

Application-Level Framing over TCP

Please read all sections of this document before you begin coding.

In this machine problem, you will develop software to transfer a file in a specific format across a TCP connection to another terminal. The novel (and perhaps challenging) aspect of your client-server protocol will be the use of application-level framing to encode the contents of the file on a line-by-line basis. A command-line argument to the server specifies the file name, and the server must deliver this file to each client that contacts it, possibly sending concurrently to several clients. Each client receives the encoded data and interprets it appropriately to recreate an exact copy of the original file, again using a file name specified on the command line.

Objectives

The primary goals of this machine problem are for you to learn to build simple client-server applications using the sockets API and for you to understand the process of sending messages over TCP. Servers generally manage multiple connections simultaneously, and you should also learn the basics of concurrency from this problem.

As mentioned above, you will implement a client-server pair to encode the contents of a file for transmission over a TCP connection. The file is defined as a set of lines with a particular format (see the Specification section below). You must design a framing strategy that allows your client to recover the original data and to create an identical copy of the original file. For this design, you must consider both correctness and efficiency (the overhead of framing). Your server must handle multiple client connections concurrently, which can be done in several ways. For this assignment, you might prefer to use multiple Unix processes; in later assignments, you will need to use multiple threads.

In addition to correctness, your design will be judged on its efficiency, i.e. how many bytes are needed to transmit the file. Your submitted results must include three or more designs and analysis of their relative efficiencies for a sample file. You need implement only one of the designs.

To get you thinking about efficiency, examine the given file format closely. The format was not random, so looking at the data the file represents may give you ideas about what can be done to send less physical data over the wire, but still send logically the same thing.

For example, to represent 8 symbols, how many bits are needed? If each symbol was sent as a char, how many bits are being sent?

Guidelines

Please work on this MP individually. Collaboration, discussion, code sharing, and any other form of group work is forbidden. Offenders will be dealt with appropriately (see the general information sheet available from the class home page).

You are required to make sure that your programs compile on the `{dclsn, glsn}*.ews.uiuc.edu` machines and to use the EWS lab. Programs suffering compilation errors when tested by the course staff will earn no credit.

You may find it useful to read Chapters 1-4, 6, and 8 in Stevens. Refer to the Unix man pages as necessary.

Please indent and document your code. Use meaningful names for variables and follow other style guidelines that enhance the clarity of your code.

Follow the guidelines in the “Hand In” section below when turning in this assignment.

Specification

Make a directory called MP2 somewhere inside your main directory to hold your code. Call the file for your server code `server.c` and call the executable `server`. Follow the same convention for the client. Note that the source

file names are recommendations; in contrast, the executable file names are requirements. Create a Makefile that automatically generates the executables from your sources.

This problem consists logically of three components: a client thread, a main server thread, and a per-client server thread. This document refers to the problem components as threads, and you are encouraged to use Posix threads to implement them (see Chapter 23 of Stevens). In later machine problems, threads will become a requirement. However, for this problem, separate Unix processes (see Beej's code from MP1) are also acceptable.

File Format: The file format for use with this machine problem is an example of a directoly listing. A sample file (from `~ece438/MP2/sample`) appears below:

```
rwrxwx--- 75 Feb 18 2005 foo/bar 7 1234 1453
r--r--r-- 134 Jan 24 2006 foo/baz/quux 12 1234 1453
r-x--x--x 12000 Oct 1 2006 fred 4 1453
rwx----- 0 Sep 13 2006 barney 6 1234
```

A file can contain any number of lines of the form shown. Each line consists of several fields, and must follow the rules below:

1. All fields must be separated by exactly two SPACE characters, and each line terminated by a single linefeed, with no additional white space.
2. The total number of characters, including the linefeed at the end, can be no more than eighty.
3. The first field is a set of flags in 3 groups of 3: `rwrxwxrx`. Each flag is either present or marked absent with a '-'.
r--r--r--
4. The second field is the file size (at most $2^{32} - 1$)
5. Fields three through five represent the date. The month is first, represented by the first three characters of the name, followed by the date, followed by the year. Assume the year is in the range 1970–2037.
6. The six field is a file name. It may contain characters 'a'–'z', 'A'–'Z', '0'–'9', '/', and '-'
7. The seventh field is the length of the file name.
8. The last fields are a list of user IDs allowed to access the file. This list contains numeric IDs in the range 0 to 65535.

Framing Styles: You must consider at least three frame formats to communicate the lines of the files from the server to its clients. For example, you might simply send the full text of the file, although such an approach is not very efficient. The required multiplicity is intended both to help you understand the possibilities and to introduce you to the complexity and utility of each approach. Consider applying the methods discussed in class to frame the variable-length list of user IDs.

Refragmentation and Logging: One very important aspect of TCP byte streams that many people find difficult to grasp is the fact that calls to read and write (or send/recv, etc.) are not correlated. In particular, two or more writes to a connection can be received by the same read, and a single write can be split amongst several reads. The EWS environment makes this lesson harder because, on a local area network, the timing issues make it seem like correlation works.

To ensure that you gain a solid understanding of framing application data on a TCP connection, we have provided a routine to make the environment much less predictable. In particular, you must use the interface below in place of read for all of your MP2 client code (no calls to any other routine to read data from a socket are allowed):

```
size_t mp2_read (int fildes, void* buf, size_t nbytes);
```

The `mp2_read` function has exactly the same format and semantics as `read`¹, but will almost always split the contents of each write across several reads, and may also merge pieces of multiple writes into a single read. Only the order of the bytes is preserved, as is effectively the case with `read`.

Use `mp2_read` just as you would use `read`; you probably want to write a wrapper function (see Stevens p. 78). The header file `/homesta/ece438/MP2/mp2.h` contains a function prototype for `mp2_read`. Link the `mp2.o` file from the class directory into your client and server executables. For example, the link command for your server might appear as:

```
gcc -o server server.o /homesta/ece438/MP2/mp2.o -lsocket -lnsl -lpthread
```

You must link the file in the class directory rather than making a local copy to guarantee proper behavior when your assignment is graded. The `mp2_read` function is not thread-safe, but your client needs only one thread to read data from the server. For interested students, commented source code is available in the same directory as the object file.

In addition to performing refragmentation, the MP2 class routine records the number of calls made to it and the number of bytes returned in the file `mp2_results`. These statistics will be used to calculate the efficiency of your framing format during grading.

Client: The client must accept two mandatory command-line arguments—the name of the server (e.g., `glsn10.ews.uiuc.edu`) and the output file name:

```
client <server> <output file>
```

A client first checks the syntax and parses the arguments, then translates the server name into an IP address and verifies that it can write to the output file. If any error occurs during this stage, the client prints a brief error message and exits. If all arguments are valid, the client opens a TCP connection to the server and begins reading lines, removing or adding any framing information, and writing them to the file in the proper format. After the last line arrives, the server closes the connection, and `mp2_read` returns 0 to the client. The client then closes the connection and exits. Note: there is an implicit assumption that the client knows the port that the server is listening on. This port should be defined in a constant at the top of your code.

Server: The server must accept a single mandatory command-line argument—the name of the file to be sent to clients:

```
server <input file>
```

Control begins in the main server thread. Like the client, the server must first verify the command syntax and the existence of an input file. Error handling is similar to that of the client. If the input file exists, the main thread creates a TCP socket and prepares it for incoming connections. You may copy Beej's code for this purpose, but be sure to credit Beej. You should also pick a new port number to avoid collisions with other students. You may also want to preprocess the input file in the main thread, although having the per-client threads each read the file separately is also acceptable.

When a client connects, the main thread creates a new process or thread to manage the connection. After cleaning up any per-client state, e.g., closing the file descriptor after forking a new process, the main server thread returns to waiting for client connections.

The per-client thread should first indicate that it is sending the file to the client by printing the IP address and TCP port number of the client to the screen. If the main thread does not preprocess the input file, the per-client thread must read it in before (or while) it sends the framed lines to the client. You need not check the format of the file—assume that it follows the specification given earlier. Checks may help with debugging, of course, if you are creating your own files for transmission. Once all lines have been sent to the client, a per-client thread cleans up any temporary state, closes the client socket, and exits.

¹The class routine does not respect Posix minimum fragmentation guarantees, but the impact on your code because of this difference is negligible.

Testing and Grading

You are responsible for testing your code to ensure that it complies with the specifications given in this document. This section is intended to give you some things to think about when testing, and to warn you about some pitfalls that people have encountered in past classes.

The sample file is not meant to serve in place of testing your code, and testing with only the sample file is likely to leave bugs in your code. The TA's will test your code much more thoroughly for grading, and your grade will be based on those tests. It may help (or amuse you) to think of the TA's as your customers, and your grade as a customer satisfaction rating.

Correctness will be determined using the `diff` tool, and no partial credit given for correctness points. If you can't get an aggressive approach to framing working properly, you may be better off using a less aggressive approach that works.

Be careful with signed and unsigned values, and be sure to test with many values for each field, including extreme values for each ranged number.

We will not limit file size for you. The TAs will not want to spend weeks testing your code, so you are safe in assuming that a file to be sent will not occupy more than some fraction of the machine's memory, but you should not assume that test files will contain only a few (hundred, thousand etc.) lines.

Hand In

Place your source code in your MP2 directory along with a Makefile that, on execution of the command `make`, causes the executable code for the programs `client` and `server` to be generated by the compiler.

Finally, create a file `README` containing two parts.

- *Part 1: Implementation Log*

This part should briefly describe and justify your design and implementation decisions, including any data structures used to support your frame format. Discuss any other interesting aspects of your implementation. As always, you should acknowledge the sources if you copied sections of your code from outside courses (even if you modified it later).

- *Part 2: Frame Format Design*

Provide a comparative analysis of three or more possible frame designs, including the one you used, which is presumably the best of the three. In particular, explain how each design works and calculate the number of bytes required to send the sample file `~ece438/MP2/sample`.

When you get the C programs, `README`, and Makefile files within the directory `MP2` in good working order, you are to transfer the directory electronically as follows. First, remove all object files and executables from the directory, leaving only the source code, the Makefile, and the `README` file. **Submission of executables will reduce your grade.** Next, from within the directory, type:

```
~ece438/Handin/handin
```

Follow the prompts and hand in the C programs, the `README` file and Makefile. (Please check the course newsgroup for tips in case our `handin` program isn't running smoothly.)

Grading Guidelines

10 pts Code

- Well commented
- Meaningful variable and function names
- Readable and indented

20 pts README

- Implementation Log
- Acknowledge sources
- Discussion of 3 framing formats
- Plain text file

10 pts Makefile/Executables

- Executables and source files have correct names
- No extra files handed in
- No errors in generating executables with 'make'

30 pts Correctness

- Diff original file with transferred one (ALL OR NONE 30 pts)

15 pts Concurrency

- Server can handle multiple concurrent clients

15 pts Performance

- Will be a formula based on class average.