

Simple TCP Server and Client in Tcl.**Aims.**

This worksheet introduces a simple client and an echo server to experiment with. The server also used to experiment with a Web browser. It shows how **telnet** can be used to test the server. The final **Tcl** server script shows how to read a message from a socket, display the data and supply a response to the message. The **Tcl** client shows how to interact with a user and send data to a server.

#

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. **Tcl** 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

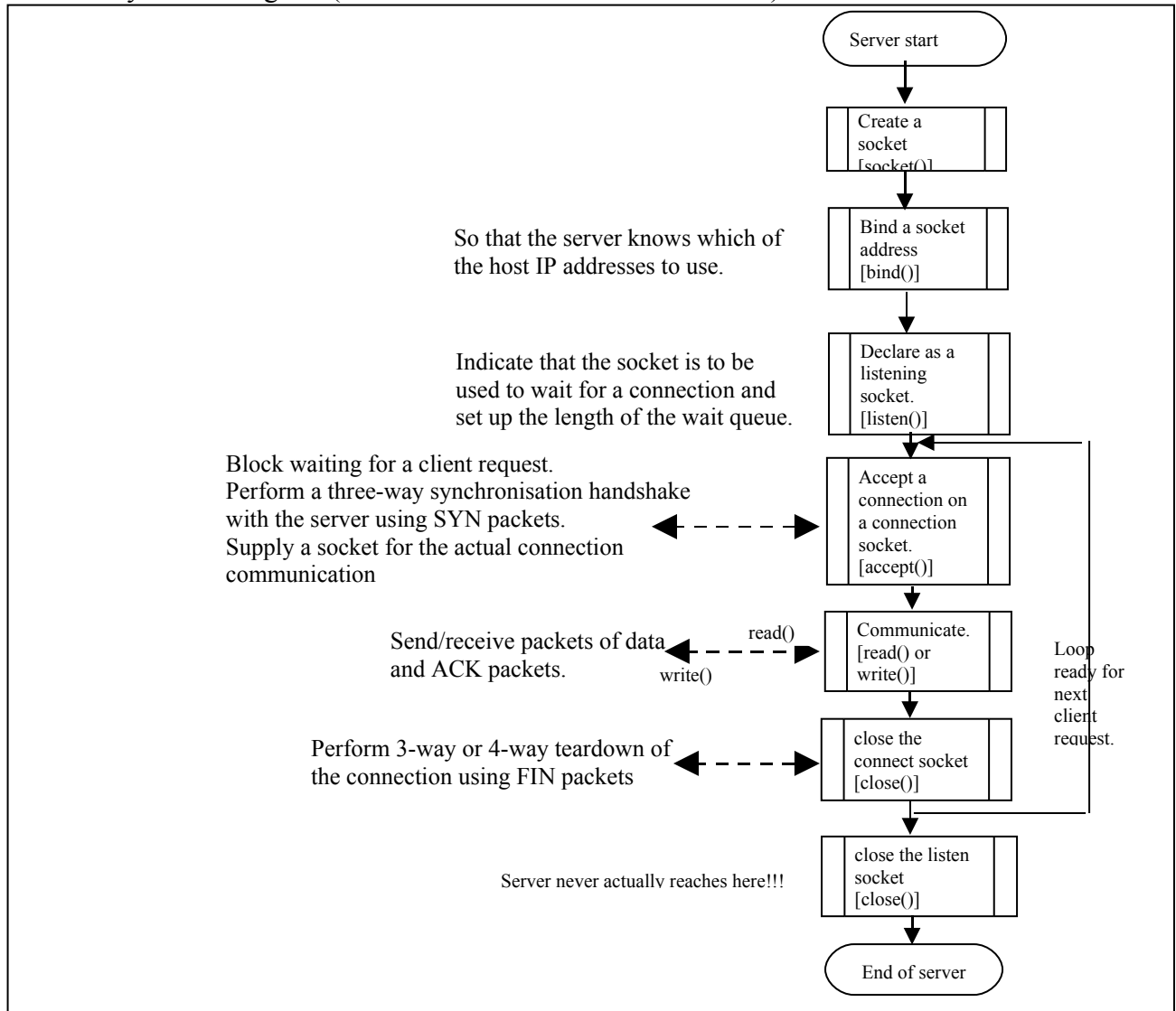
1. another structure (an *in_addr* structure) by `servaddr.sin_addr`,
- or
2. a binary representation by the more complicated looking `servaddr.sin_addr.s_addr`!

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 *TclSockets* directory (use `mkdir`) to hold your **Tcl** experiments. In this directory use **emacs** to enter the following server code, type:

```
emacs echo_server.tcl &
```

```

#!/usr/bin/tclsh
#
# Based on examples in 'Practical Programming in Tcl and Tk'
#                               by Brent B. Welch
#
# Callback procedure/function to handle socket requests from clients.
# When an initial request is made this procedure is passed:
#   connSock: the socket on which the server 'talks' to the client.
#   connAddr: the IP addr of the client.
#   connPort: the port number the client is using.
#
proc Accept {connSock connAddr connPort} {
    global requestLine ""
    set running 1
    set msgType ""
    set filePath ""
    set proto ""
    puts "\n\tConnection on socket $connSock, from $connAddr on port $connPort\n"
    fconfigure $connSock -buffering line
    while {$running} {

        if {[eof $connSock] } { ;# socket closed before blank line read!
            puts "\n\tClient finished early, closing socket!\n"
            set running 0
        } else {
            # read the requestLine
            catch {gets $connSock requestLine}
            if {[string compare $requestLine ""] == 0} {
                # blank line is end of header fields or end of connection
                set running 0
            } else { ;# process requestLine
                puts $requestLine
                puts $connSock $requestLine
                if { [string last "HTTP" $requestLine] > 0} {
                    # break header line into component strings
                    scan $requestLine "%s %s %s" msgType filePath proto
                    # puts "$msgType $filePath $proto"
                } ;# processed first header line
            } ;# done processing requestLine
        } ;# catch requestLine
    } ;# end while loop

    close $connSock
    puts "\n\tClosed connection to $connAddr"
    puts "\t\tWaiting a new connection\n"
}

# The main 'program' opens a socket and allocates ...
# ... a callback procedure.
# Then loops forever waiting on the event of a client request.

set port 3456
puts "\n\t<<<Opening socket>>>\n"
set sock [socket -server Accept $port]
puts "\t\t<<<Waiting connection>>>"
set forever 1
vwait forever

```

'Run' the script using the **Tcl shell**:

```
tclsh echo_server.tcl
```

This creates a local socket using the default port of 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. Each line is echoed back to the sender. **Tcl** works differently to other systems in that the *socket function* not only creates a socket, but allocates a function to be called when a connection is made – the callback function, in this case it is called 'Accept', and it performs all the work!

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

```
telnet localhost 3456
```

Enter some lines of text. The running server should display them and send them back. When you've finished just enter a blank line and the server should close the connection. If necessary, exit from **telnet** in the usual way (*Ctrl-]* and *quit*). The **Tcl** 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:3456/anything.      (use the correct hostname and port number!)
```

The server should display all the request header lines sent by the browser (and send them back!). Do all the browsers send the same information? Do they all display the repond in the same way?

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;

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 html as shown:

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

Again use **emacs** to edit the *echo_server.tcl* file.

Add the following lines before the line `close $connSock.`

```

if { [file exists $filePath]} {
    puts $connSock "HTTP/1.1 200 OK"
    puts $connSock "Content-Type: text/plain"
    puts $connSock "Connection: close"
    puts $connSock "Content-Length: [file size $filePath]"
    puts $connSock ""
    set inFile [open $filePath]
    fcopy $inFile $connSock
} else {
    set errMsg "<html><body><h1>Not found </h1>$filePath</body></html>"
    puts $connSock "HTTP/1.1 404 NotFound"
    puts $connSock "Connection: close"
    puts $connSock "Content-Length: [string length $errMsg]"
    puts $connSock ""
    puts $connSock $errMsg
}
unset msgType
unset filePath
unset proto

```

This uses the *filePath* obtained from the first request line sent to it and attempts to open the file. If the file exists it is sent line by line, with an appropriate HTTP header, otherwise an error message is sent.

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

```
chmod 755 echo_server.tcl
```

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

```
ls -l echo_server.tcl
```

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

```
-rwxr-xr-x echo_server.tcl .....
```

To execute the script just type:

```
./echo_server.tcl
```

Run the **Tcl** script and test it using **telnet**. Supply a header line such as:

```
GET /home/netlab/myname/TclSockets/data.txt HTTP/1.1
```

Don’t forget to alter *myname* to your own user name and don’t forget to **send a blank line!**

Test the server using different browsers. Try an address such as:

```
http://localhost:3456/home/netlab/myname/TclSockets/echo_server.tcl
```

Does it make any difference to the browser if you comment out the echo line?

i.e. line puts \$connSock \$requestLine

A simple TCP Client.

For completeness here is a simple client that takes command line arguments to specify a host and port number. It interacts with the user to get message lines to send to the server. Try sending GET /home/netlab/myname/data.txt HTTP/1.1 as you did in **telnet**.

```
#!/usr/bin/tclsh

# if any command line args use list index to set host:port
if { $argc > 0 } {
  if { $argc < 3 } {
    set host [lindex $argv 0]
    set port [lindex $argv 1]
  }
  else {
    set host 127.0.0.1
    set port 3456
  }
}
puts stdout "\n\nUsing:\n\tHost $host\n\tPort $port\n\n"

set sock [socket $host $port]
configure $sock -buffering line

set running 1
while { $running } {

  puts "Enter text (blank line to end): "
  set message [gets stdin]
  puts $sock $message

  set reply [gets $sock]
  puts $reply

  if { [string compare $message ""] == 0 } {
    set running 0
  }
} ;# end while running
exit
```

Notice that the client does **not** need an accept callback procedure.

This client does not seem to work completely! Can you see why not? Can you alter it so that it does display the returned file?