Tcp/IP socket programming

Sockets enable your program or application to talk to other machines over the network. When you type in google.com in your browser, it talks to google.com over the internet and fetches the webpage. For socket programming we need to use the socket library or api which provides some very simple functions to do such a network communication. In this post we shall learn about socket programming in perl.

Sockets basically provide an "virtual endpoint" through which data can be transmitted or received. Now this data has to be in a certain format (or say language), and that format must be understood by both ends of the communication channel for the communication to be meaningful. These formats are called "Protocols" or rules which are accepted by all the communicating partners. There are many protocols in socket communication and each has a different purpose depending on its features. When it comes to data transfer over the internet the most commonly used protocol is the TCP/IP. Over here there are actually 2 protocols involved. The first is IP (Internet Protocol) and second one is TCP (Transmission Control Protocol).

A single message in socket communication can have data formatted according to multiple protocols. For example when you are browsing webpages, the socket messages have 3 protocols involved, namely IP, TCP and HTTP. Each protocol serves a certain purpose.

Socket programming in Perl can be done using the "low level socket functions" or the IO::Socket module. The IO::Socket module provides an object oriented interface to the low level socket functions. The low level socket functions resemble the socket functions of C. In this tutorial we are going to use the low level socket functions.

Create a socket

The socket function can be used to create a socket. In the following code example we shall create a tcp socket.

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto) or die $!;
```

Note the die "$!" towards the end of the line. It prints out the error message if an error occurs.

Closing a socket

When a socket is no longer required it can be closed by calling the close function over the socket descriptor.

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto) or die $!;
close($sock);
```

Connecting to a remote server

Once a socket is created, the next meaningful task would be to connect to some remote server using the socket. The server or system can be over the internet or LAN.

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto) or die $!;
# 2. connect to remote server
$remote = 'www.google.com';
$port = 80;
$iaddr = inet_aton($remote) or die "Unable to resolve hostname : $remote";
$paddr = sockaddr_in($port, $iaddr);    #socket address structure
connect($sock , $paddr) or die "connect failed : $!";
print "Connected to $remote on port $port\n";
close($sock);
exit(0);
```

Output

```
$ perl test.pl
Connected to www.google.com on port 80
```

In the above example we connected to google.com on port 80. Before the connect function is called the sockaddr_in structure has to be setup. The sockaddr_in structure stores the information about the remote address, address type and port number. The inet_aton function converts a hostname/ip to ip address in long number format.

Send Data

Once connected its time to start communication by sending some data. The send function can be used for this.

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto) or die $!;
```

```
# 2. connect to remote server
$remote = 'www.google.com';
$port = 80;
$iaddr = inet_aton($remote) or die "Unable to resolve hostname : $remote";
$paddr = sockaddr_in($port, $iaddr);    #socket address structure
connect($sock , $paddr) or die "connect failed : $!";
print "Connected to $remote on port $port\n";
# 3. Send some data to remote server - the HTTP get command
send($sock , "GET / HTTP/1.1\r\n\r\n" , 0);
close($sock);
exit(0);
```

In the above example we are sending the message "GET / HTTP/1.1\r\n\r\n" to the remote server. This is actually the http command for getting the index page of a http server.

Receive data

After sending data, its time to receive the reply of the server. The recv function can be used for this

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto)
or die "could not create socket : $!";
# 2. connect to remote server
$remote = 'www.google.com';
$port = 80;
$iaddr = inet_aton($remote)
or die "Unable to resolve hostname : $remote";
$paddr = sockaddr_in($port, $iaddr);    #socket address structure
connect($sock , $paddr)
or die "connect failed : $!";
print "Connected to $remote on port $port\n";
# 3. Send some data to remote server - the HTTP get command
send($sock , "GET / HTTP/1.1\r\n\r\n" , 0)
or die "sendo failed : $!";
# 4. Receive reply from server - perl way of reading from stream
# can also do recv($sock, $msg, 2000 , 0);
while ($line = <$sock>)
{
print $line;
}
close($sock);
exit(0);
```

Output

```
$ perl test.pl
Connected to www.google.com on port 80
HTTP/1.1 302 Found
Location: http://www.google.co.in/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=ba64f3db0170b4d4:FF=0:TM=1347869092:LM=1347869092:S=sTs9RU5ODJJRZsGk; expires=Wed, 17-Sep-2014 08:04:52 GMT; path=/; domai
Set-Cookie: NID=64=zYnXu70YZzwCSr2GJ_QwXQbsq39dFC_k4jfMa-bIQgVi3eY05HAVzxkR0_0keZSKeZYWhJ1DBLk4Kv2qo5dkGi8YF4BwQMGDGnvxs2h7ywLnnMYLe021JCxTxiQ
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for more info."
Date: Mon, 17 Sep 2012 08:04:52 GMT
Server: gws
Content-Length: 221
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.in/">here</A>.
</BODY></HTML>
```

So our socket program connected to google.com on port 80 and send a message and received the reply. The reply is an http page.

Revise

In the above examples we learned the following socket operations.

1. Create a socket
2. Connect to remote server/system.
3. Send data
4. Receive a reply

The socket program we wrote above is called a client. A client connects to a remote server for exchange of information/data. Apart from the client the other type of socket program we are going to write is called a server. The server serves data to clients connecting to it. So lets move on to the next section to do some server programming in perl

Server

The basic steps for making a server are :

1. Create a socket
2. Bind to a local address and local port
3. Listen for incoming connections
4. Accept incoming connections
5. Communicate the with the newly connected client - send and receive data.

So a server does not connect out to a system, instead it waits for incoming connections. We have already seen how to create a socket. So the next task to make a server would be to bind the socket. The bind function can be used for this task.

Lets take a quick example

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$| = 1;
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto)
or die "could not create socket : $!";
# 2. bind to local port 8888
$port = 8888;
bind($sock , sockaddr_in($port, INADDR_ANY))
or  die "bind failed : $!";
```

The socket is bound to port 8888 and address "any". After bind the socket has to be put in listen mode.

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$| = 1;
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto)
or die "could not create socket : $!";
# 2. bind to local port 8888
$port = 8888;
bind($sock , sockaddr_in($port, INADDR_ANY))
or  die "bind failed : $!";
listen($sock , 10);
print "Server is now listening ...\n";
```

After calling listen, the socket is ready to accept incoming connections using the accept function. This shall be done in a loop so that the program can accept connections again and again.

Accept incoming connections

```
#!/usr/bin/perl -w
use Socket; # For constants like AF_INET and SOCK_STREAM
$| = 1;
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
my($sock);
socket($sock, AF_INET, SOCK_STREAM, $proto)
or die "could not create socket : $!";
# 2. bind to local port 8888
$port = 8888;
bind($sock , sockaddr_in($port, INADDR_ANY))
or  die "bind failed : $!";
listen($sock , 10);
print "Server is now listening ...\n";
#accept incoming connections and talk to clients
while(1)
{
my($client);
$addrinfo = accept($client , $sock);
my($port, $iaddr) = sockaddr_in($addrinfo);
my $name = gethostbyaddr($iaddr, AF_INET);
print "Connection accepted from $name : $port \n";
#send some message to the client
print $client "Hello client how are you\n";
}
#close the socket
close($sock);
exit(0);
```

Connect to this server using the telnet command in another terminal.

The output of the telnet command should be similar to this

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello client how are you
Connection closed by foreign host.
```

The connection closed immediately because in the server program, the $client variable is reset and hence the connection. This server is too simple and cannot handle multiple clients at a time.

Asynchronouse I/O

Multiple clients can be handled by asynchronous event driven I/O. IO::Select can be used for this.

```
#!/usr/bin/perl -w
use strict;
use Socket; # For constants like AF_INET and SOCK_STREAM
use IO::Select;
$| = 1;
my($proto, $port , $sock , $s , @ready , $so , $addrinfo , $client , $inp);
$proto = getprotobyname('tcp');    #get the tcp protocol
# 1. create a socket handle (descriptor)
socket($sock, AF_INET, SOCK_STREAM, $proto)
or die "could not create socket : $!";
# 2. bind to local port
$port = $ARGV[0];
bind($sock , sockaddr_in($port, INADDR_ANY))
or  die "bind failed : $!";
listen($sock , 10);
print "Server is now listening ...\n";
#accept incoming connections and talk to clients
$s = IO::Select->new();
$s->add($sock);
while(1)
{
@ready = $s->can_read(0);
foreach $so(@ready)
{
#new connection read
if($so == $sock)
{
my($client);
$addrinfo = accept($client , $sock);
my($port, $iaddr) = sockaddr_in($addrinfo);
my $name = gethostbyaddr($iaddr, AF_INET);
print "Connection accepted from $name : $port \n";
#send some message to the client
send($client , "Hello client how are you\n" , 0);
$s->add($client);
}
# existing client read
else
{
chop($inp = <$so>);
chop($inp);
print "Received -- $inp \n";
#send reply back to client
send($so , "OK : $inp\n" , 0);
}
}
}
#close the socket
close($sock);
exit(0);
```

output

```
$ perl server.pl 8888
Server is now listening ...
Connection accepted from localhost : 37556
Received -- hello
Received -- how are you
Connection accepted from localhost : 37558
Received -- I am the second client
Received -- I am the first one again
```

To test the program, first run it in 1 terminal, then connect to it from 2 other terminals and send some message.
The server would accept the connection from both terminals and communicate with both.

Last Updated On : 14th May 2013