

## The Socket API, Part 3: Concurrent Servers

By [Pankaj Tanwar](#) on October 1, 2011 in [Coding](#), [Developers](#) · [2 Comments](#)

*In this part of the series, we will learn how to deal with multiple clients connected to the server.*



Welcome to another dose of socket programming! Till now, we've created servers that are capable of creating connections with multiple clients [[Part 1](#) & [Part 2](#)], but the problem is that the server will communicate with only one client at any point in time. This is because there is only one socket descriptor, `cfd`, created to communicate with a client — and all connections will wait on the same descriptor. Now, let's use the `fork()` system call to fork a copy of the server for each client.

Here is the code included from the previous article. This time it is for IPv4...

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netinet/in.h>
6  #include <signal.h>
7
8  int main()
9  {
10     int sfd, cfd;
11     socklen_t len;
12     char ch, buff[INET_ADDRSTRLEN];
13     struct sockaddr_in saddr, caddr;
14
```

```

15     sfd= socket(AF_INET, SOCK_STREAM, 0);
16
17     saddr.sin_family=AF_INET;
18     saddr.sin_addr.s_addr=htonl(INADDR_ANY);
19     saddr.sin_port=htons(1205);
20
21     bind(sfd, (struct sockaddr *)&saddr, sizeof(saddr));
22
23     listen(sfd, 5);
24     signal(SIGCHLD, SIG_IGN);
25
26     while(1) {
27         printf("Server waiting\n");
28         len=sizeof(caddr);
29         cfd=accept(sfd, (struct sockaddr *)&caddr, &len);
30
31         if( fork() == 0) {
32             printf("Child Server Created Handling connection with %s\n",
33                 inet_ntop(AF_INET, &caddr.sin_addr, buff, sizeof(buff)));
34
35             close(sfd);
36
37             if(read(cfd, &ch, 1)<0) perror("read");
38
39             while( ch != EOF) {
40                 if((ch>='a' && ch<='z') || (ch>='A' && ch<='Z'))
41                     ch^=0x20;
42                 /* EXORing 6th bit will result in change in case */
43
44                 if(write(cfd, &ch, 1)<0) perror("write");
45
46                 if(read(cfd, &ch, 1)<0) perror("read");
47             }
48             close(cfd);
49             return 0;
50         }
51
52         close(cfd);
53     }
54 }
55

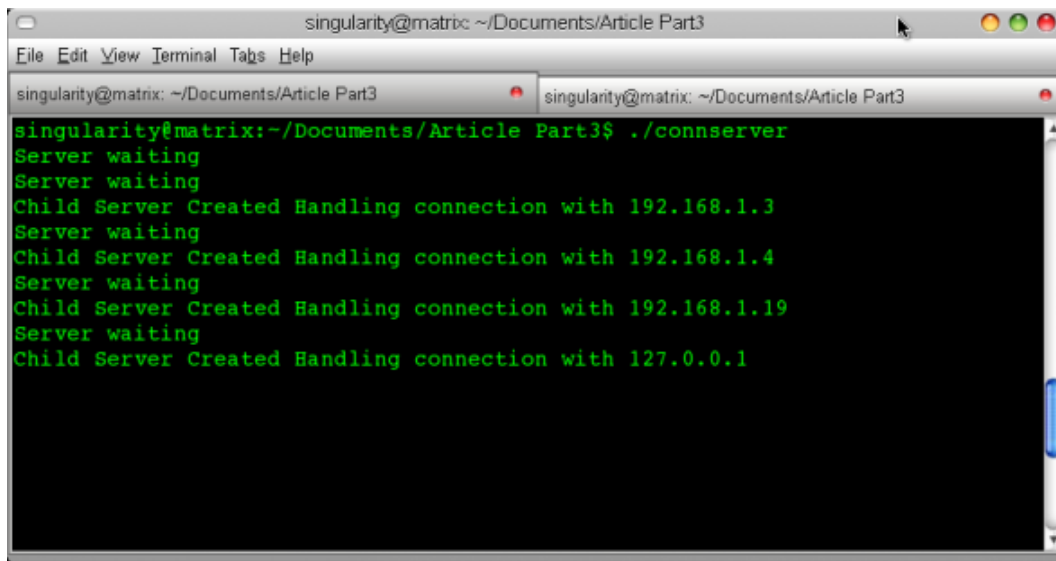
```

Let's see what we've done here. At line 23, after getting the socket descriptor `cfd` from the call to `accept`, we forked the server. The child process (where `pid==0`) closes the listening descriptor with `close(sfd)`, does the work the server is intended to do, and when finished, closes the descriptor and returns (see lines 27-39).

The server, on the other hand, where `pid>0`, just closes the `cfd` (line 39), and is again ready to accept more connections. Thus, for each incoming connection, a new server process will be created.

Another method of doing this is using threads, but we're not getting into that right now.

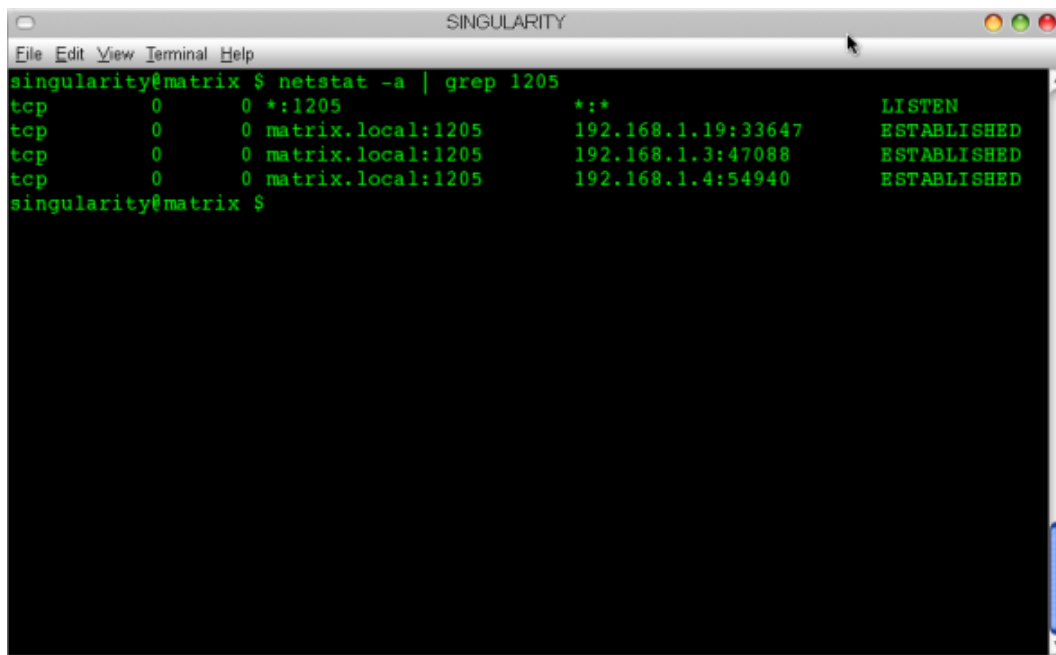
Now, compile, run and see the server running and handling multiple clients. See Figure 1 — I'm running the server on a network now (;) though these clients are VirtualBox running VMs with Backtrack (192.168.1.19) and Arch (192.168.1.4), and Android phone running ConnectBot to create a TCP connection.



```
singularity@matrix: ~/Documents/Article Part3
File Edit View Terminal Tabs Help
singularity@matrix: ~/Documents/Article Part3
singularity@matrix:~/Documents/Article Part3$ ./connserver
Server waiting
Server waiting
Child Server Created Handling connection with 192.168.1.3
Server waiting
Child Server Created Handling connection with 192.168.1.4
Server waiting
Child Server Created Handling connection with 192.168.1.19
Server waiting
Child Server Created Handling connection with 127.0.0.1
```

Figure 1: Server screen

Also run `netstat -a | grep 1205` to check for current network connections; I've greped 1205, our port, to show only connections to our server (see Figure 2).



```
SINGULARITY
File Edit View Terminal Help
singularity@matrix $ netstat -a | grep 1205
tcp        0      0 *:1205                *:*                    LISTEN
tcp        0      0 matrix.local:1205     192.168.1.19:33647     ESTABLISHED
tcp        0      0 matrix.local:1205     192.168.1.3:47088     ESTABLISHED
tcp        0      0 matrix.local:1205     192.168.1.4:54940     ESTABLISHED
singularity@matrix $
```

Figure 2: Output of netstat

We can also see the parent server process LISTENing and the ESTABLISHED connections, with IPs and ports.

We added `signal(SIGCHLD, SIG_IGN)` to the code to prevent child processes going into a zombie state. A child process, when it finishes (a clean termination or killed by some signal), returns the

exit status to its parent process, which is notified by the `SIGCHLD` signal sent by the system. If the parent doesn't handle this signal, the child process still uses some memory, and remains in a zombie state. If the parent finishes before the child, or doesn't collect the status and terminates, the status will be provided to the parent of all processes, i.e., `init` with `pid 1`.

Let's examine some code from the `signal()` man page:

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

`signal()` sets the disposition of the signal `signum` to handler, which is either `SIG_IGN`, `SIG_DFL`, or the address of a programmer-defined function (a "signal handler"). It also indicates that the behaviour of `signal()` differs among different Linux and UNIX versions, and tells us to use `sigaction()` instead.

The problem is whether the blocked system call, after running the signal handler, will be restarted or not. We can look at this later, when we write our own signal handler — or just go check the `sigaction` structure in the man pages.

Now, just receive the signal and ignore it by setting handler to `SIG_IGN`. This will not let the child enter a zombie state. What if the parent finishes before the child (though not the case here, because the server is in an infinite loop)? In that case, the parent can wait for the child using `wait()` or `waitpid()` calls, of which `waitpid()` is preferred.

These system calls wait for a state change in the child, such as the child being terminated or stopped by a signal, or resumed by a signal (check the man pages).

## A little science

Now, before moving further, let's talk about TCP here since better code requires a sound understanding of the basics.

In Figure 3, the rectangles represent the states, and the edges, the transitions.

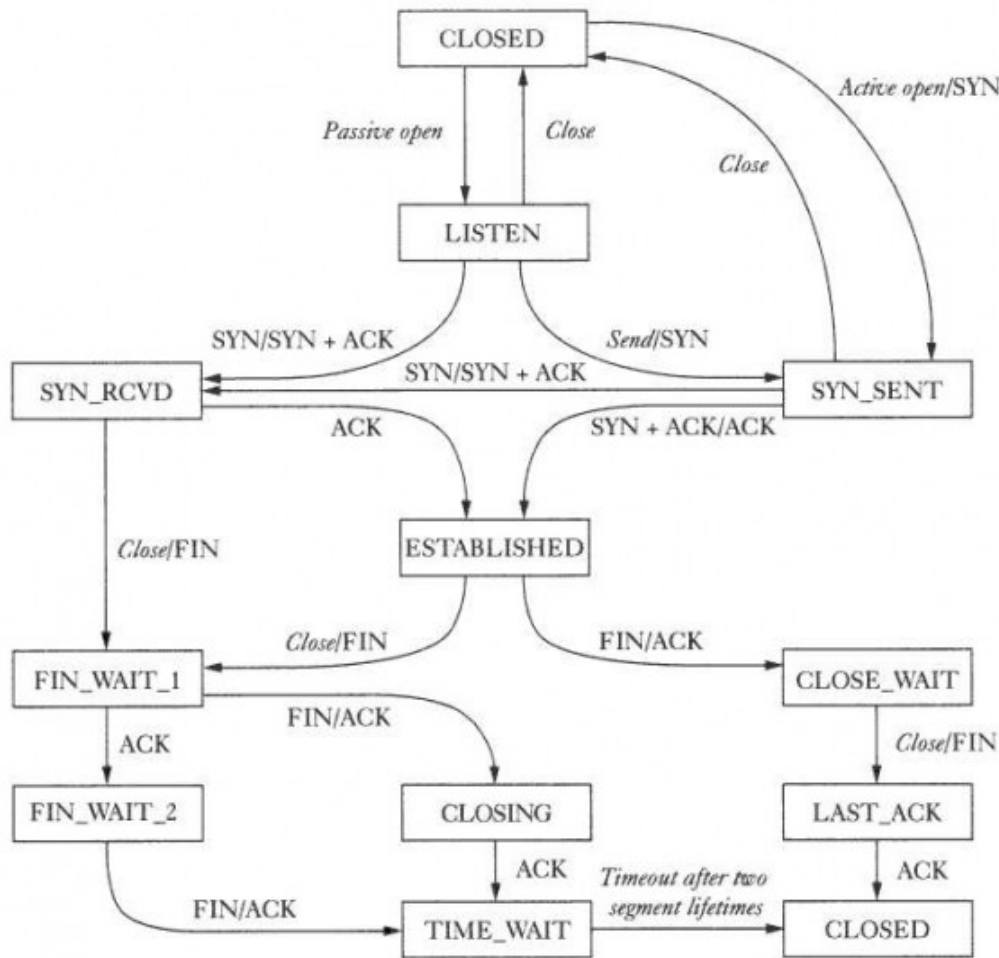


Figure 3: TCP state diagram

We need to visualise the server and client in the diagram. Two edges come out of the **CLOSED** state; one is Active Open. This transition occurs when a client sends a SYN packet to the server, and the system is initiating the connection. Another is Passive Open, where the server enters the listen state and waits for connections.

First, from the client side, after the SYN is sent the client enters the **SYN\_SENT** state. Now, after receiving the SYN from the server and sending SYN and ACK, it transitions to **ESTABLISHED**. From the **ESTABLISHED** state (where communication takes place), sending a FIN will initiate an Active Close to terminate the connection, and enter the **FIN\_WAIT\_1** state. Receiving ACK will move it to the **FIN\_WAIT\_2** state.

Receiving the FIN from the server will result in sending the ACK, and going to the **TIME\_WAIT** state. In this state, the system waits for twice the MSL (maximum segment lifetime); the recommended value in RFC 1337 is 120 seconds, Linux implements 60 seconds. This state helps reliable termination of connections in case of lost packets, and allows old duplicate segments to expire in

the network. Finally, it goes to the CLOSED state.

For the server, passive open is the LISTEN state. Receiving a SYN results in sending SYN and ACK, and going to the [SYN\\_RCVD](#) state. Receiving an ACK will take the server to the ESTABLISHED state for data communication. Then, receiving a FIN will result in sending an ACK, and will initiate the passive close and it going to the [CLOSE\\_WAIT](#) state.

After the operation completes, the server sends the FIN, and transitions to the [LAST\\_ACK](#) state. On receiving the ACK, it will terminate the connection and go to the CLOSED state.

Here we can see the “ThreeHandshake” — the exchange of three packets to establish a TCP connection. It is initiated when the client calls [connect\(\)](#). Packet 1 is SYN x from client to server; Packet 2 is ACK x+1 and SYN y from server to client; and Packet 3 is ACK y+1 from client to server. Here, x is the sequence number from the client, and y the sequence number from the server.

To terminate the connection, we need four packets. The client calls [close\(\)](#) to initiate termination: Packet 1 is FIN m from client to server; and Packet 2 is ACK m+1 from server to client. Now, the server finishes the operation, and then calls [close\(\)](#) and sends its FIN n. The client sends ACK n+1 to terminate the connection.

Here's where I close the connection, even though it was short this time! Next month, I'll be back with a [new connection to socket programming](#)... you can now ACK my FIN!

## Related Posts:

- [iLinuxBot: Designing Botnets to Manage Linux Clients](#)
- [Implementing HTML5 WebSockets Using Java](#)
- [Function Pointers and Callbacks in C — An Odyssey](#)
- [Getting Started with HTML5 WebSockets](#)
- [Use XMPP to Create Your Own Google Talk Client](#)

Tags: [ACK](#), [Android](#), [API](#), [cfd](#), [inter process communication](#), [IPC](#), [IPv4](#), [LFY October 2011](#), [Linux](#), [sfd](#), [Socket API](#), [Socket API series](#), [socket descriptor](#), [socket programming](#), [SYN](#), [system call](#), [unix](#), [unix socket](#), [UNIX Socket API](#), [VirtualBox](#), [VMs](#)

Article written by:



**Pankaj Tanwar**

The author is a FOSS enthusiast who loves rock music and C.

Connect with him: [Website](#)

[Previous Post](#)

[Next Post](#)

[Chkrootkit -- Eliminate the Enemy Within](#)[Backups and More with rsync](#)

ALSO ON LINUX FOR YOU

WHAT'S THIS?

[PHP Development: A Smart Career Move](#)

1 comment

[Cross-Platform: Fact or Fiction?](#)

1 comment

[Make Your Android Smartphone Work For You!](#)

5 comments

[Taming the Big Data Beast with Hadoop and Alternatives](#)

1 comment

2 Comments

LINUX For You

 Login Recommend Share


Sort by Newest



Join the discussion...

**kray257** • 5 months ago

"Let's see what we've done here. At line 23, after getting the socket descriptor cfd from the call to accept, we forked the server." -- You might want to proof read this... The call to accept() is on line 29...

 |  • Reply • Share >**Pankaj Tanwar**  kray257 • 5 months ago

Hello kray this article was published in "Linux for You" magazine and everything is according to the print edition. This electronic edition was later published by them and so has some problems here there.

I am sure you are figuring out and correcting them

Thanks and sorry for inconvenience


 |  • Reply • Share > Subscribe Add Disqus to your site Privacy

Search for:

Search

OPENING  
OPEN

- Open Source Con
- OSS
- Enterprise Supp
- Outsourced Prod
- Open Source Tra



UNOT  
Your path t

CLO



Call on our Tollfre

1-800-21

Get Connecte


RSS Feec

Facebook









Find us on Face


 Op


520,890 people





Facebook soc


Popular C

 Decer  
Pandu  
Settir  
Fun!


 Nover  
Crea

 Janua  
Getti

 Janua  
A Pe

 Janua  
Unloc

Reviews   How-Tos   Coding   Interviews   Features   Overview   Blogs



Search

Popular tags

For You & Me  
Developers  
Sysadmins  
Open Gurus

Linux, ubuntu, Java, Android, MySQL, Google, python, Fedora, PHP, C, open source, html, Microsoft, web applications, Windows, Security, India, programming, Apache, Red Hat, unix, operating systems, JavaScript, Oracle, RAM, xml, LFY April 2012, Developers, firewall, FOSS

CXOs  
Columns

All published articles are released under [Creative Commons Attribution-NonCommercial 3.0 Unported License](#), unless otherwise noted.  
[Open Source For You](#) is powered by [WordPress](#), which gladly sits on top of a [CentOS](#)-based [LEMP stack](#).

---

3