TCP Echo Server and Client
ECEN 602 Network Programming Assignment 1

Introduction

In this assignment, you will implement the client and server for a simple TCP echo service, which does the following:

1. Start the server first with the command line: `echos Port`, where `echos` is the name of the server program and `Port` is the port number on which the server is listening. The server must support multiple simultaneous connections.
2. Start the client second with a command line: `echo IPAdr Port`, where `echo` is the name of the client program, `IPAdr` is the IPv4 address of the server in dotted decimal notation, and `Port` is the port number on which the server is listening.
3. The client reads a line of text from its standard input and writes the line to the network output to the server.
4. The sever reads the line from its network input and echoes the line back to the client.
5. The client reads the echoed line and prints it on its standard input.
6. When the client reads an EOF from its standard input (e.g., terminal input of Control-D), it closes the socket and exits. When the client closes the socket, the server will receive a TCP FIN packet, and the server child process' read() command will return with a 0. The child process should then exit.

Assignments must be written in C or C++, and they are to be compiled and tested in a Linux environment. Because the goal of the exercise is to understand system calls to the socket layer, you are prohibited from using any socket "wrapper" libraries; however, you may use libraries for simple data-structures. It is also acceptable, to use the Unix Network Programming, Vol. 1, 3rd Edition "wrappers" for the basic networking function calls (e.g., `socket`, `bind`, `listen`, `accept`, `connect`, `close`, etc.…). These "wrapper" functions check for error returns from the network functions [1], and you can get the source code online (see Supplementary References, Socket Programming in C, item 5).

Echo Protocol Specification

RFC 862, the Echo Protocol, May 1983, by Jon Postel (1943-1998) is shown in Figure 1 below. Jon Postel made many significant contributions to the development of the Internet. He was the editor of the Request for Comments (RFC) series of documents and administered the Internet Assigned Numbers Authority (IANA) until his untimely death. Postel's law (RFC 760) states, "an

implementation should be conservative in its sending behavior, and liberal in its receiving behavior." There is little detail in RFC 862. In contrast, the RFCs for all of the major Internet protocols are very specific.

A client and server that echoes input lines is a simple example of a network application. This program will require you to go through all the basic steps to implement a TCP client/server application. To expand this example for your own application, all you need to do is to change what the server does with the input it receives from the clients.

```
Network Working Group                                          J. Postel
Request for Comments: 862                                          ISI
                                                                May 1983


                              Echo Protocol


This RFC specifies a standard for the ARPA Internet community.  Hosts on
the ARPA Internet that choose to implement an Echo Protocol are expected
to adopt and implement this standard.

A very useful debugging and measurement tool is an echo service.  An
echo service simply sends back to the originating source any data it
receives.

TCP Based Echo Service

   One echo service is defined as a connection based application on TCP.
   A server listens for TCP connections on TCP port 7.  Once a
   connection is established any data received is sent back.  This
   continues until the calling user terminates the connection.

UDP Based Echo Service

   Another echo service is defined as a datagram based application on
   UDP.  A server listens for UDP datagrams on UDP port 7.  When a
   datagram is received, the data from it is sent back in an answering
   datagram.
```

Figure 1: RFC 862 Echo Protocol

Figure 2 shows the echo server and client and the functions used for input and output. The functions `fgets`, `fputs`, and `read` are the standard system calls. The functions `writen` and `readline` are functions for you to program. There are two arrows in the figure between the client and the server, but this is really just one full-duplex TCP connection.

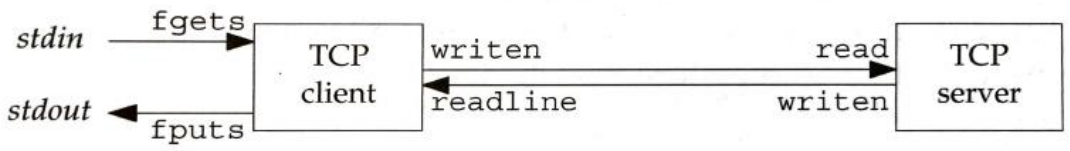

Figure 2: Echo client and server

`fgets()` – The fuction `fgets` is safer than `gets` because it is not possible for the input length to exceed the storage available in the storage array.

`writen()` – The function `writen` should write n bytes to a socket and return the number of bytes written or a -1 on error. As we noted in class, TCP sockets are different from normal file I/O. A `read` or `write` on a TCP socket might input or output fewer bytes than requested, but this is *not* an error. It is possible for socket buffer limits in the OS kernel to be reached, which can result in this behavior. All that is necessary to input or output the remaining bytes is to call `read` or `write` again. This possibility is always present with a `read` to a TCP socket, but it is normally only seen on a `write` if the socket is *non-blocking*. All of the sockets are *blocking* in this assignment. Consequently, you can get away without a loop around your call to `write`, but best practice would be to include the loop. [2]

One error from "slow system calls" (e.g., `accept`, `read`, `write`, `select`, `fgets`, etc.) that is non-fatal is "`EINTR`" (interrupted system call). The basic rule that applies here is that when a process is blocked in a slow system call, the process catches a signal, and the signal handler returns, the system call can return an error of `EINTR`. A signal, sometimes called a software interrupt, is a notification to a process that an event has occurred. It is good form in your `writen` and `readline` functions to check for the `EINTR` error and re-issue the socket `read` or `write` command. Handling the `EINTR` error in the function prevents the caller of the function from having to deal with a short count. [2]

`readline()` – The `readline` function should read a "line" of characters from a socket and return the number of bytes read or a -1 on error. Inputs to the function include the following: socket descriptor, pointer to the buffer, and the maximum size of a line + 1 (the "+1" is necessary in order to store a null termination '\0' for the string, see below, and the "max length of the line + 1," `maxlen`, is also the max size of the buffer). Reading a line from a socket is more complicated than you might think. Good defensive programming practices (see the Postel quote earlier) require programs to check for unexpected network traffic so you can recover from problem traffic, report protocol errors, and detect malicious attempts. There are many line-based protocols (e.g., the Simple Mail Transport Protocol, SMTP, the File Transfer Protocol control connection protocol, and the Hypertext Transport Protocol, HTTP), so the need to operate on lines comes up again and again. [2]

A really slow, but simple, version of the `readline` function might call the system `read` function with the number of bytes to be transferred equal to one in a `for` loop from `n` = 1 to `n` < `maxlen` and check each character read as follows: (1) Newline – check for a newline character '\n' and if a match store a '\n', null terminate the string (`*ptr = 0` like `fgets()`), and return the number of characters read; (2) `n` = `maxlen` – 1 and no newline character received, null terminate the string and return the number of characters read; (3) EOF – check for an EOF, which corresponds to getting zero characters returned from a call to `read()` without previously having found a newline character, and then null terminate the string as above and return the number of characters read; (4) EINTR error from `read()` – read again without incrementing the loop counter `n`; and (5) Other error from `read()` – return a -1. [2]

The basic idea in developing a more efficient version of the `readline` function is to code your own read-a-character function that reads a buffer of data using `read` and returns one character per function call [2]. Your read-a-character function needs to check for (1) EINTR error, calling `read` again if an EINTR error is returned, (2) other errors from `read()`, and (3) EOF. To code this more efficient version of `readline`, you will have to use static variables to maintain state (i.e., the read buffer, current buffer pointer, and current buffer count) across calls to your read-a-character function, which has the downside that the functions will not be *thread-safe* or *re-entrant*. It is possible to develop a *thread-safe* version, but we will not do that in this assignment [3].

`fork()` – The echo server is required to support multiple simultaneous connections from clients. To implement this feature, you will need to call the `fork` system call, which creates a "child" process to handle each client connection [4]. The IP addresses and port numbers four-tuple is unique for each client/child-server pair (the child-server has the same IP and port number, but the ephemeral port number of the client will be unique). The hard part in understanding `fork` is that it is called *once* but returns *twice*. It returns once in the calling process (known as the parent) with a return value that is the Process ID (PID) of the newly created process (known as the child). It also returns once in the child process with a return of 0. Consequently, the return value tells the process whether it is the parent or the child. A child has only one parent, and it can easily find its parent's PID by calling `getppid`. A parent can have many children, however, and there is no way to obtain the PIDs of its children. If a parent needs to keep track of its children, it must save the PIDs returned from `fork`. All socket descriptors open in the parent before the call to `fork` are shared with the child after the `fork` returns. You will need to use this feature in your echo server. The parent server will call `accept` and then call `fork`. The connected socket is then shared between the parent and the child. The child will `read` and `write` to the connected socket, and the parent will close the connect socket and then loop back to issue another `accept`.

Errors – It is very important to check all error returns from network calls. When an error occurs in a Unix/Linux function, the global error variable `errno` is set to a positive value indicating the type of error, and the function itself normally returns a -1. For this assignment, you should print out the error description. There is an `err_sys` function [1] in the UNP code (see Supplementary References handout) that prints out the error message in English (e.g., ETIMEDOUT is printed as "Connection Timed Out"), which is useful for debugging. The value of `errno` is only set when there is an error; it is undefined if the function does not return an error. Storing `errno` does not work with multiple threads that all share global variables. There is a solution to this problem discussed in [3], but you do not need to worry about this subtlety for this assignment.

Zombie Processes – When your child processes terminate, the signal `SIGCHLD` will be sent to the parent process. If you do not have a signal handler, the child process will enter the zombie state, which you can see by running `ps`. We should clean up our zombie processes, but you have enough on your plate to worry about for a first assignment. The POSIX way to deal with signals is to establish a signal handler by calling the `sigaction()` function. [5]

Submission Guidelines

1. My expectation is that each team member will contribute equally to the network programming assignments. My recommendation for this assignment is that one of you develop the client and the other develop the server. Please include a statement in your README that describes the role of each team member in completing the assignment.
2. Test cases – Please develop a set of test cases to demonstrate that your client and server work correctly. Submit a short report describing the test cases along with screen captures of the test cases to document correct operation. At a minimum, include the following test cases: (1) line of text terminated by a newline, (2) line of text the maximum line length without a newline, (3) line with no characters and EOF, (4) client terminated after entering text, and (5) three clients connected to the server.
3. The network programming assignments are due by 5:00 pm on the due date.
4. Your source code must be submitted to Turnitin.com using eCampus by 5:00 pm on the due date. Turnitin.com is plagiarism detection software that will compare your code to files on the Internet as well as your peers' code. Additional details on how to submit your code will be provided shortly.
5. All programming assignments must include the following: makefile, README, and the code.
6. The README should contain a description of your code: architecture,

usage, errata, etc.

7. Make sure all binaries and object code have been cleaned from your project before you submit.
8. Your project must compile on a standard Linux development system. Your code will be graded on a Linux testbed.
9. Explanation of the submission procedure will be provided shortly by the TA.

Notes

1. The best way of learning new system calls is to build very small programs that demonstrate simple things. Unless you are familiar with the subject matter, you should experiment at first.
2. Once you are ready to begin, you should start with a conceptual model/ architecture of your client and server? What is your basic data model? What are the major decision points in the code? You should sketch these thoughts out on paper before you begin (it is also required with submission of the project).
3. Test your code as you write it (unit testing is a great thing).
4. Coding style is very important. People, including yourself, are more likely to understand your code if it follows a style that is simple and self-consistent. We will not enforce any specific style; however, you should spend a little time on deciding on a style and sticking with it. Two good references are: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml, and http://lxr.linux.no/#linux+v2.6.35/Documentation/CodingStyle. What you pick is not as important as being consistent through your code.
5. A makefile is a standard tool for any software project. You will need to create one to build and clean your project. There are many online resources that can get you started.
6. A great way to do random testing is to partner with another team and perform interoperability testing. Your client should work with their server, and their server should work with your client. This is guaranteed to reveal implementation bugs, as well as possible design defects.
7. Come to Recitation and/or the TA's Office hours to make sure you understand the assignment or if you are having trouble making progress.
8. You must comment your code.
9. Be sure to check all error returns from the network functions and to check all buffer writes to avoid buffer overruns. You don't want to be the network programmer responsible for the next "Heartbleed" bug.

References

[1] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, *Unix Network Programming, Volume 1, The Sockets Networking API, 3rd Edition,* Addison-Wesley, 2004, Chapter 1.

[2] W. Richard Stevens, et. al., *op. cit.,* Chapter 3.
[3] W. Richard Stevens, et. al., *op. cit.,* Chapter 26.
[4] W. Richard Stevens, et. al., *op. cit.,* Chapter 4.
[5] W. Richard Stevens, et. al., *op. cit.,* Chapter 5.