Australian Journal of Basic and Applied Sciences, 7(4): 398-408, 2013
ISSN 1991-8178

Review Socket Programming

Behzad Mahjour Shafiei

Ashtian Branch, Islamic Azad University, Ashtian, Iran

Abstract : This document describes the syntax of the TCP/IP application programming interface (API). The APIs that are described in this document can be used to create TCP/IP client and server applications or modify existing applications to communicate using TCP/IP. The information in this document supports both IPv6 and IPv4. Unless explicitly noted, information describes IPv4 networking protocol. IPv6 support is qualified within the text. To provide flexibility in writing new applications and adapting existing applications, the following programming languages and interfaces are described:

- C sockets
- X/Open Transport Interface (XTI)
- Assembler, PL/I, and COBOL sockets
- REXX™ sockets
- Pascal language

Key words: Socket, client, server, port

INTRODUCTION

Sockets are interfaces that can "plug into" each other over a network. Once so "plugged in", the programs so connected communicate. This article discusses only simple aspects of stream inet sockets (don't worry about exactly what that is right now). For the purposes of this article, a "server" program is exposed via a socket connected to a certain /etc/services port number. A "client" program can then connect its own socket to the server's socket, at which time the client program's writes to the socket are read as stdin to the server program, and stdout from the server program are read from the client's socket reads. This is one subset of socket programming, but it's perhaps the easiest to master, so this is where you should start.

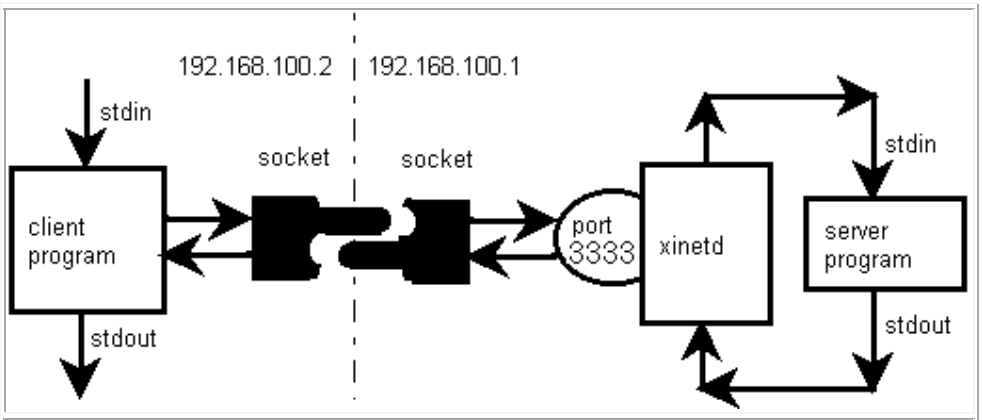

Diagram of client-server socket connection via xinetd. Note that the client communicates by reading and writing the socket, but the server program communicates via stdin and stdout.

This tutorial requires a Linux box. It hasn't been tested on other types of UNIX, but I think it might work. This tutorial is centered around a system using xinetd, but it would be simple enough to adapt it to older inetd systems. This tutorial will not work under Windows. I think it's important that this complex type of programming be learned on the most reliable, straightforward system possible, so Windows is out.

For the purposes of this tutorial, the server application will be at port 3333. Note that you can implement both the client and the server on a single computer, in which case the client is connected to a port on the computer containing both the client and the server. So if you have only one Linux box you can still do this tutorial.

Corresponding Author: Behzad Mahjour Shafiei, Ashtian Branch, Islamic Azad University, Ashtian, Iran.
Research.behzad@yahoo.com

ACKNOWLEDGMENTS

This material was told to me by LEAP-CF members Mark Alexander and NickolaiZeldovich. Other LEAPsters provided additional information.

A big shout out to all my buddies at Linux Enthusiasts and Professionals of Central Florida (LEAP-CF). With mailing list and meetings, they (we) form a 70 brain parallel supercomputer.

And of course, thanks to the help and support of Troubleshooters.Com's visitors.

Hello World:

Our first socket implementation uses a simple write-only shellscript as a server application, and telnet as the client. This is the simplest possible implementation. I've tested this on a mandrake box, and I'd imagine it would work on any Linux box using xinetd. If your box still uses the older inetd system, the only difference is that you modify inetd.conf instead of a file in /etc/xinetd.d, and you restart inetd instead of xinetd. The following steps are involved:

1. Create the server application (a simple shellscript)
2. Test the server application at the command line
3. Decide on a port number and service name
4. Declare the port and name in /etc/services
5. Create a file for this service in /etc/xinetd.d
6. Restart xinetd
7. Telnet into the service, and see the server app's output
8. Review what you learned

Create the Server Application (A Simple Shellscript):

We want the simplest possible script for this Hello World, so here it is:

```
#!/bin/bash
/bin/echo -n "Hello World:" | /usr/bin/tee /tmp/log.log
/bin/date | /usr/bin/tee -a /tmp/log.log
```

Save the preceding script as hello.sh, in your home directory (we'll assume your user id is myuid).

Note several things. The top line, showing the program to read the script, is ABSOLUTELY ESSENTIAL. Although most environments run shellscripts without that top line, those same scripts will not output when called through sockets as certain users (user nobody, for instance). For a similar reason, make sure every executable is declared with a complete path, and is not a link. With a user like user nobody, you don't know what the path will look like. Similarly, the log file is completely specified so you know where to find it. If you don't specifiy it, it goes in the home directory of the user hooked to the service by xinetd. Better to specify it so you know exactly where to find it./tmp is an excellent place, because on most systems it has universal read and write permissions.

This script writes a single line containing "Hello World:" followed by a timestamp, both to stdout and to a log file called /tmp/log.log. The log file will become essential in socket debugging, because when the program runs under a socket the program has no terminal.

Test the Server Application at the Command Line:

Run the program, and it should output the hello world phrase followed by a timestamp. When you look at /tmp/log.log it should contain the same information. See the session below:

```
[myuid@mydeskmyuid]$ ./hello.sh
Hello World:Tue Jan 23 12:27:08 EST 2001
[myuid@mydeskmyuid]$ cat /tmp/log.log
Hello World:Tue Jan 23 12:27:08 EST 2001
[myuid@mydeskmyuid]$
```

If you don't get the results shown above, troubleshoot until you do.

Decide on a Port Number and Service Name:

First, the port number needs to be unused. The port number you choose should not exist in /etc/services. Furthermore, for security reasons the port number should not be below 1024. For this exercise I use port 3333.

You may wish to use a different port so someone knowing you've taken this tutorial doesn't know of an open port on your system. But in this tutorial I'll use 3333.

```
[myuid@mydeskmyuid]$ cat /etc/services | grep 3333
[myuid@mydeskmyuid]$
```

The first thing to do is make sure the port number is not contained in the /etc/services file. Do that as follows:

If there's no output, there's no such port number. If there's output, investigate thoroughly, and you'll probably want to select a different number.

You also need to check for the existence of anxinetd declared service with a declared port number matching the new one. Do that with a directory wide grep:

```
[myuid@mydeskmyuid]$ grep -r 3333 /etc/xinetd.d/*
[myuid@mydeskmyuid]$
```

Once again, no output means no such port is declared. If there is output, investigate, and change your new port number if necessary.

As far as the service name, we're going to call the service "hello". Here's how we check for the existence of a current service called "hello":

```
[myuid@mydeskmyuid]$ /sbin/chkconfig -- list | grep -i hello
[myuid@mydeskmyuid]$
```

Once again, if there's no output, there's no such service yet, so you're free to use that service name.
Once you've decided on a port number and a service name (3333 and hello in this tutorial), continue on...

Declare the Port and Name in /Etc/Services:

Add the following line to the /etc/services file:

```
hello           3333/tcp                # hello tutorial, delete when finished
```

The first field is the service name. The second field is the port number and the type of protocol (tcp in this case). Anything to the left of a hash mark (#) is a comment.

Beware: The xinetd man page says anxinetd invoked service needn't be declared in /etc/services. That does not match my experience. I was unable to make this work without declaring it in /etc/services, even using the xinetd port= parameter. Make your life easy. Declare it in /etc/services for the purposes of this tutorial. Then, if you wish, go ahead and find a way to remove it from /etc/services while retaining the service.

Create a File for this Service in /etc/xinetd.d:

As user root, create the following file in /etc/xinetd.d:

```
# default: on
# description: Hello World socket server
service hello
{
  port          = 3333
  socket_type     = stream
  wait          = no
  user          = nobody
  server         = /home/slitt/hello.sh
  log_on_success  += USERID
  log_on_failure  += USERID
  disable        = no
}
```

As root, save the preceding file as /etc/xinetd.d/hello. It should read/write for owner, read for group and other:

```
[root@mydeskxinetd.d]# ls -ldF hello
-rw-r--r--    1 root     root          303 Jan 23 13:14 hello
[root@mydeskxinetd.d]#
```

Once you've saved the file properly, you're ready to "turn on" the server by restarting xinetd...

Restart Xinetd:

According to the xinetd man page there are many ways to "reconfigure" xinetd. In the xinetd man page search for SIGUSR2 and you'll find some of the less intrusive methods. But in my experience, there's no substitute for a known state. I prefer a complete restart. Here's how it's done on a Mandrake box:

```
[root@mydeskxinetd.d]# /etc/rc.d/init.d/xinetd restart
Stopping xinetd: [  OK  ]
Starting xinetd: [  OK  ]
[root@mydeskxinetd.d]#
```

Different Linux distros use different commands and output different text. So your mileage may vary. Once you've restarted xinetd, your new service is theoretically working. Time to test using telnet as a client...

Telnet into the Service, and see the Server App's Output:

Run the following command:

telnet 192.168.100.2 3333

The IP address is the address of the server containing the hello world server. The 3333 is the port number. Here's one example of what might happen:

```
[myuid@mydeskmyuid]$ telnet 192.168.100.2 3333
Trying 192.168.100.2...
telnet: Unable to connect to remote host: Connection refused
[myuid@mydeskmyuid]$
```

In the preceding, note the "Connection refused" message. This means there's a basic problem with the service, and the user listed in /etc/xinetd.d/hello cannot run hello.sh. Note that the log file will not have been rewritten, so if one exists it has an old timestamp (this is why we put the timestamp in the server script). Here are some things to check:

- Make sure the service and name are listed in /etc/services.
- In /etc/xinetd.d/hello, make sure:
  - The port= line's value matches the port number in /etc/services.
  - The service name (just before the squiggly braces) is correct.
  - If there's a disable=, its value must be no.
  - If there's a default: line, its value must be on.
  - The user= line names a valid user.
  - The server= line must point to the script to be run (hello.sh). It must be fully pathed.

After correcting any problems, restart xinetd:

/etc/rc.d/init.d/xinetd restart

If none of the above helps, try **TEMPORARILY** changing the user= line to root. That should fix any permission problems. But make sure that before you change it to root, you disconnect from the Internet. Otherwise the script kiddies will be on you like squirrels on a tree.

If none of the above helps, use the Universal Troubleshooting Process to diagnose the problem. Try to find ways to narrow the problems, and ways to divide and conquer.

Once you actually get the connection to work, you may be confronted with a no-output run, as shown below:

```
[myuid@mydeskmyuid]$ telnet 192.168.100.2 3333
Trying 192.168.100.2...
Connected to 192.168.100.2.
Escape character is '^]'.
Connection closed by foreign host.
[myuid@mydeskmyuid]$
```

In the preceding, the connection was made, but no output was recieved. There's probably something wrong with the script itself. Most likely the top line declaring the script's interpreter (/bin/bash in this example) is missing or wrong. It could also be a pathing problem with the "run service as" user, which is why it's so important to fully path all files in the script. Note that the log file will not have been rewritten, so if one exists it has an old timestamp (this is why we put the timestamp in the server script).

Verify that the script's name is the same as named in /etc/xinetd.d/hello, that it's executable for all, that its directory is executable all the way up the path, and that it contains the #!/bin/bash line at the top, and that there really is a /bin/bash, and that it's a command interpreter. Verify that all files in the script are fully pathed. Beyond that, troubleshoot.

Once you get it running, you might get output like the following:

```
[myuid@mydeskmyuid]$ telnet 192.168.100.2 3333
Trying 192.168.100.2...
Connected to 192.168.100.2.
Escape character is '^]'.
/usr/bin/tee: /tmp/log.log: Permission denied
Hello World:/usr/bin/tee: /tmp/log.log: Permission denied
Tue Jan 23 13:26:44 EST 2001
Connection closed by foreign host.
[myuid@mydeskmyuid]$
```

You can tell you're close. You're connecting, and you're getting output obviously originating from your script. But you have a problem creating /tmp/log.log. That's not surprising, because in a previous step you created it as user myuid, and user nobody hasn't the permissions to delete the old one. As root or another user permissioned to delete /tmp/log.log, delete the old one manually and you should clear the problem. Once all problems have been cleared, the telnet session should look as follows:

```
[myuid@mydeskmyuid]$ telnet 192.168.100.2 3333
Trying 192.168.100.2...
Connected to 192.168.100.2.
Escape character is '^]'.
Hello World:Tue Jan 23 13:33:55 EST 2001
Connection closed by foreign host.
[myuid@mydeskmyuid]$
```

If you obtain the preceding output, congratulations -- you've just implemented a Hello World sockets setup. So let's review...

Review What You've Learned:

Sockets are a software methodology to connect different processes (programs) on the same computer or on different computers. The name "socket" reminds us that once we "plug in" one process into another process's socket, they can talk to each other by reading and writing the socket.

Sockets are a huge and diverse subject. This tutorial deals with the subset in which one of the two processes is a "server" by virtue of its configuration into xinetd (or inetd if you're running an older UNIX or Linux distro). In such a scenario, the "server" process, which in our example is a simple shellscript, is associated with a port number and a process name via a config file in the /etc/xinetd.d directory. In our case, the file is /etc/xinetd.d/hello. Additionally, the port number is associated with the service name and the tcp protocol via an entry in /etc/services. Once those associations are made, restarting xinetd makes the script available as a service at the specified port. Pictorially, it looks something like this:

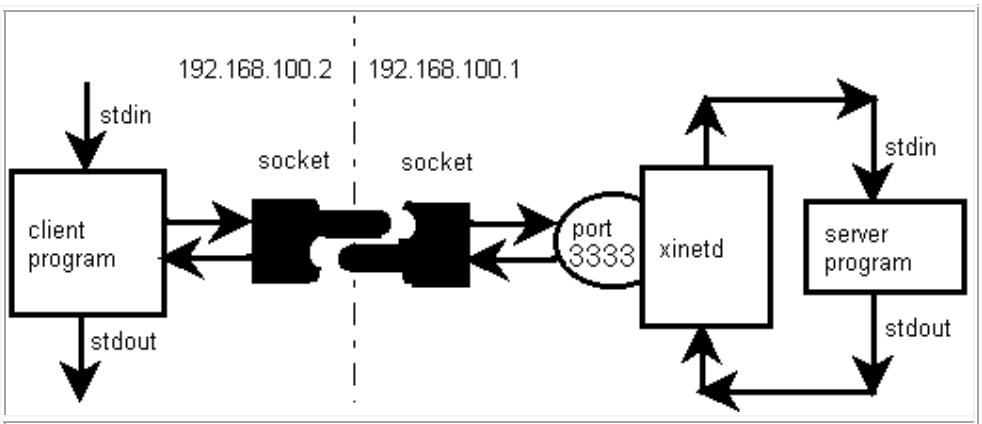

Diagram of client-server socket connection via xinetd. Note that the client communicates by reading and writing the socket, but the server program communicates via stdin and stdout.

In the case of our tutorial, we used telnet as the client program, connecting it to port 3333 on 192.168.100.2 as follows:
telnet 192.168.100.2 3333
The server script's output then appears in the telnet program's output.

This has been a simple implementation indeed. You didn't code the client, but instead used telnet. And communication goes only one way. It's output from the server app and read in telnet. Nevertheless, you're over the biggest hurdle.The next step is two way socket communication. Read on...

A Bidirectional Implementation with Telnet Client:

Now let's make a bidirectional program. We'll simply attach a new script, called hellobidirectional.sh, to the hello service. First, let's make the hellobidirectional.sh script:

```
#!/bin/bash
logfile="/tmp/log.log"
firstname="initialization"
lastname="initialization"
/bin/echo "LOG: $(/bin/date)" > $logfile
/bin/echo >> $logfile
while test 1; do           ###Forever, see the break in the case statement
  /bin/echo >> $logfile
  /bin/echo "Begin Cycle: $(/bin/date) **********" >> $logfile
  /bin/echo
  /bin/echo "Please type Linux celebs name below:" | /usr/bin/tee -a $logfile
  read firstname
  ##### NEXT STATEMENT REMOVES EXTRA CTRL+M PUT IN BY TELNET #####
  ##### PROGRAMMER: MAKE SURE THE CHAR IS CTRL+M, NOT A CARAT AND AN M! #####
  firstname=$(/bin/echo ${firstname}|/bin/sed -e 's/^M$//')
  /bin/echo ":$firstname, " | /usr/bin/tee -a $logfile
  case "$firstname" in
       (richard) lastname="stallman";;
       (linus)   lastname="torvalds";;
       (maddog)  lastname="hall";;
       (q*)      lastname="quit"; break;;
       (*)       lastname="Unknown celeb";;
  esac
  /bin/echo "$lastname" | /usr/bin/tee -a $logfile
done

/bin/echo  | /usr/bin/tee -a $logfile
/bin/echo "Finished" | /usr/bin/tee -a $logfile
```

Warning: The ^M is a CTRL+M, a single character. Unfortunately, this html page can represent it only as a carat followed by an M. If you copy and paste this script into VI, be sure to change the carat and M to a genuine CTRL+M! If your script works on the command line but appears to garble the first name in Telnet, be sure to check for a legitimate CTRL+M.

This script repeatedly queries for a first name, and for three Linux celebrities it returns their last name. For other people it returns the string "Unknown celeb", and for any first name beginning with q it terminates, as you can see in the case statement.

Make the script executable by all, and once again make sure itsdiirectory is executable all the way up the tree. Now run it from the command line. The results should look something like the following, keeping in mind that the bolded text is what you typed in:

```
[myuid@mydeskmyuid]$ ./hellobidirectional.sh

Please type Linux celebs name below:
richard
:richard,
stallman

Please type Linux celebs name below:
linus
:linus,
torvalds

Please type Linux celebs name below:
q
:q,

Finished
[myuid@mydeskmyuid]$
```

Now you know you have a working script. The next step is to attach this script to the hello service. Simply edit /etc/xinetd.d/hello, and replace the following line:

```
  server          = /home/slitt/hello.sh
```

With the following line:

```
  server          = /home/slitt/hellobidirectional.sh
```

Now restart xinetd with the following command:

```
/etc/rc.d/init.d/xinetd restart
```

And finally, use telnet as the client. The following shows the telnet session with text you type bolded:

```
[myuid@mydeskmyuid]$ telnet 192.168.100.2 3333
Trying 192.168.100.2...
Connected to 192.168.100.2.
Escape character is '^]'.
Please type Linux celebs name below:
richard
:richard,
stallman
Please type Linux celebs name below:
maddog
:maddog,
hall
Please type Linux celebs name below:
q
:q,
Finished
Connection closed by foreign host.
[myuid@mydeskmyuid]$
```

If the first names returned by the script (the first names after the ones you type in) are garbled, suspect that the sed command is not removing telnet's trailing CTRL+M characters, and investigate. Troubleshoot any problems as necessary.

A Bidirectional Implementation with Custom C Client:

Use the exact same hellobidirectional.sh you used in the A Bidirectional Implementation with Telnet Client section. But this time we'll access it with a C client. It's not as simple as it might seem. If this section does its job right, you'll learn some of the socket comm gotchas, how to recognize them, how to overcome them, and why professional strength socket clients and servers must know about each others' protocols.

File Blocking, and Deadly Embraces:

One problem you'll see in your career as a socket programmer is file blocking. If you attempt to read an empty socket, you'll wait until the server sends more data. Forever if necessary. That's called File Blocking, and it's the default behavior of sockets. Let's say your client code has a "read all data sent" function containing code something like this:

```
while((len = read(sd, buf, buflen)) > 0)
  write(1,buf,len);
```

That's standard Programming 101, and works perfectly with normal files. But with sockets, the default behavior (called blocking) is that when the read that would have returned 0 is attempted, the read waits for data to come in. So what's wrong with that?

The client is waiting for data from the server, and will not send data to the server until it receives data from the server. But the server will send no data to the client until it receives data from the client. Deadly embrace! The preceding code snippet will hang a socket client under normal conditions. Blocking can be removed with ioctl(), or the buffer can be peeked into with select() or poll(). But these are all rather complicated.

As a Hello World compatible solution (kludge really), instead of reading past the end of data, we'll read up to the end of data by terminating the loop when we read less than the requested number of bytes:

```
while((len = read(sd, buf, buflen)) >= buflen)
  write(1,buf,len);
```

Interestingly enough, even though you requested to read past the end of data, it returns only up to the end of data. Blocking doesn't occur until you try to read a completely drained socket. There's a bug in the preceding code. If the data in the socket is an exact multiple of buflen, the loop won't terminate on end of data, but will try one more read on an empty buffer, causing the previously described deadly embrace. Although in production code that would be inexcusable, in this Hello World exercise we simply minimize the likelihood of it happening by making the buffer large with respect to the expected data. 512 is a good number, and that's what we will use in the client.

Timing Issues -- the sleep() Solution:

Imagine this. After the user inputs data into the client, the client sends data to the server, then quickly iterates the loop and grabs server data out of the socket. The only trouble is, the server hasn't sent the data yet because it hasn't received data from the client. So the client says "the server gave me no data" and proceeds to the next requst for user input. Some time during that user input the server recieves the previous client iteration's data, and sends back a response. This is how server responses can continously be one iteration behind the client, making for user confusion and a Troubleshooter migraine.

Because this exercise is only to teach theory, we "solve" this problem with an expedient kludge -- we insert a sleep() command immediately after the client writes data to the socket. In that way, we guarantee that the server gets the client data and returns its response before the client tries to grab the response. For a similar reason, a sleep() is inserted immediately after the client connects to the socket. For a socket connection to a server on the same box, a 1 second sleep is sufficient. On a heavily trafficked network, that interval may need to be increased.

Please remember, the purpose of this kludge is so you can learn socket programming fundimentals without needing to learn about protocols, stop characters, blocking, ioctl(), and a host of other stuff. As time goes on I will add content to this page describing and walking you through the right way to accomplish these things.

The Client Code:

So here, without further adieu, is the client code:

```
#include <stdio.h>
#include <string.h>
#include <netdb.h>
#include <linux/in.h>
```

```
#include <sys/socket.h>
#include <unistd.h>


#define buflen 512
unsigned intportno = 3333;
char hostname[] = "192.168.100.2";

char *buf[buflen];     /* declare global to avoid stack */

void dia(char *sz) { printf("Dia %s\n", sz); }

intprintFromSocket(intsd, char *buf)
  {
  intlen = buflen+1;
  intcontinueflag=1;
  while((len>= buflen)&&(continueflag)) /* quit b4 U read an empty socket */
    {
    len = read(sd, buf, buflen);
    write(1,buf,len);
    buf[buflen-1]='\0';  /* Note bug if "Finished" ends the buffer */
    continueflag=(strstr(buf, "Finished")==NULL); /* terminate if server says "Finished" */
    }
  return(continueflag);
  }


main()
   {
   intsd = socket(AF_INET, SOCK_STREAM, 0);  /* init socket descriptor */
   structsockaddr_in sin;
   structhostent *host = gethostbyname(hostname);
   char buf[buflen];
   intlen;

   /*** PLACE DATA IN sockaddr_instruct ***/
   memcpy(&sin.sin_addr.s_addr, host->h_addr, host->h_length);
   sin.sin_family = AF_INET;
   sin.sin_port = htons(portno);

   /*** CONNECT SOCKET TO THE SERVICE DESCRIBED BY sockaddr_instruct ***/
   if (connect(sd, (structsockaddr *)&sin, sizeof(sin)) < 0)
     {
     perror("connecting");
     exit(1);
     }

   sleep(1);   /* give server time to reply */
   while(1)
     {
     printf("\n\n");
     if(!printFromSocket(sd, buf)) break;
     fgets(buf, buflen, stdin);   /* remember, fgets appends the newline */
     write(sd, buf, strlen(buf));
     sleep(1);              /* give server time to reply */
     }
   close(sd);
   }
```

The main routine can be divided roughly into two parts:

1. Socket initialization with connection, ending with its connect statement and associated error handling
2. Interaction with server via socket, beginning with the first sleep() command.

Believe it or not, part 1 is the more straightforward. Basically, you prepare a socket descriptor (sd) by creating it, initializing a hostentstruct (host) and a sockaddr_instruct (sin). Various copies and other calls prepare the socket descriptor, and then the connect statement connects the socket descriptor with the service defined in the sockaddr_instruct. Once that connect is done, the socket descriptor can be treated very much like a file descriptor or file handle. They're both integers, and they can both be used in read() and write() calls.

The second part is the challenge. Most obvious, there's no clean way to end the loop unless we get a clue from the server. In this case, the server prints the word "Finished" when it's finished. We create a function called printFromSocket() to handle all aspects of reading server data from the socket, as well as telling the main loop when "Finished" has been recieved so we can terminate the loop.

To prevent deadly embrace, the printFromSocket() function does the "to the end but not beyond" read discussed previously. It also searches for the word "Finished" after every read. When it finds that string it terminates its loop and also returns the fact that it found that string, so that the main loop can terminate. An understanding of the printFromSocket() function yields a good understanding of this client program.

Other than the above, the only tricky part is that a sleep() is inserted after the connect, and after every data transmission from the client. This is to prevent the timing problems previously discussed.

Conclusion:

Although the preceding exercises were obviously academic, using workarounds unacceptable in production code, upon their completion you have an excellent understanding of socket basics. At this point you can take advantage of more advanced socket documentation to learn to make production code. And better still, in the coming months I'll be adding more info to this page...

Looking Ahead:

I'll be placing some more content on this page in the coming months. That content will address the practicalities of production code, eliminating the need for the obvious kludges you've seen in the previous exercises. Some of the issues that will be addressed are:

- Turning off blocking with ioctl()
- Buffer introspection with select() and poll()
- Using end of transmission and end of program bytecodes to implement synchronous timing the right way, instead of the sleep() workaround.
- Discussing differences between synchronous and asynchronous communication between client and server. Asynchronous communication occurs when the client can respond to the server's data at any time, regardless of the client's state. And vice versa -- the server can respond to the client's data at any time, regardless of the server's state. Telnet is an excellent example of asynchronous client-server communication.
- Creating a Perl server app
- Creating a Perl client

A Perl Socket Server:

```
#!/usr/bin/perl -w

use strict;

my($logfile)="/tmp/log.log";
my($firstname)="initialization";
my($lastname)="initialization";

sub dia
  {
#  print "dia $_[0] \n";
  }

sub teeappendlog
  {
  system("/bin/echo $_[0]  | /usr/bin/tee -a $logfile");
  }
```

```
sub appendlog
  {
  system("/bin/echo " . $_[0] . " >> " . $logfile);
  }

dia "b4 log trunk";
system("/bin/echo  > $logfile");
dia "b4 bin date to log trunk";
appendlog "LOG: " . `/bin/date`;
dia "afer bin date to log trunk";
appendlog "";
while (1)            ###Forever, see the break in the case statement
   {
   appendlog "";
dia "b4 begincycle";

   my($temp) = `/bin/date`;
   chomp($temp);
   appendlog "Begin Cycle: $temp  **********";

   teeappendlog "Please type Linux celebs name below:";
   $firstname = <STDIN>;
dia "after read";
   ##### NEXT STATEMENT REMOVES EXTRA CTRL+M PUT IN BY TELNET #####
   ##### PROGRAMMER: MAKE SURE THE CHAR IS CTRL+M, NOT A CARAT AND AN M! #####
   $firstname =~ s/
$//;
   chomp($firstname);

dia "after chomp";
   teeappendlog ":perlversion:$firstname, ";
   $lastname = "Unknown celeb";
   if ($firstnameeq "richard") { $lastname = "stallman";}
   elsif ($firstnameeq "linus") { $lastname = "torvalds";}
   elsif ($firstnameeq "maddog") { $lastname = "hall"}
   elsif ($firstname =~ /^q/) { $lastname = "quit"; last;}
   teeappendlog "$lastname";
   dia "endloop " . $lastname . ", " . $firstname;
   }

teeappendlog "$logfile";
teeappendlog "Finished";
```