CSE 533 (Fall 2015) - Assignment 1

Due date: Wednesday, September 30, at 11:55pm

Overview

The aim of this assignment is to develop *TCP* socket based client and server programs using I/O multiplexing, child processes and threads. It also aims at getting you to familiarize yourselves with several other network programming techniques such as the operation of the *inetd* superserver daemon, using the '*exec*' family of functions, handling various socket error scenarios, setting some socket options, and calling some basic domain name / *IP* address conversion functions.

Apart from the materials in Chapters 1 to 6 covered in class, you will also need to refer to the following reading materials, mainly from the textbook:

- The *exec* family of functions (Section 4.7 of Chapter 4)
- Using pipes for interprocess communication (IPC) in *Unix*. See file `pipes.pdf` in folder
- *gethostbyname* & *gethostbyaddr* functions (Sections 11.3 & 11.4, Chapter 11)
- *setsockopt* function & *SO_REUSEADDR* socket option (Section 7.2 & *pp*.210-213, Chapter 7)
- The basic structure of *inetd* (Section 13.5, Chapter 13)
- Programming with threads (Sections 26.1 to 26.5, Chapter 26)
- Error scenarios induced by process terminations & host crashes (Sections 5.11 to 5.16, Chapter 5)

You are to work on this assignment individually, not in groups.

Client

Command line arguments processsing - The client is evoked with a command line argument giving either the server *IP* address in dotted decimal notation, or the server domain name. The client has to be able to handle either mode and figure out which of the two is being passed to it. If it is given the *IP* address, it calls the *gethostbyaddr* function to get the domain name, which it then prints out to the user in the form of an appropriate message (*e.g.*, 'The server host is compserv1.cs.sunysb.edu'). The function *gethostbyname*, on the other hand, returns the *IP* address that corresponds to a given domain name. In this case, the IP address should be printed out to the user in an appropriate message.

The client parent - The client then enters an infinite loop in which it queries the user what service the user would like to request. There are two services: ***echo*** and ***time*** (note that *time* is a slightly modified version of the *daytime* service – see below). After the user requests a service, the client forks off a child process to handle this request. The parent process on the other hand, enters a second loop in which it continually reads and prints out status messages received from the child via a half-duplex pipe (see below). The parent exits the second loop when the child closes the pipe (how does the parent detect this?), and/or the SIGCHLD signal is generated when the child terminates. The parent then repeats the outer loop, querying the user again for the (next) service s/he desires. This cycle continues till the user responds to a query with ***quit*** rather than ***echo*** or ***time***.

The client child - The child process is the one which handles the actual service for the user. It *exec*s (see Section 4.7, Chapter 4) an ***xterm*** to generate a separate window through which all interactions with the server and the user take place. For example, the following *exec* function call evokes an *xterm*, and gets the *xterm* to execute *echocli*, located in the current directory, passing the string 127.0.0.1 (assumed to be the *IP* address of the server) as the command line argument *argv*[1] to *echocli* (see file `exec.pdf` in folder) :

execlp("xterm", "xterm", "-e", "./echocli", "127.0.0.1", (char *) 0)

xterm executes one of two client programs (*echocli* or *timecli*, say) depending on the service requested. A client program establishes a *TCP* connection to the server at a 'well-known port' for the service (in reality, this port will, of course, be some ephemeral port of your choice, the value of which is known to both server and client code). All interaction with the user, on one hand, and with the server, on the other hand, takes place through the child's *xterm* window, not the parent's window. At the same time, the child will use a half-duplex pipe to relay status information to the parent which the parent prints out in its window (see below).

Service request termination and program robustness - To terminate the *echo* client, the user can type ^D (CTRL D, the EOF character). To terminate the *time* client, the only option is for the user to type in ^C (CTRL C). (This can also be used as an alternative means of terminating the *echo* client.) Note that using ^C in the context of the *time* service will give the server process the impression that the client process has 'crashed'. It is your responsibility to ensure that the server process handles this correctly and closes cleanly. We shall address this further when discussing the server process.

It is also part of your responsibility in this assignment to ensure that the client code is robust with respect to the server process crashing (see Sections 5.12 & 5.13, Chapter 5). Amongst other implications, this means that it would probably be a good idea for you to implement your *echo* client code along the lines of either : Figure 6.9, *p*.168 (or even Figure 6.13, *p*.174) which uses I/O multiplexing with the *select* function; or of Figure 26.2, *p*.680, which uses threads; rather than along the lines of Figure 5.5, *p*.125.

IPC using a pipe - When the child terminates, either normally or abnormally, its xterm window disappears instantaneously. Consequently, any status information that the child might want to communicate to the user should not be printed out on the child's *xterm* window, since the user will not have time to see the final such message before the window disappears. Instead, as the parent forks off the child at the beginning, a half-duplex pipe should be established from child to parent. The child uses the pipe to send status reports to the parent, which the parent prints out in its window. For example, when the child can not establish a connection with the server, a status report should be sent back to the parent. Other error conditions that occur when interacting with the server should also be reported to the parent via the pipe.

More robustness - In general, you should try to make your code as robust as possible with respect to handling errors, including confused behaviour by the user (*e.g.*, passing an invalid command line argument; responding to a query incorrectly; trying to interact with the service through the parent process window, not the child process *xterm*; *etc*.). Amongst other things, you have to worry about *EINTR* errors occurring during slow system calls (such as the parent reading from the pipe, or, possibly, printing to *stdout*, for example) due to a *SIGCHLD* signal.

Server

Multi-threading and server services - The server handles multiple (potentially concurrent) clients **using threads** (specifically, **detached** threads), **not** child processes (see Sections 26.1 to 26.4, Chapter 26). It can handle multiple types of service; in our case, two: *echo* and *time*. *echo* is just the standard echo service that we have seen in class. *time* is a slightly modified version of the *daytime* service (see Figure 1.9, *p*.14): instead of sending the client the 'daytime' just once and closing, the service sits in an infinite loop, sending the 'daytime', sleeping for 5 seconds, and repeating, *ad infinitum*.

Relation to Inetd superserver - The server is **loosely** based on the way the *inetd* daemon works: see Figure 13.7, *p*.374. However, note that the differences between *inetd* and our server are probably more significant than the similarities: *inetd* forks off children, whereas our server uses threads; *inetd* child processes issue *exec* commands, which our server threads do not; *etc*. So you should treat Figure 13.7 (and Section 13.5, Chapter 13, generally) as a source of ideas, not as a set of specifications which you must slavishly adhere to and copy. Note, by the way, that there are some similarities between our **client** and *inetd* (primarily, forking off children which issue *exec*s), which could be a useful source of ideas.

Listening on multiple services - The server creates a listening socket for each type of service that it handles, bound to the 'well-known port' for that service. It then uses *select* to await clients (Chapter 6; or, if you prefer, *poll*; note that *pselect* is not supported in Solaris 2.10). The socket on which a client connects identifies the service the client is seeking. The server *accept*s the connection and creates a thread which provides the service. The thread detaches itself. Meanwhile, the main thread goes back to the *select* to await further clients.

Thread safety - A major concern when using threads is to make sure that operations are **thread safe** (see *p*.685 and on into Section 26.5). In this respect, Stevens' *readline* function (in Stevens' file *unpv13e/lib/readline.c*, see Figure 3.18, *pp*.91-92) poses a particular problem. On *p*.686, the authors give three options for dealing with this. The third option is too inefficient and should be discarded. You can implement the second option if you wish. Easiest of all would be the first option, since it involves using a thread-safe version of *readline* (see Figures 26.11 & 26.12) provided in file *unpv13e/threads/readline.c*. Whatever you do, remember that Stevens' library, *libunp.a*, contains the non-thread-safe version of Figure 3.18, and that is the version that will be link-loaded to your code unless you undertake explicit steps to ensure this does not happen (*libunp.a* also contains the 'wrapper' function *Readline*, whose code is also in file *unpv13e/lib/readline.c*). Remaking your copy of *libunp.a* with the 'correct' version of *readline* is not a viable option because when you hand in your code, it will be compiled and link-loaded with respect to the version of *libunp.a* in the course account, *~cse533/Stevens/unpv13e_solaris2.10* (I do not intend to change that version since it risks creating confusion later on in the course). Also, you will probably want to use the original version of *readline* in the client code anyway. We are providing you with a sample *Makefile* (see file `Makefile` in folder) which picks up the thread-safe version of *readline* from directory *~cse533/Stevens /unpv13e_solaris2.10/threads* and uses it when making the executable for the server, but leaves the other executables it makes to link-load the non-thread-safe version from *libunp.a*.

Robustness - Again, it is part of your responsibility to make sure that your server code is as robust as possible with respect to errors, and that the server threads terminate cleanly under all circumstances. Recall, that the client user will often use ^C (CTRL C) in the *xterm* to terminate the service. This will appear to the server thread as if the client process has crashed. You need to think about the error conditions that will be induced (see Sections 5.11 to 5.13, Chapter 5), and how the *echo* and *time* server code is to detect and handle these conditions. For example, the *time* server will almost certainly experience an *EPIPE* error (see Section 5.13). How should the associated *SIGPIPE* signal be handled?

What about errors other than *EPIPE*? Which ones can occur? How should you handle them? For example, if a thread terminates without explicitly closing the connection socket it has been using, the connection socket will remain existent until the server process itself dies (why?). Since the server process is supposed, in principle, to run forever, you risk ending up with an ever increasing number of unused, 'orphaned' sockets unless you are careful.

Time server implementation - The *time* server should use the *select* function. On one hand, *select*'s timeout mechanism can be used to make the server sleep for the 5 seconds. On the other hand, *select* should also monitor the connection socket read event because, when the client *xterm* is ^C'ed, a *FIN* will be sent to the server *TCP*, which will prime the socket for reading; a read on the socket will then return with value 0 (see Figure 14.3, *p*. 385 as an example).

Proper status messages at the server - Whenever a server thread detects the termination of its client, it should print out a message giving appropriate details: *e.g*., "Client termination: *EPIPE* error detected", "Client termination: socket read returned with value 0", "Client termination: socket read returned with value -1, errno = . . .", and so on.

***SO_REUSEADDR* socket option** - When debugging your server code, you will probably find that restarting the server very shortly after it was last running will give you trouble when it comes to *bind* to its ' well-known ports'. This is because, when the server side initiates connection termination (which is what will happen if the server process crashes; or if you kill it first, before killing the client) *TCP* keeps the connections open in the *TIME_WAIT* state for 2*MSL*s (Sections 2.6 & 2.7, Chapter 2). This could very quickly become a major irritant. I suggest you explore the possibility of using the *SO_REUSEADDR* socket option (*pp*.210-213, Chapter 7; note that the *SO_REUSEPORT* socket option is not supported in Solaris 2.10), which should help keep the stress level down. You will need to use the *setsockopt* function (Section 7.2) to enable this option. Figure 8.24, *p*.263, shows an instance of server code that sets the *SO_REUSEADDR* socket option.

Nonblocking *accept* - Finally, you should be aware of the sort of problem, described in Section 16.6, *pp*.461-463, that might occur when (blocking) listening sockets are monitored using *select*. Such sockets should be made nonblocking, which requires use of the *fcntl* function (see file `fcntl.pdf` in folder) after *socket* creates the socket, but before *listen* turns the socket into a listening socket.

Environment / platform

Your assignment program must execute correctly on the Solaris 10 *compserv* (not the Linux *compute*) nodes in the *cs.sunysb.edu* domain. It should compile using, where necessary, the Stevens' environment in the course account, *~cse533/Stevens/unpv13e_solaris2.10*.

Basically, if we are unable, when logged in as *cse533*, to simply unpack your submission and issue a *make* which causes your *Makefile* to successfully compile your code, resulting in executables that can run, then, so far as we are concerned, your assignment is not even compiling successfully, and will be graded accordingly.

Completing the assignment

Warning: This is a very heavy and time-consuming assignment. You simply need to start working on it right away, waiting until the last week leaves you with un-implemented parts with near 1 certainty.

Incremental development - You are adviced to develop your programs **incrementally** - starting from a simple version, and then add the various functionalities and capabilities one at a time. Clearly state at the start of your documentation which parts you are able to complete.

Piazza discussion board - Read the man pages for various system calls and library functions. Reading the related textbook pages is must. You should make use of Piazza discussion board to ask and answer questions. A forum called "**Assignment 1**" has been created for this purpose. It will be read by the teaching staff and all students. Note that no exact answer to any question should be provided at any time by any student or teaching staff, online or offline. Hints can be provided but not the exact answers.

Moss and academic integrity Your submitted code will be checked using the Stanford Moss package - *A System for Detecting Software Plagiarism*, to detect cheating. Source code from previous years of this course, which has all been archived, and from the current offering, i.e., your classmates, will be checked against by this package. Also keep in mind that assignment materials are within the scope of the exam. So you must understand the steps and knowledge elements of the assignment.

Submission instructions

Important: You can only submit the source code and the documentation! No executable or object file is accepted; including them will result in deducted marks. This means before you submit, you must make a clean submit directory that has only the required files in it. Source code includes only `.c` and `.h` files. The documentation includes two files: `Makefile` and `Readme`, with those names.

What to submit: - Your `Readme` file must be well organized. It should contain a **User Documentation**, explaining how to compile and run your program. Though a **Testing Documentation** is not required, you should give several examples on how to run your program. Readme should also contain a brief **System Documentation**, stating the major parts that you have completed, and where these parts are in your source, and how they are implemented. Clearly label each of these documentation sections in your write-up. Your source code should have brief **Program Documentation** in it. This is also called comments. Your source code should have a decent amount of comments with good style. Your comments should not repeat what the line of code says, rather should add to it to ease code understanding.

Blackboard submission only - The handing-in will be through Blackboard Assignments. Note that there are more than one types of assignments supported by Blackboard. We do not use "SafeAssign" or "Digital Dropbox". Rather we use "Assignments" only. The submission instructions are at: http://it.stonybrook.edu/help/kb/creating-and-managing-assignments-in-blackboard. You must read the submission instructions very carefully, and check to make sure your assignment has been submitted correctly **before** the deadline. You can only submit once! However you can save your work by clicking "Save" as many times as you like. Only click "Submit" after you have checked and are certain that all requirements are followed.

The due date is 11:55pm on Wednesday, September 30. No late submissions will be accepted.

Last updated: 9/11/2015