

Beginning Perl Programming (X52.9543.001)**Handout 10: July 27, 2000*****Example of a socket client: webget.pl***

This code is from the PerlIPC man page in perlipc.pdf:

```
#!/App/Perl/bin/perl -w

use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host document ..."; }

$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;

foreach $document ( @ARGV ) {
    $remote = IO::Socket::INET->new(
        Proto      => "tcp",
        PeerAddr   => $host,
        PeerPort   => "http(80)",
    );
    unless ($remote) { die "cannot connect to http daemon on $host"; }
    $remote->autoflush(1);
    print $remote "GET $document HTTP/1.0" . $BLANK;
    while ( <$remote> ) { print; }
    close $remote;
}
```

relay.pl.template

```
#!/usr/bin/perl
#
# relay.pl.template -- HTTP/CGI Agent for connecting to the Interface Process
#
# This Perl5 script is a CGI program which serves http protocol requests,
# then acts which as a client to a long-lived process, called the
# "Interface Process". This technique allows preservation of state information,
# maintenance of database connections, and in-memory datastores which would
# not be feasible on a one-fork-per-page-hit basis.
#
# This script:
# (1) Checks if Interface Process is running
#     Note that this code does not actually launch the Interface Process.
# (2) Forwards the information in the http protocol request (CGI post/get
#     variables, environment variables) to the Interface Process port.
# (3) Echoes the response back to the HTTP client.
#
# Note that some of the code is more complex than necessary due to performance
# requirements: this code is compiled and executed on EVERY PAGE HIT!
# One of the issues, for example, is that this script is a -->TEMPLATE<--
# for the real perl script. The __XXXX__ 'macros' (which used to be read from
# a configuration file) need to be substituted for real values
# (eg by a sed script).
#
# This code was initially written by Mohamed Hendawi (moe@pobox.com) between
# July? and December 1996.
#
# This code was substantially overhauled by Clint Goss (clint@goss.com)
# in Feb 1997.
#

use Socket;
use FileHandle;

require "cgi-lib.pl";

# Configuration parameters set by our Makefile, for various instances
# of this client CGI.

my ($serverPort) = __SERVER_PORT__;

# Read in all CGI variables
&ReadParse (*input);

# Set auto-flush mode.

$| = 1;

# Determine the IP of the Interface Process host.
# It currently always runs locally.

my $remote = 'localhost';
my $iaddr = inet_aton ($remote) || &fatal_error ("no host: $remote");
```

```

# Pack the info with AF_INET for the connection.

my $paddr = pack_sockaddr_in ($serverPort, $iaddr);

# Fetch the TCP protocol number

my $proto = getprotobyname ('tcp');

# Open a stream communication endpoint and map it to the filehandle SOCK.

socket (SOCK, PF_INET, SOCK_STREAM, $proto) ||
    &fatal_error ("Cannot create socket: $!");

# Try connecting on the new socket

if (!connect (SOCK, $paddr)) {

    # If I can't connect to the server, then the server is probably
    # not running. This version of the code simply complains and exits.
    # A kinder, gentler solution would be to start the Interface Process
    # via a fork/exec.

    print "Content-type: text/html\n\n";
    print "<html>\n";
    print "<head><title>Application Not Running</title></head>\n";
    print "<body>\n";
    print "<h2>The application is not running</h2>\n";
    print "</body></html>\n";
    exit 0;
}

# If we got here, we know the server is running and we are connected to it.

# Set auto-flushing on the socket

autoflush SOCK 1;

# Create the message that is sent to the server. Server messages are
# a pair of lines (each terminated by a newline) and consist of
# "var value var value..." pairs. For example:
#
# ACTION SHOW_TREE OPEN_NODES 0 EXPAND_ALL 1
#
# Special characters are encoded by the enc() routine below.

my ($message) = &create_message (\%input, \%ENV);

print SOCK "$message\n";

# Read the response from the server

while (<SOCK>) {
    print "$_";
}

# All done!

exit 0;

# Formulate a message to the server

```

```

sub create_message {
    my ($cgi_vars, $env_vars) = @_;

    # A message is two lines. The first line is all of the environment
    # variables containing such variables as REMOTE_USER, etc..
    # The second line is all of the cgi vars

    my ($var_name, $message);

    # Environment variables

    foreach $var_name (keys (%{$env_vars})) {

        # Don't send the CGI vars twice
        next if ($var_name eq "QUERY_STRING");

        $message .= "$var_name " . &enc ($env_vars->{$var_name}) . " ";
    }

    $message .= "\n";

    # Now the CGI variables

    foreach $var_name (keys (%{$cgi_vars})) {

        # Remove CR's from input <textarea> values

        my ($val) = $cgi_vars->{$var_name};
        $val =~ s/\x0a//g;
        $message .= "$var_name " . &enc ($val) . " ";
    }

    return $message;
}

# Subroutine to handle serious problems

sub fatal_error {
    my ($mesg) = @_;

    print <<END;
Content-type: text/html

<html>
<head><title>Application - Fatal Error</title></head>
<body>
<h2>Application - Fatal Error!</h2>
<p>
<h3>$mesg</h3>
</body>
</html>
END

    exit 1;
}

```

```

# Encode an arbitrary binary string into a VERY limited set of characters.
#
# The output has only the characters: % + - . a-z A-Z 0-9
# This routine is useful for encoding many strings which can have embedded
# characters which are 'meta-characters' in some 'larger context':
#
# - Passing a string as an http URL, a Cookie, or a Location: reference.
#
# - Inserting a string into a data file which uses some 'delimiter' character
#   such as TAB or @.
#
# - Passing a string on a command line (through a shell) to another executable.
#
# The following transformations are done:
#
#   Input          Output
#
#   a-z A-Z 0-9   No change in these characters
#   - .           No change in these characters
#   (space)      +
#   all other     %HH where HH is 2 upper case characters 0-9, A-F

```

```

sub enc {
    my ($arg) = @_ ;

    # Convert existing %'s FIRST, so they don't get munged by later
    # transformations.

    $arg =~ s/\%\/\%25/g;

    # Protect plusses and map spaces onto plusses

    $arg =~ s/\+\/\%2B/g;
    $arg =~ s/\ /+/g;

    # Convert everything else in one swell foop

    $arg =~ s/([^\^a-zA-Z0-9%+.-])/'%' . sprintf ("%02X", ord($1))/ge;
    return $arg;
}

```

Makefile

```
...  
  
# Scrunch the oft-compiled CGI.  
  
relay.pl : relay.pl.template Makefile  
    echo "#!/perl" >relay.pl  
    echo "#THIS FILE IS COMPRESSED - SEE relay.pl.template" >>relay.pl  
    sed -n \  
        -e "s/^[ \t]*//" \  
        -e "s/___SERVER_PORT___/`./fetchAppParameter PORT`/" \  
        -e "s/___SERVER_ARGS___/\"\"/" \  
        -e "s/^[^#].*$$/&/p" \  
        relay.pl.template >>$@  
    chmod a+x $@
```

relay.pl

```
#!/perl  
#THIS FILE IS COMPRESSED - SEE relay.pl.template  
use Socket;  
use FileHandle;  
require "../sys/csbased/lib/cgi-lib.pl";  
my ($serverPort) = 29104;  
&ReadParse (*input);  
$| = 1;  
my $remote = 'localhost';  
my $iaddr = inet_aton ($remote) || &fatal_error ("no host: $remote");  
my $paddr = pack_sockaddr_in ($serverPort, $iaddr);  
my $proto = getprotobyname ('tcp');  
socket (SOCK, PF_INET, SOCK_STREAM, $proto) ||  
&fatal_error ("Cannot create socket: $!");  
if (!connect (SOCK, $paddr)) {
```

...

Application Server Code

```
#
# IntProcess.pm -- Interface Process main object
#
# Contains code to set up server, listen for requests on a port,
# and forward requests to User Interface (Ui) object.
#

package IntProcess;

...

# This "entry point" initiates operation of this Interface Process object
# as a server.
#
# This routine never returns.
# It sits in an indefinite loop, waiting for requests.
# These requests are typically initiated from a web server.
# Each request causes a spawn (fork-exec) of a child process.

sub start {
    my ($self) = @_;

    my ($server_desc) = "My Application";

    # Open the network protocol database, find the first entry for TCP,
    # and fetch the protocol number.
    # Note that, in a list context, getprotobyname() returns
    # the list ($name, $aliases, $protocol_number).
    # So the return value canNOT be placed in parentheses.

    my $proto = getprotobyname ('tcp');

    # Fetch the port.
    # If it is not a port number, then it must be a name and we
    # fetch the real port number from the network services database.

    my ($port) = "29104";
    if ($port !~ /^^\d+$/) {
        $port = getservbyport ($port, 'tcp');
    }

    print "Listening on port $port\n";

    # Open a socket, and attach the LISTEN filehandle to input
    # from the socket.

    socket (LISTEN, Socket::PF_INET, Socket::SOCK_STREAM, $proto) ||
        $self->debug_out_and_exit (1, "socket: $!\n");

    # Set the SO_REUSEADDR option on the socket to get around the
    # problem of "not being able to bind to a particular address while
    # the previous TCP connection on this port is still making up
    # its mind to shut down" (from "Programming Perl").

    setsockopt (LISTEN, Socket::SOL_SOCKET, Socket::SO_REUSEADDR, 1) ||
        $self->debug_out_and_exit (1, "setsockopt: $!\n");
}
```

```

# Compose a packed SOCKADDR_IN address from the TCP port number, and
# bind it to the already opened socket in the LISTEN filehandle.
# This call will fail if (typically) there is already an
# Interface Process listening on the same port.

bind (LISTEN, Socket::sockaddr_in ($port, Socket::INADDR_ANY)) ||
    $self->debug_out_and_exit (1, "bind: $!\n");

# Set the TCP queue size to 5 which is (probably) sufficient if we are
# spawning child processes on each request.

my ($queue_size) = 5;

# Tell the system that we're going to be accepting connections
# on the socket bound to the LISTEN filehandle and that
# the system can queue up to the number of waiting connections
# specified by $queue_size.

listen (LISTEN, $queue_size) ||
    $self->debug_out_and_exit (1, "listen: $!\n");

print "Interface Process is running...\n";

# Attempt to modify the argument area that the ps(1) program displays.
# (does not work on all operating systems, but it's
# the effort that counts, right?)

$0 = $server_desc;

# Loop accepting requests on our socket

my ($client_addr, $child_pid, $con);
for ($con = 1; ; $con++) {

    # Accept a connection from a client on the socket
    # bound to the LISTEN filehandle.
    # Our running process blocks (hangs) on this accept () call
    # until there is a connection request from a client.
    # When the accept returns, CLIENT is an open
    # filehandle attached to the newly made connection
    # to which we can write.

    $client_addr = accept (CLIENT, LISTEN);

    # Check for a failed accept. This can happen in
    # "routine and normal" processing if (for example)
    # the accept() call gets interrupted by a signal.

    if (! $client_addr) {

        # Complain if the error code is not one of the
        # "expected" error codes which result from an
        # interrupted system call.

        if (($! != EINTR) && ($! != ERESTART)) {
            print "Interface Process accept() failed " .
                "with unexpected error code $!\n";
        }
    }
}

```

```

        # ... in any case, loop back for another accept()
        next;
    }

    # Set automatic flush mode on our the CLIENT file handle -
    # causes a flush of the output after every write or print
    # to the CLIENT socket.

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

    # Show the name of the client who is connecting.

    if ($trace_client_connections) {
        my ($port, $iaddr) = Socket::sockaddr_in ($client_addr);
        my $name = gethostbyaddr ($iaddr, AF_INET);
        print "Connection from $name on port $port accepted\n";
    }

my ($forkOnRequest) = 1;

    if ($forkOnRequest) {

        # Fork off a child process to handle this client
        # request

        if (($child_pid = fork ()) == 0) {

            # Here only if we are the spawned
            # child process.

            # Close the LISTEN filehandle which is
            # bound to the socket. This is the same
            # filehandle which the parent will use
            # to accept another connection, so this
            # (probably) needs to be done to free up
            # the socket.

            close (LISTEN);

            # Service this connection request.
            # This is where all the "real" application
            # work is done.

            $self->service_connection_request ($con);

            # Close the filehandle we used to
            # read from and write HTML to the client.
            # Web browsers use this close() to determine
            # that no more data is coming from us,
            # so this close() is important.

            close (CLIENT);

            # We are the child, so exit with a "all-OK"
            # exit status.

            exit 0;
        }
    }

```

```

    } else {
        # No fork. Dangerous. Buggy.

        $self->service_connection_request ($con);
    }

    # Here only if we are the parent process.
    # Record the PID of the spawned child in a hash table.
    # When we get the SIGCHLD for the child process, we look
    # it up in this table.

    $self->{'CHILD_PIDS'}{$child_pid} = 1;

    # Some hand-waving here.
    # We're not really sure that this close() is needed, because
    # the child already closed this filehandle.
    # This close "probably" returns with an error status.
    # However, it might have been inserted in the early days
    # to handle some "situation".

    close (CLIENT);
}
}

# Service a connection request
sub service_connection_request {
    my ($self, $con) = @_;

    # Fetch the message from the client.
    # Note that, for requests originating from web page hits,
    # this is reconstituted version of the URL parameters
    # (or form parameters if the request method is POST)
    # together with an encoding of the environment variables
    # associated with the HTTP request.

    my ($client_mesg) = $self->get_client_message ();

    # Handle the message. This is where all the "real" application work
    # is performed.

    my ($result) = $self->handle_message ($client_mesg);
}

# Fetch the message for a pending client request.
# We read two \n-terminated lines from the socket,
# which we return as a single string partitioned by a single \n.
sub get_client_message {
    my ($self, $client_mesg);

    $client_mesg = <CLIENT>;
    $client_mesg .= <CLIENT>;

    chop ($client_mesg);
    return $client_mesg;
}

# Rip apart the incoming message and set up the anonymous hashes for the
# Environment data and the CGI variables.

```

```

sub parse_message {
    my ($self, $client_mesg) = @_;

    # Carve up the single client_mesg string into its
    # environment variable and CGI variable parts.

    my ($mesg_env, $mesg_cgi) = split (/\\n/, $client_mesg);

    # Build two new anonymous hashes

    my ($env_vars) = {};
    my ($cgi_vars) = {};

    # The format of each string is
    #   NAME-1 VALUE-1 NAME-2 VALUE-2 ... NAME-N VALUE-N
    # Each component is separated by a single space
    # Use perl's swift hash-initialization rules to populate our
    # new hashes.

    %{$env_vars} = split (/\\s/, $mesg_env);
    %{$cgi_vars} = split (/\\s/, $mesg_cgi);

    return ($env_vars, $cgi_vars);
}

# The big enchillada - handle a message

sub handle_message {
    my ($self, $client_mesg) = @_;

    my ($env_vars, $cgi_vars) = $self->parse_message ($client_mesg);

    my ($action) = $cgi_vars->{'ACTION'};

    # If we ain't got no ACTION, we better supply our own.

    if (! $action) {
        # A bit sneaky: Tack the FIRST_ACTION onto the end of the
        # client's message and simply re-parse it.

        my ($first_action) = "ACTION MAIN_PAGE";

        $client_mesg .= $first_action;
        ($env_vars, $cgi_vars) = $self->parse_message ($client_mesg);
    }

    # Decode vars

    my ($var_name);
    foreach $var_name (keys (%{$env_vars})) {
        $env_vars->{$var_name} = &Encode::dec ($env_vars->{$var_name});
    }

    foreach $var_name (keys (%{$cgi_vars})) {
        $cgi_vars->{$var_name} = &Encode::dec ($cgi_vars->{$var_name});
    }

    # ... Call a routine to produce the output for this web request

```

```
}  
  
# Generic complain-and-die routine  
  
sub debug_out_and_exit {  
    my ($self, $status, @args) = @_;  
  
    print @args;  
    exit $status;  
}  
  
1; #return true
```

This handout is Copyright © 2000 Clint Goss, All Rights Reserved.