A Client Process Speaking to a Server Process

Due: 2/19/19 Tuesday 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 goal is to obtain some electrocardiogram (ecg) data points of several patients suffering from various cardiac diseases. In addition to sending data points in units, the server supports a set of commands to send a range of data packed in files. Your client must implement this file transfer functionality as well.

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 communicate with another process. This class has a read and a write function to exchange data. You can just use this class talk back and forth with the server. The usage of the function is demonstrated in the given `client.cpp`. No change in this class is necessary for PA2.
- A `dataserver.cpp` that contains the server logic. When compiled with the `makefile`, an executable called `dataserver` 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 implement the client 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 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 potentially shared between the server and the client. For instance, if you decide to create classes for define 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. Then, run the `dataserver` to start the server. After that, 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. You can run the client a few times while keeping the server running.

Server Specification

The server supports several functionalities. A client can request for a certain functionality by sending appropriate messages 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

First, connecting to the server. In the dataserver main function, you will see the following:

```
FIFORequestChannel control_channel(“control”, FIFORequestChannel::SERVER_SIDE);
```

which sets up a communication channel over “named pipe”, which is one of several methods of inter-process communication. Note that the first argument in the channel constructor is a string that indicates the name of the channel. To connect to this server, the client has to create an instance with the same name, but as the “client side” as is done by the given client:

```
FIFORequestChannel control_channel(“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 servers maintain ecg values (at 2 contact points) for each patient in time series where there is one data point collected every 4ms (see any of the .csv file under the BIMDC/ directory) for a minute. That means there is 15K data points for each patient kept in separate data file for each patient.

The client requests a specific 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. Again the data type is an 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`. One of you tasks is to prepare and send a data message to the server and collect its response.

Requesting Files

To request a file, you need package the following information in the 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.
- How many bytes to transfer starting 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` encodes these information. However, you won't see a field for the file name, because it is a variable length field and thats why it does not get an entry. 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. To avoid this inefficiency, we set the limit of each transfer by the constant `MAX_MESSAGE` defined in `common.h`. Therefore, instead of requesting the whole file, you just request a portion of the file indicated by the offset and length. As a result, you can allocate a buffer that is only `length` bytes long.

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. This is understood by the server and 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. From the file length, the client then knows how many times it has to request a regular file message, because every transfer is limited by `MAX_MESSAGE` constant, which tells how big the uf

Also, note that the 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. To do that, you would open the file properly, seek to the correct offset, and then write the content there.

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 way to do that is sending a special message with message type set to `NEWCHANNEL_MSG`. In response, the server creates a new request channel object, which the client can then join just by using the appropriate name. This is shown in the dataserver's `process_new_channel` function.

Your Task

The following are your tasks:

- *Requesting Data Points:* (worth 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 a Text File:* (worth 20 pts) Request an entire file by first sending a message to get its length, and then a series of messages to get the actual content of the file. Put the content of the flie in a single output file called `y1.csv`. Compare the file against the original and demonstrate that they are exactly same. Also, measure the time to obtain the file.

- *Requesting a Binary File:* (worth 15 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 C++ STL string will not do, because C++ strings are also 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 using `truncate`), transfer that file, and then compare to make sure they are identical.

- *Requesting a New Channel:* (worth 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 also speak to the server using that new channel. Sending a few data point request and receiving their response is adequate for that demonstration.

- *Run the server as a child process* (worth 20 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.

  You must also ensure that there are NO temporary files remaining in the directory. The server would clean up this resources as long as you send `QUIT_MSG` at the end. This part is worth 5 points of the score.

- *Report* (worth 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.

  For the binary file, experiment with a large 5GB file, and document how much time that transfer takes. What is the bottleneck here? Can you change the transfer time by varying the bottleneck? To what extent. Provide supporting evidence.

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.