Perl and Socket-Based Client/Server Programming in Introductory OS Courses

Golden G. Richard III
Dept. of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: *golden@cs.uno.edu*

Abstract

Perl has become a popular language for many tasks, including solving problems in system administration, for report generation, and for adding functionality to pages on the World Wide Web (WWW). This paper describes my experiences with introducing Perl into undergraduate operating systems courses and in using Perl to teach sockets-based client/server programming. One of the major impediments to this effort was a lack of documented Perl examples; this paper also attempts to fill that need by providing some examples I have successfully used. Several client/server Perl assignments are also discussed. The response from students to these efforts has been very positive.

1. Introduction

Perl [2, 5] has become a popular language for doing report generation, writing Common Gateway Interface (CGI) scripts for the World Wide Web (WWW), for various system administration tasks, and others. The popularity of Perl stems partly from the fact that it melds C and shell programming with the functionality of several traditional Unix tools such as *sed* and *awk*. Perl has integrated regular expression matching and provides an easy interface to most Unix system calls, including the various socket calls which support interprocess communication. Often problems that would require hundreds of lines of code in C can be solved using only tens of lines of Perl. And Perl has been ported to a wide variety of platforms.

One of Perl's drawbacks is that it can be difficult to master because of its broad functionality. Another is that poorly documented Perl can be very difficult to understand. W. Richard Stevens, author of the excellent book on network programming [3] once said "No, I don't use Perl. I consider it an unreadable write-only language." [4] There's truth in that statement. One learns Perl by writing Perl. And reading documented Perl examples, if one can find them. And then writing more Perl.

This paper discusses my motivation for and experiences with introducing Perl into my introductory operating systems classes. It also provides some documented examples that I offer to my students. It is not intended in itself as a tutorial on sockets programming in Perl, though examination of the examples will help in this regard. In the remainder of this section, I explain my motivation for introducing Perl to undergraduate operating systems students. Section 2 very briefly describes the differences between the C and Perl sockets interfaces. Section 3 describes some teaching materials I have developed and Section 4 offers conclusions.

It makes sense to introduce undergraduate computer science students to Perl for a variety of reasons. One is the steady industrial demand for systems support staff and the wide acceptance of Perl in these environments. Another is the fact that it behooves students to have a variety of tools in their toolbelt. Often I see my students choose Ada or C or some other traditional language to solve a small problem which could be solved very quickly in a language like Perl. While I would certainly not advocate doing large-scale software development in Perl, I find it very useful for relatively small problems. So my goal became introducing senior-level students to Perl by having them work on interesting projects which had a purpose other than Perl-in-itself: client/server networking programming using Unix sockets. A first round of course evaluations reveals that the students were overwhelmingly enthusiastic about their Perl experiences.

2. Socket Based Communication in Perl

Perl provides IPC facilities based on the Berkeley Unix socket mechanism [6], so the usual functions such as *socket*, *socketpair*, *accept, select*, etc. are available.[1] Calls generally have the same name and similar arguments as under C, but there are several differences. The first is that Perl has "proper" strings and thus information about the length of Perl strings generally needn't be provided. Another difference is that when using the *select* system call, the helper macros *FD_SET, FD_CLR, FD_ZERO,* and *FD_ISSET* aren't available. Instead, Perl provides an assignable function *vec* which can be used to build the arguments to *select*. For example, the Perl statement *vec($rin, fileno(S), 1) = 1;* sets the bit corresponding to file handle *S*, and *if (vec($rin, fileno(S), 1))* checks the same bit. A further difference is that calls which require C *struct* arguments, such as *bind*, require use of the Perl *pack* function for proper setup.

A major difference between Perl and C is that file handles are not first class objects in Perl, but rather are symbols. This is particularly troublesome when arrays of file handles are necessary, as in I/O multiplexing servers. My examples use the following method to generate arrays of file handles:

```
$con++;
$sockets[$con -1] = "NS".$con;
```

Assuming the integer variable *$con* is initialized to 0, this snip of Perl creates file handles named *NS1*, then *NS2*, etc. A new file handle can then be used like this:

```
($addr = accept($sockets[$con - 1], S))
|| die $!;
```

This call accepts an incoming connection on the bound socket *S* and associates it with the newly created handle *$sockets[$con - 1]*. Of course the file handles *NS1*, *NS2*, etc. are global to the entire Perl code.

I had a good deal of interaction with undergraduate students taking both a computer networking course and my introductory operating systems course during Spring 1996. The students in the computer networking class were doing some sockets programming in C whereas my students were using Perl. Some students were attending both classes. The students versed in Perl had little difficulty applying their Perl-based sockets programming experience to C. In fact, their enthusiasm for Perl and sockets programming quickly spread around the department…at the beginning of one OS lecture, when we were to wrap up the discussion of Perl-based sockets programming, I arrived to find my classroom with standing room only. Quite a few of the networking students who were not also enrolled in OS and a handful of others had decided to come and see what was going on!

3. Perl Client/Server Examples and Assignments

It has been my experience that students initially find sockets programming very confusing and then after seeing a number of examples, gradually warm up to the subject. I would be surprised if this experience was atypical. While there are many examples of small, socket-based applications in Perl available (via the World Wide Web, for example), *very* few of them are carefully documented, if they contain any documentation at all. The combination of Perl's terseness *and* the unfamiliarity of sockets concepts make such examples a daunting challenge for students. I finally gave up the search and took examples from several

---

[1] Note that some ports of Perl to non-Unix environments do *not* support all the socket functions. For example, the Windows NT ports do not currently support *select, socketpair,* and several other common functions.

sources and carefully documented them, along with writing a few documented examples from scratch. A few of these examples are:

- A forking server, which is essentially a documented version of the server example on pp. 344-345 of *Programming Perl* [5]. I've rearranged some of the source to put related bits together, such as the *pack* and *bind* stuff.
- An I/O multiplexing server, which I wrote from scratch. One of the ugly spots in Perl (discussed in Section 2) shows through in this example, namely that Perl doesn't treat file handles as first class objects. This makes arrays of file handles somewhat troublesome. This server sends information about new connections to all existing clients and echoes data from individual clients back to the sender.
- A simple client, which can be used with either the I/O multiplexing server or the forking server. Input typed at the client is relayed to the server. Data received from the server is echoed to the terminal. This client is essentially a heavily documented version of the client example found on pp. 342-344 of *Programming Perl* [5].

I've included the last two in the present paper in Appendix A. The examples are also available via my Web page at:

http://www.cs.uno.edu/~golden/teach.html

Students in my undergraduate operating systems courses are generally given a choice between several socket-based client/server assignments. If additional functionality is required, the assignments are typically good candidates for teams of two students. Two assignments which students have particularly enjoyed working on are:

- A distributed remote control facility for Netscape. For this assignment, the server reads a list of machine names and uses *rsh* to spawn an identical client on each of the machines. The server then accepts commands and directs the clients to display specified web pages, open files, add bookmarks, etc. The clients use Netscape's remote control facility [7] to interact with the browser, whereas communication between the server and clients is performed using sockets. This assignment requires students to use *.rhost* files to fully automate the spawning of clients, which can be both educational (for the student) and somewhat troublesome (for the system administrator; there may be policies against such things). An option which I haven't explored is to require students to use perl/Tk, a Tk-like [1] facility for Perl, to design a more sophisticated user interface for the server.
- A simple distributed database server, which accepts commands such as *QUERY*, *DEFINE*, etc. from any number of clients. All clients compete for access to a shared database. The number of commands and sophistication of the server can be adjusted to suit a particular course's timetable. Clients request service from the server at a well-known port address. A sample interaction on the client side might look like:

```
> QUERY blarg
< Server:  No definition
associated with 'blarg'.
> DEFINE blarg A word that
pirates say
< Server: Definition for
'blarg' accepted.
> QUERY blarg
< Server:  'blarg' : A word
that pirates say
> DEFINE blarg A word that
pirates sometimes say when they
are upset
< Server: Definition for
'blarg' accepted.
> QUERY blarg
< Server: 'blarg' : A word that
pirates sometimes say when they
are upset
> EXIT
```

The latter assignment provides an opportunity to introduce I/O multiplexing in Perl using select. The appropriate server example is the I/O-multiplexing example provided at the end of this paper. This assignment also presents an opportunity for students to use Perl's associative array facility.

4. Conclusions

This paper discusses my motivation for introducing Perl into OS classes where C is traditionally used to teach sockets programming. Major factors include the wide acceptance of Perl in the systems support arena and the use of Perl in WWW applications. Another factor is the applicability of Perl to problems where the need for features such as regular expression matching and powerful report generation might result in many, many more lines of code being written if traditional languages such as C or Ada were used. The Perl and C sockets mechanisms are closely related, so in my experience students find it relatively easy to migrate between them.

In addition, the paper also presents several fully-documented Perl sockets examples. In my opinion, one of the difficulties in introducing Perl to students is the lack of properly documented examples. I also briefly discuss two Perl sockets programming assignments that I have used in my introductory operating systems classes. The response from my students regarding the introduction of Perl and the use of Perl for sockets programming in the first OS course has been very positive.

References

[1] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[2] R. L. Schwartz, *Learning Perl*, O'Reilly and Associates, 1993.

[3] W. R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.

[4] W. R. Stevens, "Experts' Choices of Must Have Books," *Addison-Wesley Innovations*, Winter 1995/1996, p. 10.

[5] L. Wall, R. L. Schwartz, *Programming Perl*, O'Reilly and Associates, 1991.

[6] *perlipc* man page.

[7] "Remote Control of Unix Netscape," *http://www.netscape.com/newsref/std/x-remote.html.*

Appendix A

```
#!/opt/local/bin/perl
##########################################################################
#
# I/O Multiplexing Server example.  This example is available via the WWW
# at http://www.cs.uno.edu/~golden/teach.html
#
# By Prof. Golden G. Richard III, Dept. of Computer Science, University
# of New Orleans, April 1996.  Use for any purpose whatsoever granted.
#
# This server accepts connections from an arbitrary number of clients.
# When a new connection is established, information about the new connection
# is sent to all connected clients.   Input from individual clients is
# simply echoed back to the sender.
#
# Command line arguments expected are:   port
#
# Notice that arrays of file handles in Perl are incredibly yucky, since
# file handles aren't first class objects.  It's not my fault.
#
##########################################################################
#
# Perl socket stuff is based on, and very similar to, C socket stuff.
# The Unix man pages for the various socket system calls such as
# 'listen', 'bind', etc. may be very useful to you.  To get
# information on a particular call, such as 'accept', use the following
# command:
#
#  % man -s3N accept
#
# The -s3N tells man to look in 3N section of the Unix manual.
#
# Ranges of port addresses will be assigned to you in class.  Generally, if
# you get an "address already in use" error, try a different port.
# We'll be using ports in the ranges 5000-6000.
#
##########################################################################

# In Perl 5 and above, the Socket module contains Perl definitions for
# the stuff in the C include file "/usr/include/sys/socket.h".
# Rather than manually poking through that include file to get values
# for each architecture, a "use Socket" assures that you'll get the right
# values.  This replaces the following lines in the examples you'll find in
# the Programming Perl book:
#
# $AF_INET = 2;
# $SOCK_STREAM = 1;
#
# which turn out to be wrong for Solaris, anyway
# ($SOCK_STREAM should be 2)
#
# In general, hard-coded socket constants in Perl 5+ are a bad practice.  Use
# the Socket module instead.

use Socket;
```

```perl
sub readline {

# 'select'-compatible function for reading one line of input from
#     a filehandle.
#
#     readline() expects one argument -- filehandle S
#
#     sample call:       $mystring = readline('S')
#

      my $filehandle = $_[0];
      my $c='';
      my $retstr='';
      my $endoffile=0;

      while ($c ne "\n" && ! $endoffile) {
            if (sysread($filehandle, $c, 1) > 0) {
                  $retstr = $retstr . $c;
            }
            else {
                  $endoffile=1;
            }
      }

      return $retstr;
}


# Sucks in the command line arguments (resident in the array variable ARGV)
# and assigns them to the variables listed in ().  In this case, the first
# command line argument is the port [more on this later] and it's
# placed into the variable 'port'.  We don't care about any other command-line
# arguments.

($port) = @ARGV;

# This checks to see if 'port' has been assigned a value and if it hasn't,
# assigns the default port value 2345

$port = 2345 unless $port;

# Looks up important information related to the network protocol you wish
# to use.   'tcp' is a connection-oriented protocol.  Look in
# the file /etc/protocols for examples of others.   Don't change this unless
# you know what you're doing.

($name, $aliases, $protocol) = getprotobyname('tcp');
```

```
# In the following, $port is the port the user specified either on the
# command line or by default.   =~ is the Perl regular expression binding
# operator and !~ is the negation of =~.   Regular expressions are
# enclosed between / / characters in Perl.   \d matches a single digit,
# \d+ matches a string of 1 or more digits, ^ specifies "beginning of
# string", and $ specifies "end of string".    /^\d+$/ as a regular
# expression, therefore, matches integers of arbitrary length.
# So...   The following line says:  If the port contains any non-digit
# characters, look up the port number associated with the symbolic
# name specified by the user.   The lookup is done against the
# file/etc/services.  Unless you're hacking code to talk to well-established
# servers such as the httpd daemon, for example, you'll use numeric port
# addresses, so don't worry about this too much.

if ($port !~ /^\d+$/) {
     ($name, $aliases, $port) = getservbyport($port, 'tcp');
}

# Let the user know what port we're listening on, just in case (s)he
# accidentally typed an incorrect port.

print "Listening on port $port...\n";

# 'socket' creates one endpoint for a communication link (think of it as
# creating a telephone. Later, someone else will create another telephone
# and wires will be attached between them).   The S parameter is the handle
# associated with the created communication endpoint.    AF_INET specifies
# that we're talking using ports.   AF_UNIX would specify that we'd be
# communicating through special files created in the filesystem.  This
# is very attractive because then you can do away with the port number
# business, but unfortunately AF_UNIX sockets only work on the same
# machine.  SOCK_STREAM sockets communicate using streams of characters.
# Another possibility is unreliable datagram communication using
# SOCK_DGRAM. Just stick with the parameters used here unless you
# know what you're doing.

# In case you're wondering, the AF_INET and SOCK_STREAM symbols are
# provided by the 'use Socket;' statement at the top of the file.
# You do NOT want to put $'s in front of these symbols.


# The 'die' causes execution to terminate with an error message ( which is
# stored in $_) if the 'socket' call fails.   $_ is a string describing
# the latest system error message (corresponds to the C 'errno' variable)

socket(S,AF_INET,SOCK_STREAM,$proto) || die "socket : $!";

# The 'bind' hooks your phone to the port number that was specified.
# Think of all the ports as a telephone switchboard.  'bind' requires its
# parameters to be in a C structure format.  'pack' smooshes things
# together into a form that 'bind' can stomach.  The $sockaddr thing below
# says "An unsigned short, followed by a short in 'network order',
# followed by a null-padded 4 character string, followed by 8 null
# bytes."  It's magic.  Don't worry too much about it.

$sockaddr = 'S n a4 x8';
$this = pack($sockaddr, AF_INET, $port, "\0\0\0\0");
bind(S, $this) || die "bind : $!";
```

```
# This says "I want 10-way call-waiting."  Basically, that you want
# requests to talk to you queued up to a depth of 10.

listen(S, 10) || die "listen: $!";

# Select S temporarily as the default output channel, turn off buffering,
# then select standard output again.

select(S);
$| = 1;
select(STDOUT);

$con = 0;

# Server runs forever, accepting connections from clients

while (1) {

      # initialize select bit vectors
      $rin = $win = $ein = '';

      # If this socket is ready, we have a new connection
      vec($rin, fileno(S), 1) = 1;

      # ...also want to snoop on existing connections.  The check for null
      # string is done because we forget about connections by zapping the
      # file handle.
      foreach (@sockets) {
            vec($rin, fileno($_), 1) = 1;
      }

      # Wait for some action... the 'select' call will block until one of
      # of the sockets whose bit we set above becomes available for reading.
      # A new connection also falls into this category.  The undef's are here
      # because we don't care about the output, exceptional conditions, and
      # timeout capabilities of 'select' in this simple example.

      print "Waiting...\n";
      ($numready, $timeleft) =  select($rout=$rin, undef, undef, undef);
      print "Someone's knocking!!\n";

      # Is it a new connection?  New connections will cause activity on S

      if (vec($rout, fileno(S), 1)) {
            # Someone new is ringing...
            print "New connection.\n";
            # Make a new file handle (yuck).  "." concats strings.
            $con++;
            $sockets[$con - 1] = "NS".$con;

            # 'accept' blocks until it notices that an incoming connection on
            # socket S.  When this occurs, the incoming connection is
            # actually attached to the socket NSn rather than S, thus leaving
            # S free for other incoming connections.
```

```
        # The value that's returned (in $addr) gives some information
        # about the address of the caller.

        ($addr = accept($sockets[$con - 1], S)) || die $!;

        select($sockets[$con - 1]);
        $|=1;
        select(STDOUT);

        # unpack the information returned by 'accept' to get some
        # (readable) information about the client we're serving and
        # print it to standard output

        ($af,$port, $inetaddr) = unpack($sockaddr, $addr);
        @inetaddr = unpack('C4', $inetaddr);

        # tell the other clients who just connected
        foreach (@sockets) {
              if ($_ ne "") {
                    print $_
                    "Got connection $con @ Internet address @inetaddr\n";
              }
        }
    }
    else {
        # An existing connection is trying to talk to us...
        # just display whatever each waiting client sends
        # and then echo the data back to the client
        $sockcount = 0;
        foreach (@sockets) {
              $sockcount++;
              $thesocket = $_;
              if (vec($rout, fileno($thesocket), 1)){
                    # read data from socket; select-oriented applications
                    # don't work well with <> type reads.  It's safer
                    # to use an unbuffered read
                    $data = &readline($thesocket);
                    if (! length($data)) {
                          # 0 length message means "goodbye"
                          print "Bye to client on socket $thesocket.\n";
                          close($thesocket);
                          $sockets[$sockcount-1]= "";
                    }
                    else {
                          print "Received from client $sockcount:$data\n";
                          print $thesocket "$data";
                    }
              }
        }
    }
}
```

```perl
#!/opt/local/bin/perl
##########################################################################
#
# Slightly hacked and heavily documented simple Client example from
# the book Programming Perl [pp. 343-344]. This example is available via
# the WWW at http://www.cs.uno.edu/~golden/teach.html
#
# By Prof. Golden G. Richard III, Dept. of Computer Science, University
# of New Orleans, April 1996.   Use for any purpose whatsoever granted.
#
# Some things are done a little differently than in the server examples
# to give you a little more experience with Perl constructs.
#
# This client accepts input from standard input, sends the input to a
# server, and sends responses from the server to standard output.
#
# Command line arguments expected are:    IP address of server, port
#
##########################################################################
#
# Perl socket stuff is based on, and very similar to, C socket stuff.
# The Unix man pages for the various socket system calls such as
# 'listen', 'bind', etc. may be very useful to you.  To get
# information on a particular call, such as 'accept', use the following
# command:
#
#  % man -s3N accept
#
# The -s3N tells man to look in 3N section of the Unix manual, a section
# that's not searched by default.
#
# Use port addresses in the range 5000-6000.   If you get an "address
# already in use" error, try a different port.
#
##########################################################################

# In Perl 5 and above, the Socket module contains Perl definitions for
# the stuff in the C include file "/usr/include/sys/socket.h".
# Rather than manually poking through that include file to get values
# for each architecture, a "use Socket" assures that you'll get the right
# values.  This replaces the following lines in the example in the book:
#
# $AF_INET = 2;
# $SOCK_STREAM = 1;
#
# which turn out to be wrong for Solaris, anyway...
# ($SOCK_STREAM should be 2)

use Socket;

# Sucks in the command line arguments (resident in the array variable ARGV)
# and assigns them to the variables listed in ().  In this case, the first
# is the IP address of the machine on which the server is
# (hopefully) running.  The second is the port on which the server is
# listening.

($them,$port) = @ARGV;
```

```
# This checks to see if 'port' has been assigned a value and if it hasn't,
# assigns the default port value 2345.   If 'them' (the machine on which
# the server is running) hasn't been assigned a value, 'localhost' is
# assumed.  This means that the server is on the same machine as the
# client (us).

$port = 2345 unless $port;
$them = 'localhost' unless $them;

# This sets up a signal handler to kill the child we'll create later
# in the event that we die

$SIG{'Int'} = 'dokill';

# This is a subprogram that does the killing.   The variable $child is
# initially undefined, so we won't try to kill a non-existant child.
# Later it will be the PID of the child we create.

sub dokill {
      kill 9, $child if $child;
}

# Strings in back quotes like `ls` run the Unix command inside the quotes
# and return the output.   The hostname command returns the name of the
# machine we're running on.   'chop' removes the newline character in the
# output from the hostname command.

chop($hostname = `hostname`);

# Looks up important information related to the network protocol you wish
# to use.   'tcp' is a connection-oriented protocol.  Look in
# /etc/protocols for examples of others. Do NOT change this unless you
# know what you're doing.

($name, $aliases, $proto) = getprotobyname('tcp');

# This is a slightly different way of doing the service lookup that was
# done in the Server code.  Basically, if the specified port is an integer,
# the port number lookup isn't done, otherwise it is.

($name, $aliases, $port) = getservbyname($port,'tcp')
                  unless $port =~ /^\d+$/;

# Let the user know what port we're using and where the server
# is expected to be, just in case (s)he accidentally typed an
# incorrect port or machine name.

print "Using port $port to connect to server on host $them...\n";

# This looks up numeric IP address information corresponding to the
# hostname for the current machine.

($name,$aliases,$type,$len,$thisaddr) = gethostbyname($hostname);

# This looks up numeric IP address infromation corresponding to the
# hostname where the server is expected to be running.
```

```
($name, $aliases,$type,$len,$thataddr) = gethostbyname($them);



# Create one endpoint of the communication link (our end).  See the Server
# code for a more complete explanation of what 'socket' does.  If we
# fail to create a socket, 'die' causes execution to terminate and
# display the reason ( stored in $_ ) for the failure.

# Remember that AF_INET and SOCK_STREAM are symbols provided by the
# "use Socket;" line.  Don't put $ in front of them!!

if (socket(S,AF_INET, SOCK_STREAM, $proto)) {
          print "Socket creation succeeded.\n";
}
else {
     die $!;
}

# 'bind' connects our socket (just created) to the specified port.  As
# in the Server code, the $sockaddr thing is just some magic that allows
# the parameters to 'bind' and (later) 'connect' to be smooshed together
# in the correct way.  For most cases this will be "cut and paste" code.
# 'this' gives information about who we are; 'that' refers to the server.

$sockaddr = 'S n a4 x8';
$this = pack($sockaddr, AF_INET, 0, $thisaddr);
$that = pack($sockaddr, AF_INET, $port, $thataddr);

if (bind(S, $this)) {
     print "Bind succeeded.\n";
}
else  {
    die $!;
}

# This is similar to the 'listen' in the server, except the server is
# waiting for a call and we're actively initiating the conversation.
# 'connect' connects our socket to the server's socket on the other end.

if (connect(S, $that)) {
     print "Connect succeeded.\n";
}
else {
     die $!;
}

# Force our socket to flush output immediately

select(S);
$| = 1;

# Make standard output the default again

select(STDOUT);
```

```
# The interaction we want is to type lines of input and have them echoed
# back by the server.  But how can we both wait for input AND be receptive
# to the server's output?   Answer:  By forking a process to accept input
# from standard input (the keyboard) and send it to the server and using
# our current process to receive and display input from the server.

if ($child = fork) {

      # We're the parent.  Read lines of input from the standard input and
      # send them to the server until end of file is seen.

      while (<STDIN>) {
            print S;
      }

      # Sleep for 3 seconds then...

      sleep 3;

      #...then kill ourselves and the child

      do dokill();
}
else {
      # We're the child.  Read lines of input from the server over the
      # socket S and output them.   Stop if end of file is seen.

      while (<S>) {
            print "Server: $_";
      }
}
```