FIT3142 Laboratory #6: Client-Server Applications Using Socket Inter-Process Communications Part 2

Dr Carlo Kopp, SMIEEE, AFAIAA, FLSS, PEng
Faculty of IT, Clayton.
Email: `Carlo.Kopp@monash.edu`

August 29, 2016

Revision Status:

```
$Id: FIT3142-Laboratory-6.tex,v 1.4 2016/08/29 06:26:37 carlo Exp carlo $
```

Contents

1 Introduction

The objective of this weeks laboratory exercise is to gain some further experience with the BSD Socket networking API, using the C language.

It is strongly recommended that you prepare for the lab by reading through all of the reference materials and coding and testing the application before you attend the lab, because the time required to complete the lab tasks is only sufficient if you have actually prepared for the lab. Do not paste from example code and tutorials as you will not learn how the code actually works if you so so. If you attend the Laboratory without any preparation it is unlikely you will be able to complete the work on time.

2 Lab Session 2

The completion of the following steps will meet the objectives of this lab.

1. Using the client and server code from Laboratory #5 and the source code for `status-shm-server.c` in Laboratory #3, replicate the functionality of the client and server programs you developed in Laboratory #3, using a Socket IPC channel instead of shared memory.
2. You must create the source files and your own executables for `MyID-socket-client.c` and `MyID-status-socket-server.c`.
3. Demonstrate your code, and then email your completed lab exercise to your lab demonstrator. The email subject must contain the unit code `FIT3142` and the week.
4. **Save all of your program files as you may need them in a future lab.**

3 Hints and Marking Guide

3.1 Implementation Hints

While completing this lab you will be marked on how you have chosen to implement your solution. Specifically, your server should not wait (block) for input from the client and you have some freedom in how you transfer the data within the `SEG_DATA` struct from the server to the client. The following describes how you may implement these solutions, with references to useful functions that were discussed previously in the network reference tutorial provided with Lab 5. Additional reading on the various library and system calls in C language can be found in Curry's book.

Preventing Blocking

The status server is expected to continually run through its main loop, updating the status values every 1 second. However, the `recv()` function causes the server to block and wait for the client to `send()` input. If the server is to regularly update the status values as intended then we need it to only attempt to `recv()` only if the client has sent something. The `select()` function allows us to inspect file descriptors (such as the descriptors used by `bind()` and `accept()`) and test if any file descriptors have a connection pending or a message that the server needs to `recv()`. Using `select()` in this manner allows the server to only execute a `recv()` if the client has executed a `send()`.

Sending the `struct`

Unlike the previous shared memory system, the server will need to send the entire contents of the `SEG_DATA` struct to the client whenever it is requested. However, the client and server can only communicate via the transmitted character buffer. There are several different methods in which the struct data can be communicated.

1. A very simple solution is to send each integer in the struct individually, with the server issuing 10 `send()`'s and the client having 10 `recv()`'s. Each single integer should be converted to network endianness with `htonl()` when sending and restored to host endianness with `ntohl()` when receiving. Alternatively, the data from the struct can be serialised, or packed, into a single `char` buffer, which is sent in a single `send()`. The receiver then unserialises the data from the buffer back into a `struct`.

2. An easy way to pack the struct into a buffer is to use `sprintf()` to print the values from the struct into a character buffer. This buffer would be unpacked by using `strtok()` and `strtol()` to split the `char` buffer into its integer values and convert these values back to integers that can be placed into a `struct`. This method is fairly straightforward

to implement and is also easy to test, since the transmitted buffer is in a human readable format, but produces a larger than necessary buffer.

3. A better way to pack the struct is take each of the 10 integers in the `struct`, split (by shifting) each integer into its corresponding bytes and write each byte sequentially into the buffer. Unpacking consists of taking each group of four bytes, shifting and OR-ing them back into integers, which are then put back into a struct. This method produces the most compact messages, but the packed messages are difficult to debug and will require a higher skill level in C programming.

3.2 Marking Guide

You will be marked according to the way in which you implement your solution, with increasing marks for the more difficult implementations:

1. Code is well commented (10%)
2. Code is compiled with `-Wall` option and produces no warnings (10%)
3. Return values of networking functions checked for errors (10%)
4. Client and server function correctly, as in Lab 3 (20%)
5. Server does not block while client waits for user input (20%)
6. Information from the server is packed into a single transmission (10%)
7. Information is packed byte by byte (20%)

4 Server and Client Function

```
deathstar[carlo]1223% status-socket-server

STATUS DUMP
UP Status        = 0
Exit Status      = 0
RPM              = 3400
Crank Angle      = -1
Throttle Setting = 69
Fuel Flow        = 49
Engine Temp      = 79
Fan Speed        = 29
Oil Pressure     = 69
Waiting for client

STATUS DUMP
UP Status        = 0
Exit Status      = 0
RPM              = 3300
Crank Angle      = -2
Throttle Setting = 68
Fuel Flow        = 48
Engine Temp      = 78
Fan Speed        = 28
Oil Pressure     = 68
Waiting for client

STATUS DUMP
UP Status        = 0
Exit Status      = 0
RPM              = 3200
Crank Angle      = -3
Throttle Setting = 67
Fuel Flow        = 47
Engine Temp      = 77
Fan Speed        = 27
Oil Pressure     = 67
Waiting for client

UP Status        = 0
Exit Status      = 0
```

```
RPM              = 3100
Crank Angle      = -4
Throttle Setting = 66
Fuel Flow        = 46
Engine Temp      = 76
Fan Speed        = 26
Oil Pressure     = 66
Waiting for client

STATUS DUMP
UP Status        = 0
Exit Status      = 0
RPM              = 3000
Crank Angle      = -5
Throttle Setting = 65
Fuel Flow        = 45
Engine Temp      = 75
Fan Speed        = 25
Oil Pressure     = 65
Waiting for client

STATUS DUMP
UP Status        = 0
Exit Status      = 0
RPM              = 2900
Crank Angle      = -6
Throttle Setting = 64
Fuel Flow        = 44
Engine Temp      = 74
Fan Speed        = 24
Oil Pressure     = 64
Waiting for client

STATUS DUMP
UP Status        = 0
Exit Status      = 1
RPM              = 2800
Crank Angle      = -7
Throttle Setting = 63
Fuel Flow        = 43
Engine Temp      = 73
Fan Speed        = 23
Oil Pressure     = 63
```

```
Waiting for client
Task completed
deathstar[carlo]1224%


deathstar[carlo]1025% MyID-socket-client

CLIENT STATUS DUMP
RPM              = 3200
Crank Angle      = -3
Throttle Setting = 67
Fuel Flow        = 47
Engine Temp      = 77
Fan Speed        = 27
Oil Pressure     = 67
Enter Command (1 to exit, 0 to continue): 0


CLIENT STATUS DUMP
RPM              = 3000
Crank Angle      = -5
Throttle Setting = 65
Fuel Flow        = 45
Engine Temp      = 75
Fan Speed        = 25
Oil Pressure     = 65
Enter Command (1 to exit, 0 to continue): 1

Task completed
deathstar[carlo]1026%
```

5 Notes

The programming task can be done directly on any `Linux`, `*BSD/MacOSX`, or `Unix` system.

If you do not have access to any of these platforms, several options are available:

1. The trial version of `Ubuntu Linux` is available (`http://www.ubuntu.com/download/help/try-ubuntu-before-you-install`);
2. You can install `CygWin` under MS Windows (`http://cygwin.com/install.html`);
3. An alternate approach is to configure an external USB hard disk as a bootable `Linux system` on your `MS Windows` system;
4. You can install `VMWare Workstation` and install any `Linux` or `*BSD` variant as a VM on your `MS Windows` system.

Please note that `Linux/Unix` skills are frequently required in many industry jobs. Most large web servers and distributed applications are not hosted on `MS Windows` platforms.

There are many online tutorials for BSD socket programming, with working examples of code. Also the `nttcp` application you will use in a future Laboratory is a socket application. Useful tutorials are:

1. Tutorials on Socket programming (Ch.11) and file I/O (Ch.3) `http://www.bitsinthewind.com/about-dac/publications/using-c-on-the-unix-system`;
2. Tutorials on Socket programming `http://beej.us/guide/bgnet/`