CIS*3210 Computer Networks

Assignment 1: Socket Programming

RFC Documentation Due Monday, September 30, 2019 @ 11:59pm
Code Due Friday, October 11, 2019 @ 11:59pm

This assignment can be done **in teams of two**. You are welcome to use any of the socket code examples that I have provided for this course.

In this assignment you will have to develop a client-server application using TCP socket in C. Your application is a model of on-line book catalogue.
Clients will:

- Request a connection with the server
- Send three types of messages through established connection:
    1. SUBMIT messages containing book description
    2. GET messages containing requests for a particular book
    3. REMOVE messages containing requests to remove a book from catalogue

Server will have to:

- Accept requested connections and support multiple connections
- Maintain an appropriate data structure that holds data received from a client. The data stored on server is a book catalogue which is empty at startup. It is not persistent.
- Process messages received from client:
    1. If message is SUBMIT, server will have to update the catalogue.
    2. If message is GET, server has to send to client a list of locations where the book is available or an error message if request cannot be satisfied.
    3. If message is REMOVE, server has to remove the instance of the book described in the request message from the catalogue.

    In any case server has to react properly to the erroneous messages.

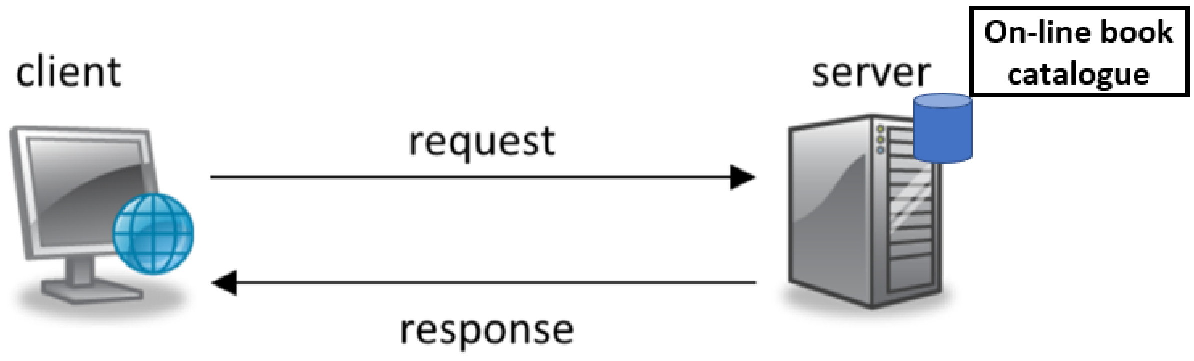

Figure 1. On-line book catalogue protocol request/response

Simplified book description has the following components: TITLE, AUTHOR, LOCATION

Here are several examples of requests from a client in ASCII:

```
SUBMIT
TITLE Introductory Computer Forensics
AUTHOR Xiaodong Lin
LOCATION Reynolds Bldg, 2210
```

```
GET
TITLE Introductory Computer Forensics
```
Client expects to receive records of all books in catalogue with this title

```
GET
AUTHOR Xiaodong Lin
```
Client expects to receive records of all books in catalogue with this author

```
GET
TITLE Introductory Computer Forensics
AUTHOR Xiaodong Lin
```
Client expects to receive records of all books in catalogue with this title and this author

Note that each request consists of multiple lines, each followed a newline character. The last line is followed by an additional newline character.

Your server is a console application (command-line program) that starts with empty catalogue and take as its first command-line argument the port to listen to. If the server cannot bind to the port that you specify, a message should be printed on standard error and the program should exit.

Your client is also a console application, which takes two command-line arguments. The first command-line argument is the IP address of the server, and the second one is the TCP port number that the server listens on. After your client establishes a TCP connection with the server, you can type a request from your keyboard and send it over the established TCP connection and wait for a server reply. Once a server response has been received, you can either repeat the above steps by typing another request or type quit to terminate the client after releasing any created sockets. Note that "quit" is case insensitive here.

Also, both your client and server **should be robust for bad arguments**! If the argument is invalid or bad, your programs should print a usage statement, and then exit graceful indicating an unsuccessful termination [1]. Further, your programs should be robust to handle abnormal situations, for example, socket errors.

There are several parts you must pay attention to about usage statements [2]

- **The usage message**: it always starts with the word "usage", followed by the program name and the names of the arguments. Argument names should be descriptive if possible, telling what the arguments refer to, like “filename” in the example below. Argument names should not contain spaces! Optional arguments are put between square brackets, like “-l” for the Linux command *ls*. Do not use square brackets for non-optional arguments! Always print to stderr, not to stdout, to indicate that the program has been invoked incorrectly.

- **The program name**: always use argv[0] to refer to the program name rather than writing it out explicitly. This means that if you rename the program (which is common) you won’t have to re-write the code.
- **Exiting the program**: use the exit function, which is defined in the header file <stdlib.h>. Any non-zero argument to exit (e.g. exit(1)) signals an unsuccessful completion of the program (a zero argument to exit (exit(0)) indicates successful completion of the program, but you rarely need to use exit for this). Or, you can simply use EXIT_FAILURE and EXIT_SUCCESS (which are defined in <stdlib.h>) instead of 1 and 0 as arguments to exit.

For example, the following is a code snippet for the server, which prints a usage statement, and then exits the program by indicating an unsuccessful termination

```
fprintf(stderr, "usage: %s <port_number>\n", argv[0]);
exit(EXIT_FAILURE);
```

NOTICE Note that a short design document is required. The design document describes your code and the design decisions that you made. You should program your client and server to each print an informative statement whenever it takes an action (e.g., sends or receives a message, close a TCP connection, etc.), so that you can see that your processes are working correctly (or not!). This also allows the TA to also determine from this output if your processes are working correctly. You should hand in screen shots (or file content, if your process is writing to a file) of these informative messages as well as the required output of the client and server (retrieved book record(s)).

Request for comments (RFC)

You will first have to design the protocol used by your on-line book catalogue application. Request for comments (RFC) is a widely used documentation format detailing the specification of a protocol used by a network application. For example, the HyperText Transfer Protocol (HTTP) is the most widely used Application layer protocol in the world today, and is first officially introduced and defined in RFC 1945 as HTTP V1.0 in 1996 [3].

In addition to your developed code, in this assignment you will have to submit documentation (single PDF file) in form of a short RFC, containing description of your protocol, format of messages sent by client and server (do not forget to describe format of <response>, since it is not given here), synchronization policies, reactions of server and client to the errors, border-cases behavior etc.

Each RFC has a unique number, and your RFC has been assigned with **RFC 32108**. Also, when you create your RFC document, you can refer to an existing RFC documentation, particularly RFC 1945.

Programming environment

The Computer Networks course includes several programming assignments which you'll need to finish to complete the course. You'll use linux.socs.uoguelph.ca for your development and testing. Your assignment grading will be based on the same system. This assignment should be completed in ANSI C or C++. It should compile and run without errors on linux.socs.uoguelph.ca producing two binaries called server and client. You'll need to master the C system calls needed for socket programming. C is a popular programming language for network programming since it gets you closest to the operating system's socket-related system calls. These include socket(), bind(), listen(), accept(), connect(), and close(). For a quick text-only tutorial of socket programming specifically under Linux, see http://www.lowtek.com/sockets/. Here's another nice tutorial: http://beej.us/guide/bgnet/. If you find better pages for C, please share with the class and post the links onto A1 discussion forum!

Remember that linux.socs.uoguelph.ca is an alias for several different hosts (george.socs.uoguelph.ca, ginny.socs.uoguelph.ca, etc.). Make sure you know which host you're connecting to. Since we will end up with multiple servers running on the same host, each of you will need a unique port number. I will post a list on CourseLink with your names and port numbers. Use whatever port number you wish for initial testing. I assume that your initial development/testing will use the localhost, anyway. Nevertheless, you may still experience your server failed to bind on a port. Then, you try to find another port available. If the server cannot bind on a port, print a message to standard error.

Nevertheless, students can use your own laptop for the assignments. If you use a Mac, you will be using the Terminal program and unix command line tools. If you use Windows, you will install a virtual machine running Ubuntu linux to use for your assignments. However, when you are done your assignment, you must test your programs on linux.socs.uoguelph.ca to make sure that your programs work properly on it.

How to Test Your Programs

You should test your implementations by attempting to send requests from your clients to your servers, and report your results. The server can be run in the background (append a & to the command) or in a separate SSH window. You should use 127.0.0.1 as the server IP and a high server port number between 10000 and 60000. You can kill a background server with the command fg to bring it to the foreground then ctrl-c. You should also test your code with multiple clients. (at least up to 5 simultaneous clients)

OK. You've got the basic assignment done. The last part of the assignment should be a fun, social thing to do. Team up with someone from the class (advertise on the A1 discussion forum on CourseLink if you are looking for a partner) and get your client to interact with their server or vice versa. If you've got a server up and running, you can advertise your server's services to anyone in the class (you're not allowed to charge for service!), and if you've got a client, all you need to do is interact with such an advertised server.

Note that if you've got your own client and server running, there isn't any more programming involved – just running your client and server with someone else's. This shouldn't actually be very hard at all! For this part of the assignment, you don't need to hand in anything. Instead, this will hand your programs over to someone else (and anonymous) for more extensive testing. Recall that when we discussed RFC standards for protocols we noted that the IETF requires that two independent implementations of a protocol must interoperate; that's what you are doing here. Again, make sure to record all requests and responses so you can verify that your programs work properly.

Skeleton

You can find a skeleton framework in which you should implement your code enclosed. This is provided as a starting point for your programs. It includes two files---"server.c" and "client.c" for implementing the server and client, respectively---as well as a Makefile for producing the executables and an empty README file which you should fill out with all the instructions necessary to run your assignment.

Your task is to complete the above skeleton code (starter code) with the appropriate c code to write your own client and server. Besides the functionalities required in this assignment, an emphasis should also be placed on writing concise easy to read code. **In general, code must be clean and organized. It must be appropriately commented as well.** Please refer to the coding style guidelines (e.g., C Programming Style Guide [4]) to provide direction regarding the format of the code you write. As the program grows, you may want to improve the structure of the code by moving certain parts into separate functions. **You will be graded on (i) accuracy and clarity of your RFC document; (ii) the correctness and reliability of your code; and (iii) its readability and structure.**

Note: In order to compile your source code in simple and fast way, you are required to use Makefile for building the executable files, including

- make all: creates executables server and client
- make clean: deletes all executables and intermediate files

A Makefile is a recipe for the make utility [5] how to create some file (called a target) from some other files (called dependencies) using a set of commands run by the shell. Please refer to the following tutorials on how to use make and write Makefiles

- How to Write a Simple Makefile from "Managing Projects with GNU Make", third edition by Rober Mecklenburg
  https://www.oreilly.com/openbook/make3/book/ch01.pdf
- Using make and writing Makefiles
  https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
- Makefile Tutorial

https://www.wooster.edu/_media/files/academics/areas/computer-science/resources/makefile-tut.pdf

- Writing Make Files
  https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Required files

Phase 1: Due Monday, September 30, 2019 @ 11:59pm

- RFC Documentation

Phase 2: Due Friday, October 11, 2019 @ 11:59pm

- All the .c and .h files for your client and server
- A Makefile that compiles the server and the client
  - ➢ make all: creates executables server and client
  - ➢ make clean: deletes all executables and intermediate files
  - ➢ make zip: create a zip archive with all your deliverables
- A README file that provides all the instructions necessary to run your assignment
- A design document that describes your code and the design decisions that you made. Also, the report should show the test results from your test environments.

Submission

To submit your RFC documents, it must be submitted as one PDF document, which is named cis3210_ass1_RFC32108_XXX.pdf, where XXX is your University of Guelph's email ID (Central Login ID).

To hand in your program, create a zip archive with all your deliverables and submit it on CourseLink. The filename must be cis3210_ass1_XXX.zip, where XXX is your University of Guelph's email ID (Central Login ID). This naming convention facilitates the tasks of marking for the instructor and course TAs. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction. Do not include any binary files in your submission.

Note: to zip and unzip files in Unix:

```
$zip -r filename.zip files
$unzip filename.zip
```

References:

[1] Exit Status.
https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html
[2] Processing command-line arguments.
http://courses.cms.caltech.edu/cs11/material/c/mike/misc/cmdline_args.html
[3] Hypertext Transfer Protocol -- HTTP/1.0.
https://tools.ietf.org/html/rfc1945
[4] C Programming Style Guide
http://faculty.cs.tamu.edu/welch/teaching/cstyle.html
[5] http://man7.org/linux/man-pages/man1/make.1.html
[6] Rober Mecklenburg. How to Write a Simple Makefile from "Managing Projects with GNU Make", third edition
https://www.oreilly.com/openbook/make3/book/ch01.pdf
[7] C - Command Line Arguments.
https://www.tutorialspoint.com/cprogramming/c_command_line_arguments
[8] Socket Programming in C/C++
https://www.geeksforgeeks.org/socket-programming-cc/
[9] Blaise Barney. POSIX Threads Programming tutorial
https://computing.llnl.gov/tutorials/pthreads/