A Client Process Speaking to a Server Process

Due: 2/21/20 Friday at 11:59pm

Introduction

In this assignment, you will write a client program that connects to a given server. The server defines a communication protocol, which the client has to implement by sending properly formulated messages over a communication pipe. The server hosts several electrocardiogram (ecg) data points of 15 patients suffering from various cardiac diseases. The client's goal, thereby your goal, is to obtain these data points by sending properly formatted messages that the server understands. In addition, the server supports is capable of sending raw files potentially in several segments (i.e., when the file is larger than some internal buffer size). Your client must implement this file transfer functionality as well such that it can collect files of arbitrary size using a series of requests/responese.

We obtained the above dataset from `physionet.org`.

1 Starter Code

You are given a source directory with the following files:

- A `makefile` that compiles and builds the source files when you type `make` command in the terminal.
- `FIFORequestChannel` class (`FIFORequestChannel.cpp/.h`) that implements a pipe-based communication channel. You can use this to communicate with another process. This class has a `read` and a `write` function to receive and send data to/from the server, respectively. The usage of the function is demonstrated in the given `client.cpp`. No change in this class is necessary for PA2.
- A `server.cpp` that contains the server logic. When compiled with the `makefile`, an executable called `server` is made. You need to run this executable to start the server. Although nothing will change in this server for this PA, you will have to refer to its code to understand the server protocol and then implement the client functionality based on that.
- The client program in `client.cpp` that, for the time being, is capable of connecting to the server using the FIFORequestChannel class. The client also sends a sample message to the server and receives a response. Once compiled, an executable file `client` is generated, which you would run to start the client program. This is the file where you will make most of the changes needed for this assignment.

- A `common.h` and a `common.cpp` file that contain different useful classes and functions potentially shared between the server and the client. For instance, if you decide to create classes for different types of messages (e.g., data message, file message), you should put them in these files.

Download the source and unzip it. Open a terminal, navigate to the directory, and then build using `make` command. After that, run the `./server` to start the server. Now, open another terminal, navigate to the same directory, and run `client`. At this point, the client will connect to the server, exchange a simple message and then the client will exit. Since pipe is a point-to-point connection, when either the client or the server exits, the other side will exit as well after receiving SIGPIPE signal for "broken pipe".

Server Specification

The server supports several functionalities. The client requests a certain functionality by sending the appropriate message to the server. Internally, the server will execute the correct functionality, prepare a reply message for the client and send it back.

Connecting to the Server

You will see the following in the server main function:

```
FIFORequestChannel control_chan ("control", FIFORequestChannel::SERVER_SIDE);
```

which sets up a communication channel over an OS-provided IPC mechanism called "named pipe". Note that the first argument in the channel constructor is the name of the channel. To connect to this server, the client has to create an instance with the same name, but with `CLIENT_SIDE` as the second argument:

```
FIFORequestChannel control_chan ("control", FIFORequestChannel::CLIENT_SIDE);
```

Requesting Data Points

After creating the channel, the server then goes in a "infinite" loop that processes client requests based on the type. Now, let us find out how the server works. The server maintains ECG values (at 2 contact points) for each patient in a time series where there are 2 data points (i.e., ecg1 and ecg2) collected every 4ms (see any of the .csv files under the `BIMDC/` directory) for the duration of a minute. That means there is 15K data points for each patient in each file and there are 15 such patients. Hence, there are 15 files each with 15K data points.

The client requests a data point by specifying:

- Message type is `DATA_MSG`. Data type is `MESSAGE_TYPE` defined in `common.h`. There are a number of message types, each serving a different purpose.
- Which patient. There are 15 patients total. Required data type is an `int` with value in range $[1, 15]$

- At what time in seconds. Data types is `double` with range $[0.00, 59.996]$
- Which ecg record: 1 or 2, indicating which ecg record the client is interested to obtain. The data type is integer.

You will find this request format in `common.h` as `datamsg`. In response to a properly formatted data message, the server replies with the ecg value as a `double`. Your first task is to prepare and send a data message to the server and collect its response.

The following is an example of requesting ecg2 for patient 10 at time 59.004 from the command line when you run the client:

```
$ ./client -p 10 -t 59.00 -e 2
```

In the above, the argument "-p" is for which patient, "-t" for time, and "-e" for ecg no.

Requesting Files

To request a file, you need to package the following information in a message:

- Message type set to `FILE_MSG` indicating that it is a file request. Data type is `MESSAGE_TYPE` defined in `common.h`
- Starting offset in the file. Data type is `__int64_t` because of the fact that the file can be large and a usual 32-bit integer will not be sufficient.
- How many bytes to transfer beginning from the starting offset. Data type is `int`.
- The name of the file as NULL terminated string, relative to the directory `BIMDC/`

The type `filemsg` in `common.h` encodes these information. However, you won't see a field for the file name, because it is a variable length field. If you were to use a data type, you would need to know the length exactly, which is impossible beforehand. You can just think of the name as variable length payload data in the packet that follows the header, which is a `filemsg` object.

The reason for using offset and length is the fact that a file can be very long and may not fit in the buffers allocated in this PA. For instance, if the file is 20GB long and if you must send the file in a single message, that message must be 20GB long, requiring the same amount of physical memory and bogging the server down. To avoid this, we set the limit of each transfer by the variable called `buffercapacity` in both `client.cpp` and `server.cpp`. This variable defaults to the constant `MAX_MESSAGE` defined in `common.h`. However, you can change that providing the optional argument `-m` as follows, which changes capacity to 5000 bytes:

```
$ ./client -m 5000
```

Note that the change must be done for both client and server to make it effective (e.g., seeing faster/slower performance).

Therefore, instead of requesting the whole file, you just request a portion of the file where the bytes are in range [offset, offset+length]. As a result, you can allocate a buffer that is only `length` bytes long, but use multiple packets to transfer a single file.

Furthermore, a client would not know the length of a file unless the server informs. To achieve that, the client should first send a special file message by setting `offset` and `length` both to 0. In response, the server just sends back the length of the file as a `__int64_t`. Note that `__int64_t` is a 64-bit integer which is necessary for files over 4GB size (i.e., the max number represented by an unsigned 32-bit integer is 2^{32} = 4GB). From the file length, the client then knows how many transfers it has to request, because each transfer is limited to max `MAX_MESSAGE`.

Also, note that the requested filename is relative to the `BIMDC/` directory. Therefore, to request the file `BIMDC/1.csv`, the client would put "1.csv" as the file name. The client should store the received files under `received/` directory and with the same name (i.e., `received/1.csv`). Furthermore, take into account that you are receiving portions of the file in response to each request. Therefore, you must prepare the file appropriately so that the received chunk of the file is put in the right place.

The following is an example request for getting file "10.csv" from the client command line:

```
$ ./client -f 10.csv
```

where the argument "-f" is for specifying file name.

Requesting New Channel Creation

The client can ask the server to create a new channel of communication. This feature will be implemented in this PA and used extensively in the following ones when you write multi-threaded client. The client sends a special message with message type set to `NEWCHANNEL_MSG`. In response, the server creates a new request channel object, returns the name back, which the client uses to join into the same channel. This is shown in the server's `process_new_channel` function.

The following is a request a new channel:

```
$ ./client -c
```

.

Your Task

The following are your tasks:

- *Requesting Data Points:* (15 pts) Request all data points for person 1 by (both ecg1 and ecg2), collect the responses, and put them in a file called `x1.csv`. Compare the file against the original `BIMDC/1.csv` using a file compare tool (e.g., `fc`) and demonstrate that they are exactly same. Also, measure the time do the entire thing by using `gettimeofday` function.
- *Requesting Files:* (35 points)

  - (20 pts) Request an entire file by first sending a file message to get its length, and then a series of file messages to get the actual content of the file. Put received file under the `received` directory with the same name as the original file. Compare the file against the original using linux command `diff` and demonstrate that they are exactly same. Measure the time for the transfer.
  - (10 pts) Make sure to treat the file as binary, because we will use this same program to transfer any type of file. Putting the data in a STL string will not work, because C++ strings are NULL terminated. To demonstrate that your file transfer is capable of handling binary files as well, make a large empty file under the `BIMDC/` directory using the `truncate` command (see man pages on how to use `truncate`), transfer that file, and then compare to make sure they are identical using the `diff` command.
  - (5 pts) Experiment with transferring larger files (e.g., 100MB), and document required time. What is the main bottleneck here? Can you change the transfer time by varying the bottleneck? [Hint: the most likely bottleneck is buffer capacity]

- *Requesting a New Channel:* (15 pts) Ask the server to create a new channel for you by sending a special `NEWCHANNEL_MSG` request and join that channel. After the channel is created, demonstrate that you can use that to speak to the server. Sending a few data point requests and receiving their responses is adequate for that demonstration.

- *Run the Server as a child process* (15 pts) Run the server process as a child of the client process using `fork()` and `exec()` such that you do need two terminals: 1 for the client and another for the server. The outcome is that you open a single terminal, run the client which first runs the server and then connects to it. Also, to make sure that the server does not keep running after the client dies, sent a special `QUIT_MSG` to the server and call `wait()` function to wait for its finish.

- *Closing Channels*worth 5 pts You must also ensure that there are NO open connections at the end and NO temporary files remaining in the directory either. The server would clean up this resources as long as you send `QUIT_MSG` at the end. This part is worth 5 points. Note that the given `client.cpp` already does this for the control channel.

- *Report* (15 points): Write a report describing the design, and the timing data you collected for data points, text file, and binary files. Compare the difference in time between transferring data points vs entire file.

2 Submission Instruction

Put everything excluding the `BIMDC` directory in a single directory, zip it and submit on ecampus. Do not forget to make necessary changes to the `makefile` should you decide to add other .h/.cpp files. Please do not include any data file because that will make your zip file very large. Make sure that your directory has everything needed to compile your program.