) key back to the prompt.

```
[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...
```

- To stop the process just issue a normal kill command. Before that verify again.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3490
tcp        0      0 *:3490        *:*
LISTEN

[bodo@bakawali testsocket]$ ps aux | grep ./serverprog
bodo  3184  0.0  0.2  1384  324 pts/3  S   23:46
0:00 ./serverprog
bodo  3188  0.0  0.5  3720  652 pts/3  R+  23:48
0:00 grep ./serverprog
```

```
[bodo@bakawali testsocket]$ kill -9 3184
[bodo@bakawali testsocket]$ netstat -a | grep 3490
[1]+  Killed                  ./serverprog

[bodo@bakawali testsocket]$
```

- The server program seems OK. Next section is a client program, **clientprog.c** that we will use to test our server program, **serverprog.c**.
- The sigaction() code is responsible for cleaning the zombie processes that appear as the forked child processes. You will get the message from this server by using the client program example presented in the next section.

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.