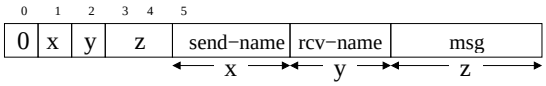
Programming Assignment #1 - HW #2

COMS W4119 - Computer Networks — Due Feb 15, 2006
Spring 2006 — Prof. Rubenstein

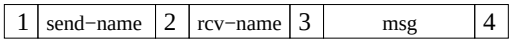
Programming Assignments must be e-mailed to the TA. The e-mail should contain a .zip file containing all files to be turned in. If desired, answers to questions can be turned in at the beginning of class on the day that the homework is due. CVN students have one additional day. Late assignments will not be accepted.

In this first assignment, you will build a chat application with some similarities to problem 1 of HW #1.

Stream Formats



(a) header type 0



(b) header type 1

Your chat application should be able to utilize two forms of headers, depicted above. The first firm, which we will call `FORM 0`, is depicted on the left. The first byte of a stream will have value 0, indicating that the rest of the stream is stored in FORM 0 format. The 2nd, 3rd, and 4th bytes whose values will be X,Y,Z respectively specify the lengths of the sender's name, receiver's name, and message. The sender and receiver's names can be at most 256 bytes (0 lengths not allowed). Hence, the value X indicates a name of length $X + 1$. The same is true for the value Y.

The length of the message can be between 0 and $65,535$ bytes, so the 2-byte quantity, stored in big endian (network byte-order), Z specifies this quantity. The sender name, receiver name, and message are then concatenated together to form the rest of the stream.

The second form, called `FORM 1`, is depicted on the right. The first byte of the stream will have value 1. The sender name, receiver name, and message must contain all ASCII characters. The receiver, sender, and message ends are delimited by the respective byte values 2, 3, and 4.

Your task

You are to design two versions of a chat application: one that works in connectionless mode, and the other that is connection-oriented. For both chat apps, the application should be built as follows:

- Suppose the executable is named `chat-app`. Then `chat-app` should take between 3 and 5 command-line parameters: `chat-app mode addr port [outfile] [infile]`
  - `mode` is a required should be set to 0 to run the connectionless chat app, and set to 1 to run the connection-oriented chat app.
  - `addr` and `port` are required parameters. Their behavior is as follows:
    - If `addr` is set to 0, then this chatter expects another "chatter" to initiate the conversation. The `port` value then contains a value of a local port. For connectionless mode, this local port is the port on which it will communicate with clients. For connection-oriented mode, the port is where it accepts incoming connection requests.

    - If `addr` is non-zero, then this is the address of the chatter to be communicated with, and `port` should contain the port being listened to by the remote chatter.
  - `outfile` and `infile` are optional parameters. `infile` will be a text (all-ASCII) file containing several lines. Each line will contain 4 fields, with each field delimited by a tab (\t), with the fields being packet version (0 or 1), sender name, receiver name, and message. If `outfile` is also specified, this is where all information should be sent instead of to `stdout`. It should be possible to provide an `outfile` without providing an `infile`.

- Your final version of the code should only output when sending or receiving a message. Output should be of the form:

  Vx: DEST: y, ME: z, MSG: m
  Vx: RCVD: y, ME: z, MSG: m

  where x is the version (0 or 1), y is the local user's name, z is the remote user's name, and m is the message. The top form is for a message being sent, the bottom for a message being received.

- If an `infile` is not specified, you should allow the user to input the sender and message. We will not test this, so how and whether you choose to implement this is up to you.

- When an `infile` is specified, you should permit the remote client to respond between sending the messages stored on the infile, i.e., the conversation should go back and forth. Who initiates the conversation is described below.

Testing/Verification

We will create an automated client that you can test your chatting client with to make sure it is operating correctly. The description below describes the procedure you can use to test your code.

There is also some code to download and execute for you to use when the remote client is to initiate the communication.

The webpage `http://www.cs.columbia.edu/~danr/4119/pa1-test.html` will contain additional information that we don't yet have available (e.g., port numbers).

- Connectionless, you initiate conversation: call `chat-app 0 ADDR PORT` with `ADDR` and `PORT` set to the values described on the webpage. Whenever you send a message with sender S and receiver R, the remote client should respond as sender R to you (S) with a bit of your message and some of its own thoughts. Note that it is possible for information to get lost, but loss is highly unlikely. If something strange happens, kill your client process and try again. If the strange event happens again, it is likely not due to packet loss.

- Connection-oriented, you initiate conversation: Everything (from the application perspective) is the same as above (except the first parameter is 1 instead of 0 and we will use a different port for the connection-oriented remote client). Of course, your implementation is considerably different than that of above, but everything else should look the same.

- Connectionless, remote client initiates conversation. We will provide you with a small piece of code, `ready-noconn` that you should call prior to executing `chat-app`. Details will be provided on the webpage. You should first call `ready-noconn ADDR PORT1 NAME`, where `ADDR` and `PORT1` are the address and port of the remote client (provided on the webpage) and `NAME` is the name of your client (see below). The program should respond with `OK PORT2` where PORT2 will be the number of a port, and then terminate, Once this happens, you should immediately call `chat-app 0 0 PORT2`. About 15 seconds after the OK, the remote client will initiate contact with your client, looking for your client on port PORT2. Slightly after, a second remote client will also initiate a conversation (to your same port, but from a different port).

- Connection-oriented, you initiate conversation: We will provide you with a small piece of code, `ready-conn` that you should call prior to executing `chat-app`. Details will be provided on the webpage. You should first call `ready-conn ADDR PORT1 NAME`, where ADDR and PORT1 are the address and port of the remote

client (provided on the webpage) and `NAME` is your client's name (see below). The program should respond with `OK PORT2` and then terminate, Once this happens, you should immediately call `chat-app 0 0 PORT2`. About 15 seconds after the OK, a remote client will initiate contact with your client on port PORT2, and during your conversation with that client, another client will initiate a conversation.

IMPORTANT: When you test, use your CUID as your local client's handle (i.e., the sender's name in the message whenever you send). So, for the examples above, your `NAME` should be your CUID.

Things to note

- Delimiters are sometimes specified as equaling 0 or 1. This is the byte value, and not the ASCII value. However, when supplied as parameters (e.g., via file `infile`, they will be supplied in ASCII. You should do the appropriate conversion.
- You can write your connection-oriented and connectionless apps separately if you want. However, in the final submitted version, you should package them into the same executable.
- Note that within a given session, we expect to permit the switch between different header versions.
- When your client is to receive the first communication, note that we plan to test your client *simultaneously* communicating with different remote users. In connectionless, this is fairly straightforward as we will have all users contact you on the same port (just originating from different ports). For connection-oriented, each user will have its own connection, and your app will have to manage multiple sockets simultaneously.
- When your client is talking to two (or more) remote clients simultaneously, it should allow either remote client to respond next at any point in time (i.e., your client should never block on a particular remote client in case it is the other remote client "speaks" next. Note for the connectionless case, both remote clients will communicate with you on your same port (through the same socket). Don't confuse who you respond to. For the connection-oriented, the two remote clients will contact you through different sockets. So don't block on one socket, neglecting the other.

What to turn in

The assignment should be turned in electronically, by e-mail to 4119-submit@cs.columbia.edu. You must turn in **as a single attachment** all work wrapped up together as a single gzipped tarfile with filename `pa1-CUID.tgz` (where `CUID` is your Columbia UNI). The tarfile should decompress to create a directory `pa1-CUID/` with the following files (example given in C):

- `chat-app.h`: header information
- `chat-app.c`: main code.
- If you have additional c code that you wrote that is included by these files, include that code as well.
- `chat-app`: the executable compiled from the code above that **must** execute on CS Linux machines.
- `Makefile`: your makefile or `make.txt` that explains the procedures you followed to compile your file.
- A file named `test.txt` that includes some tests on which you ran your code.
- `questions.txt`: your answers to the questions below, in text (no Microsoft Word, PDF, etc.)

`tar` is a Unix command. Suppose your current directory is `pa1-CUID`. Then

- `cd ..`
- `tar fvczh pa1-CUID.tgz pa1-CUID`

will generate the desired tarfile. To verify this worked correctly, create a directory `temp/`, enter that directory, and type `tar xvfz pa1-CUID.tgz` and the directory `pa1-CUID` and all its contents should be created (i.e., copied) into `temp/`.

Grading

Your grade will be based on *our* execution of your code against some pre-determined test cases (that may be slightly different) from the tests we provide above. So make sure that your code works, or we will not be able to test. We will also randomly spot-check the source code turned in to verify that it matches the object code, and will also compare object code across student assignments.

Again, *the code must execute on Columbia CS Linux boxes to receive credit.* More details on this forthcoming (e.g., a specific machine on which it must work).

Questions

Turn the following questions in with your assignment in file `questions.txt` Keep answers short (i.e., 2 to 3 sentences if possible).

1. For header type 0, was it necessary to specify Z: the length of the message? Why not just do a `send` (or `sendto` of a string that terminates at the end of the message?

2. Communicating with multiple remote clients is handled somewhat differently in connectionless and connection-oriented versions. Explain the differences (in terms of sockets and ports) and how you prevented your app from blocking at inappropriate times due to the need to make blocking calls.