

**Simple TCP Servers in Perl.****Aims.**

This worksheet introduces a simple server to experiment with a Web browser. It shows how **telnet** can be used to test the server. The final **Perl** script shows how to read a message from a socket, display the data and supply a response to the message.

#

**Socket Internet Data Structures.**

To enable us to use sockets we should also understand the various structures that hold information for the socket objects/functions. These structures are fairly complex and therefore have many functions available to manipulate them. **Perl** uses the same sort of socket functions as C! Here is a list of the structures underlying this worksheet.

**sockaddr.**

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

struct sockaddr {
    uint8_t      sa_len;
    sa_family_t  sa_family; /* Address family e.g. AF_xxxx value */
    char         sa_data[ 14]; /* Protocol specific data*/
};
```

This allows a standard data structure to be passed to functions/methods, which then use the `sa_family` to determine how to cope with the data.

**sockaddr\_in and in\_addr.**

```
#include <netinet/in.h>

struct sockaddr_in {
    uint8_t      sin_len; /* length of struct (16 bytes) */
    sa_family_t  sin_family; /* AF_INET */
    in_port_t    sin_port; /* 16-bit TCP or UDP port- */
                    /* number network-byte ordered */
    struct in_addr sin_addr; /* 32-bit IPv4 addr network- */
                    /* byte ordered */
    char         sin_zero[ 8]; /* unused always zero!*/
};

struct in_addr {
    in_addr_t    s_addr; /* 32_bit Ipv4 address network- */
                    /* byte ordered */
};
```

`sockaddr_in` is the 'internet socket address' structure. This is the actual structure that contains the port number and the host IP address, both in network byte order rather than the byte order used by the host machine. The actual IP address is held in a separate structure (for historical reasons!). This complicates things a bit because the host address can be referred to in two ways.

If we had a variable called `servaddr` to hold our socket internet data we would need

```
struct sockaddr_in servaddr;
```

The actual IP part of this structure may be referenced as

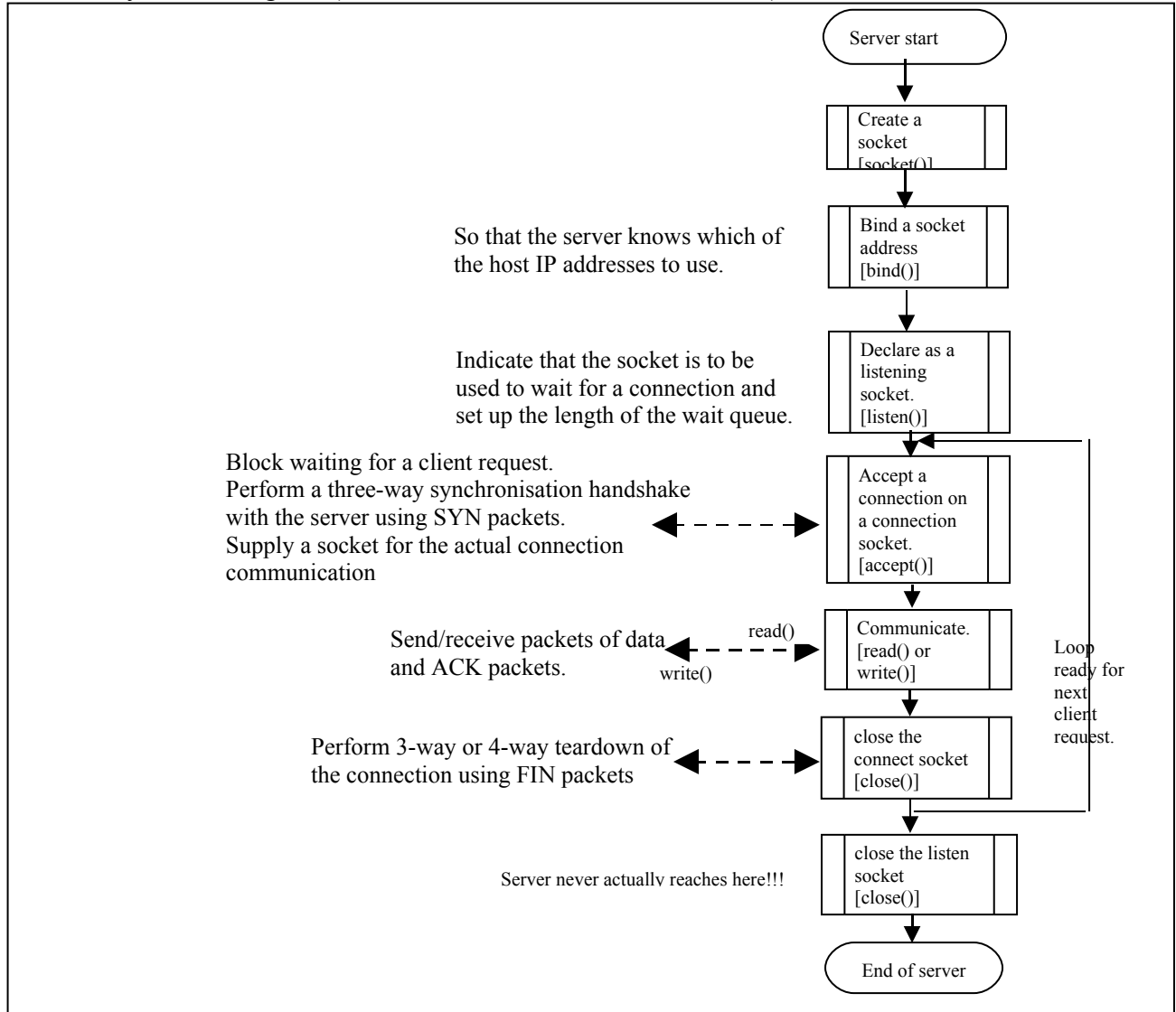
1. another structure (an *in\_addr structure*) by `servaddr.sin_addr`,
- or
2. a binary representation by the more complicated looking `servaddr.sin_addr.s_addr!`

Which method is used depends on the methods used for manipulation of the data and **Java** hides this from us!

Note that the *struct sockaddr\_in* has a member `sin_zero[]` which is unused and always has its elements cleared to zero. In fact, by convention, all the members of a variable of this type are always set to zero before setting the values actually wanted.

**A Simple TCP Server.**

The server code takes a standard form: *create a socket, bind to an IP address, listen for a connection to the server on this socket, accept the client connection, read/write data using the socket, and finally, close the socket.* The order in which these functions are used is shown by a flow diagram (the associated C functions are shown):



Note that:

A server must bind its IP address to the socket. If the server has more than one network interface card it can elect to bind to any of its available addresses.

Servers listen for connection requests and queues requests that it cannot deal with yet.

A server must *accept* a request from those queued by the *listen* function/method. It associates the accepted connection with a new socket descriptor.

Create a *PerlSockets* directory (use `mkdir`) to hold your **Perl** experiments. In this directory use **emacs** to enter the following server code, type:

```
emacs sockPerl.pl &
```

```
#!/usr/bin/perl

# sockPerl.pl a simple socket server Perl script!
# Based on script in 'HTTP The Definitive Guide',
#           by Gourley and Totty, publisher O'Reilly

use Socket;
use Carp;           # allow for error msg e.g. croak and die
use FileHandle;    # for socket and file handles

# use port 8000 by default, unless overridden by command line!
$port = (@ARGV ? $ARGV[ 0] : 8000);

# create local TCP socket, LISTNSOCK, and listen for connections
$proto = getprotobyname( 'tcp');
socket( LISTNSOCK, PF_INET, SOCK_STREAM, $proto) || die "Could not open
a socket port ", $port;

# set socket option to 'pinch' the socket if in use!
setsockopt( LISTNSOCK, SOL_SOCKET, SO_REUSEADDR, pack( "l", 1)) || die
"could not set socket option on port ", $port;
# bind to the default network interfac card!
bind( LISTNSOCK, sockaddr_in( $port, INADDR_ANY)) || die "Could not bind
port ", $port;
listen( LISTNSOCK, SOMAXCONN) || die; # max number of queued clients

while( 1) { # listen forever
  # show a listening message
  printf( "\n\n    <<<sockPerl accepting on port %d>>>\n\n", $port);

  # listen for connection on LISTNSOCK and service it on socket CONN
  $cport_caddr = accept( CONN, LISTNSOCK);
  ($cport, $caddr) = sockaddr_in( $cport_caddr);
  CONN -> autoflush( 1); # to ensure buffer gets sent?!?!

  # display who the connection is from
  $cname = gethostbyaddr( $caddr, AF_INET);
  printf( "\n    <<<Request from '%s'>>>\n", $cname);

  # Read msg from connection socket, CONN, until blank line.
  # Display the msg
  do {
    $line = <CONN>;
    print $line;           # display a line of the message

    $line =~ s/^\r//;      # get text up to CR or ...
    $line =~ s/\n//;      # ... LF
  } until( $line =~ /^$/); # until blank line
  close( CONN);          # flush the buffer and close the connection socket.
} # go accept any other connection.
```

‘Run’ the script using the **Perl** interpreter:

```
perl sockPerl.pl
```

This creates a local socket using the default port of 8000. A different port can be allocated at ‘run’ time, e.g. `perl sockPerl.pl 3456`. The ‘code’ just accepts a connection, reads and displays data from the socket, one line at a time, until a blank line is read.

Test the script by using **telnet** in another terminal, e.g.

```
telnet localhost 8000
```

Enter some lines of text. The running server should display them. When you've finished just enter a blank line and the server should close the connection. Exit from **telnet** in the usual way (*Ctrl-]* and *quit*). The **Perl** server must be closed by using *Ctrl-C*. You could try using **telnet** from another host to connect to the server.

Try using a Web browser, e.g **Firefox**, **Konqueror** or even **Internet Explorer**, to connect to the server. Just enter the address as:

```
http://localhost:8000/anything.      (use the correct hostname and port number!)
```

The server should display all the request header lines sent by the browser.  
Do all the browsers send the same information?

### **Interacting.**

Let's make the server do something a little more useful. The following lines will cause the server to interact with the administrator to set up a response message. Use **emacs** to add them before the `close( CONN)` statement in the script.

```
# prompt for response msg and get lines of response
# and send until a blank line.
printf( "\n    <<<Type a response (blank line to end)>>>\n");

do {
    $line = <STDIN>;
    $line =~ s/\r//;          # get text up to CR or ...
    $line =~ s/\n//;         # ... LF
    print CONN $line . "\r\n"; # concat CRLF to line and send
} until( $line =~ /^$/);    # until blank line
print "\n    <<<Sent entered text>>>\n";
```

An alternative way of 'running' the script is to make the file executable by changing the mode (or permission) bits, type:

```
chmod 755 sockPerl.pl
```

Check that the file is now executable by using the long form of **ls**:

```
ls -l sockPerl.pl
```

This should show that the file is executable by a sequence of leading characters in the listing:

```
-rwxr-xr-x sockPerl.pl .....
```

To execute the script just type:

```
./sockPerl.pl
```

Again test it out with **telnet** or a Web browser.

At the server type in some text and end it with a blank line, e.g. try some html;

```
<html><body><h1>
Hello from me!
</h1></body></html>
```

Don't forget the blank line!

Does the browser manage to render the text in a header level one font?

### **Sending Files.**

One further little thing we can try is to allow the server to respond with the contents of a text file. Use **emacs** to create the file:

```
emacs data.txt &
```

Enter some text, you could use the html shown above!

Again use **emacs** to edit the *sockPerl.pl* file.

Add the following lines before the line `close( CONN) .`

```
# Now try to send contents of a text file!
if( !open( DATAF, "< data.txt")) {
    print "could not open text file\n";
    print CONN "could not open file\r\n";
}
else {
    print "\n    <<<Now sending contents of test file>>>\n";
    while( $line = <DATAF>) {
        print $line;
        $line =~ s/\r//;           # get text up to CR or ...
        $line =~ s/\n//;         # ... LF
        print CONN $line . "\r\n"; # concat CRLF to line and send
    }
}
printf( "\n    <<<Sent text file>>>\n");
```

This opens the *data.txt* file and sends it line by line.

Run the **Perl** script and test it using **telnet** and different browsers.