CIS–3152 Lab
Concurrent TCP Servers

Vermont Technical College

Last Revised: January 10, 2014

Introduction

In this lab you will write a concurrent TCP server using the POSIX sockets API. Your server will implement the simple echo protocol described in RFC–862. You should review that RFC (it is very short). In particular, the server should accept lines of text from the client and then return those same lines to the client exactly as received. This should continue until the client closes the connection.

Despite the simplicity of this protocol it actually has uses. It can be valuable as a debugging aid or for network timing and performance measurements. An echo client and server can also be used as a starting point for implmenting more complex protocols.

In this lab it is also essential that the final server be able to support multiple clients with no particular limit on the number of simultaneous clients. However, no support for timeouts on either the server or client side are required for this lab; it is acceptable if both the client and server may potentially wait "forever" for a response from the other side.

1 The Client

The client should take the IP address and port number of the server on the command line. Use a default port number of 7 if the user does not provide one (see RFC–862). The client should establish a connection with the server and then enter a loop where it accepts lines of text from the user and sends those lines to the server one at a time. It should display the server's response to each line, presumably a copy of the line sent. The user specifies when a session is over by entering an "end-of-file" indication at the client terminal (control+D). In response the client closes the connection to the server and exits.

Much of the client's behavior is similar to the simple daytime client discussed in class and used in Lab #1. *After making a suitable branch* in your sample code repository, I recommend that you copy the entire daytime sample to create a starting point for this lab. You can copy the client and server source files, along with all the Code::Blocks configuration information by going to the `daytime` folder in `cis-3152` and issuing the commands

```
$ mkdir -p ../echo/C
$ cd C
$ tar cf - . | (cd ../../echo/C; tar xf -)
```

This uses tar to copy an entire directory hierarchy from one place to another. After doing this I recommend that you rename `daytimeC.workspace` to `echoC.workspace`. Note that the copy operation above will also copy the threaded version of the daytime server. That version implements concurrency using POSIX threads. It is not directly useful to you in this lab since we'll be using the more traditional approach of forking child processes, but having the

threaded server in your workspace does no harm and you might find it useful later.

You may wish to add and commit the `echo/C` directory and all its contents to your Git repository (be sure you are working in a branch). You can then load the Code::Blocks workspace in the `echo/C` directory to begin working on your system.

2 The Server

Start by modifying the daytime server to implement the echo protocol for a single client at a time (in other words implement an iterative server first). Once you have that working you can then modify it to provide support for multiple, simultaneous clients.

The server program should follow the structure of a concurrent server as described in class. In particular, it should fork to handle each client. Eventually it should also properly clean up its children by catching the SIGCHLD signal and calling `waitpid` in the corresponding signal handler. However, in your server do not take this step initially so that you can observe the zombies created by improper handling of SIGCHLD as described below.

To illustrate the support for concurrency do the following:

1. Modify the server so that the children include their process ID numbers in each line echoed to the client. You should probably also have each child display its process ID number on the server's terminal when it starts servicing a client. Use the `getpid` system call to obtain the process ID number.
2. Connect to your server several times simultaneously and observe that a new process is created for each connection. Note the process ID numbers of the children.
3. As each client is serviced note the process ID numbers sent to the clients and verify that they match.
4. Terminate each connection and then use the `ps aux` command to view the process list and verify that the child processes created above are now zombies. This is due to the parent server failing to recover the exit status of the terminated children.
5. Change the server to eliminate the zombies by setting up a signal handler for SIGCHLD and calling `waitpid` inside that signal handler. Refer to class notes, the text, or on-line resources for more details. Repeat the experiment above to be sure no zombies are created.

3 Optional

There are several ways your server program could be improved to make it more professional. *You do not need to implement any of these features* but if you have time and would like to explore I invite you to consider the following.

- The server should write a log file that records the date and time when each client connects as well as the IP address of the client. Where in the program should this be done so as to avoid corrupting the log file in the case when two clients connect simultaneously?
- Currently the server must be explicitly run in the background by typing an & character at the end of the command line. Professional server programs don't require this and can "demonize" themselves by dissociating themselves from their controlling terminal. Find out how this works and implement it.

4 Report

Write up a report for this lab using the LaTeX typesetting system. Your report should describe how your client and server software work and report on your observations of their behavior.