PA 1:
Client-Server Communication using Named Pipes

Introduction

In this assignment, you will write a client program that connects to a given server.

The server hosts electrocardiogram (ECG) data points of 15 patients suffering from various cardiac diseases. The client has to communicate with the server such that it can fulfill two main objectives:

1. Obtain individual data points from the server.
2. Obtain a whole raw file of any size in one or more segments from the server.

The client has to send properly-formatted messages to the server using a communication protocol defined by the server to implement this transfer functionality.

The above-referenced dataset was obtained from physionet.org.

Starter Code

You are given a source directory with the following files:

- A ***makefile*** that compiles and builds the source files when you type the *make* command in the terminal. We do not expect you to edit the provided *makefile* in this PA but if you do edit the makefile for a justifiable reason, please document the reasoning in your report. For a detailed introduction to makefiles, you can refer to the Lab1B's Advanced Makefile Primer.

- The ***FIFORequestChannel*** **class** (`FIFORequestChannel.cpp/.h`), that implements a pipe-based communication channel. The client and server processes use this class to communicate with each other. This class has a read and a write function to receive and send data from/to another process, respectively. The usage of the function is demonstrated in the given `client.cpp`. No change in this class is necessary for PA1.

- The **server program** in `server.cpp` 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 no changes to this file are required in PA1, you will need 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` in its current starter code state is capable of connecting to the server using the *FIFORequestChannel* class; the client sends a sample message to the server and receives a response.
  When compiled with the makefile, an executable file **client** is generated. You need to run this executable to start the client. You will make most of the changes needed for this programming assignment in the client program.

- The **`common.h` and `common.cpp`** files that contain different useful classes and functions shared between the server and the client. Classes for different types of messages (e.g., a data message, a file message) are defined here. You are not required to change this class.

Server Specification

The server supports several functionalities. The client requests a certain functionality by sending the appropriate message to the server through a FIFORequestChannel class. Internally, the server will execute the correct corresponding functionality, prepare a response for the client, and send it back through the same channel.

Connecting to the Server

You will see the following in the server main function:

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

Note that the first argument in the channel constructor is the name of the channel, and the second argument is the side (server or client) that is connecting to the channel.

To connect to the server, the client has to create an instance with the same name, but with CLIENT_SIDE as the second argument:

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

This sets up a communication channel over an OS-provided IPC mechanism called “named pipe”. Named pipes are created by the system call `mkfifo`. They are used by processes to receive (read system call) and send (write system call) information to one another. For the purposes of PA1, the client would have to call the `cread` and `cwrite` functions appropriately to communicate with the server. For more on “named pipes”, refer to https://man7.org/linux/man-pages/man7/fifo.7.html.

After creating the channel, the server then goes into an “infinite” loop that processes client requests. Note that the client and the server can connect with each other using several channels.

Data Point Requests

The server hosts the BIMDC directory which contains 15 files (1.csv - 15.csv), one for each patient. The files contain ECG records for a duration of one minute, with a data point every 4 ms, resulting in a total of 15000 data points per file. A particular row (data point) in any of these CSV files is represented in the following format:

time (s), ecg1, ecg2

You will find the request format in `common.h` as a datamsg. The client requests a data point by constructing a datamsg object and then sending this object across the channel through a buffer. A datamsg object is constructed with the following fields:

- **Patient ID** is simply specified as a number. There are 15 patients total. The required data type is an int with a value in the range [1,15]
- **At what time** in seconds. The type is a double with range [0.00,59.996]
- **Which ECG record**: 1 or 2, indicating which record (ecg1 or ecg2) the client should be sent. The data type is an integer.

Note that **the message type** field MESSAGE_TYPE is implicitly set to a constant, *DATA_MSG.* Both the message type and its possible values are defined in `common.h`.

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 59.004 -e 2
```

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

An appropriate datamsg object constructed would therefore be:

```
datamsg dmsg(10, 59.004, 2); // [replace with variables holding these
                               values]
```

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.

File Requests

Let us first understand the role buffer capacity plays in file requests. If, for example, we were to transfer a large file of 20 GB in one go, the message sent across the channel and therefore physical memory required will also be 20 GB, 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 (256 Bytes) defined in common.h. However, the user can change this value by providing the optional argument -m to any command as follows:

$ ./client -m 5000

In this example, the buffer capacity is changed to 5000 bytes. Note that the change must be done for both client and server to make it effective (e.g., seeing faster/slower performance). We can request the file in segments through chunks referenced by the corresponding byte number intervals in the following way:

[0 - 5000), [5000 - 10000), [10000 - 15000),.....

In this particular example when transferring the chunk of 10000 - 15000 Bytes, our offset is 10000 and the length we are transferring is 5000 Bytes. Therefore, instead of requesting the whole file, you may just request each 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.

To request a file, you (the client) will need to package the following information in a message:

- **Starting offset in the file**. The data type is __int64_t because of the fact that the usual 32-bit integer might not be sufficient to represent files that may be very large.

- **How many bytes to transfer beginning from the starting offset.** The data type is int. If you are transferring a file larger than a 32-bit integer you must request it in chunks using the offset parameter for the reasons mentioned above.

- **The name of the file** as a NULL-terminated string, relative to the directory BIMDC/

Note that **the message type** field MESSAGE_TYPE is implicitly set to a constant, *FILE_MSG.* Both the message type and its possible values are defined in `common.h`.

For example, to retrieve 30 bytes from a file at an offset of 100 you would construct a filemsg object:

```
filemsg msg(100,30);
```

Here, offset is the first parameter and length is the second parameter. Note that you could also set the offset and length by setting msg.offset and msg.length to the desired value. The type filemsg in `common.h` encodes this information. When sending a message across the channel to the server, we can then send a buffer that contains the filemsg object, and the name of the file we are attempting to transfer (as a NULL-terminated string) following the filemsg object. The server responds with the appropriate chunk of the contents of the requested file. 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 to determine at compile time. You can just think of the file name as variable-length payload data in the packet that follows the header, which is a filemsg object.

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 the 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.

Let us consider the following case: The client attempts to transfer a file of size 400 Bytes, and the buffer capacity is 256 Bytes. In our first transfer we would set the offset to 0, and length to 256. In the next transfer, we would have to set the offset to 256 and the length to 144. In this transfer (which is also the last transfer), the client would have to know the whole size of the file prior to the transfers so it can make appropriate adjustments to the length in the last transfer. Therefore, it must initially send a message to the server asking for the size of the file.

To achieve this, 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 the `buffercapacity`.

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

$ ./client -f 10.csv

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

New Channel Creation Request

The client can ask the server to create a new channel of communication. The flag used to create a new channel can be used in tandem with any other command. All communication for that execution will occur over the new channel. Note that this feature will be implemented in this PA and then used extensively in the later programming assignments when you write a multi-threaded client. The client sends a special message with the message type set to NEWCHANNEL_MSG. In response, the server creates a new request channel object, returns the channel 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 an example of a new channel being requested to transfer file "5.csv":

$ ./client -c -f 5.csv

Your Tasks

The following are your tasks:

- ***Run 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 not need two terminals. 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, send a QUIT_MSG to the server for each open channel and cal thel `wait(...)` function to wait for its finish. Note that the autograder will fail on all test cases if you do not run the server as a child process of the client. It would be beneficial to implement this functionality first so that the autograder will work; however, you can run them separately on two different terminals to locally test your client functionality.

- ***Requesting Data Points:*** **(15 pts)** First, request one data point from the server and display it to stdout by running the client using the following command line format:

  $ ./client -p <patient no> -t <time in seconds> -e <ecg no>

  You must use the Linux function `getopt(...)` to collect the command line arguments. You cannot scan the input from the standard input using `cin` or `scanf`. After demonstrating one data point, request the first 1000 data points for a patient (both ecg1 and ecg2), collect the responses, and put them in a file called x1.csv. Compare the file against the corresponding data points in the original file and check that they match.

    - For collecting the first 1000 data points of a given patient, use the following command line format:

      $ ./client -p <patient no>

- ***Requesting Files:*** **(35 points)**

    - **(20 pts)** Request a file from the server side using the following command format again using `getopt(...)` function:

      $ ./client -f <file name>

  Note that the file does not need to be one of the *.csv* files currently existing in the BIMDC directory. You can put any file in the *BIMDC/* directory and request it from the directory. The steps for requesting a file are as follows. First, send a file message to get its length, and then send a series of file messages to get the actual content of the file. Put the received file under the *received/* directory with the same name as the original file. Compare the file against the original using the Linux command diff and demonstrate that they are exactly the same. Measure the time for the transfer for different file sizes (you may use the Linux command `truncate -s <s> test.bin` to create a <s> bytes empty file) and put the results in the report as a chart.

    - **(10 pts)** Make sure to treat the file as binary, because we will use this same program to transfer any type of file (e.g., music file, ppt, and pdf files which are not necessarily made of ASCII text). Putting the data in an STL string will not work because C++ strings are NULL terminated. To demonstrate that your file transfer is capable of handling binary files, 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 large files (e.g. around 100MB), and document the 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 joining that channel. Use the command format shown in the example above. 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.

- ***Closing Channels:*** **(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 these resources as long as you send QUIT_MSG at the end for the new channels created. 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 files, and binary files. Show the time of file transfers as a function of varying file sizes and buffer capacity.

Location of the Code

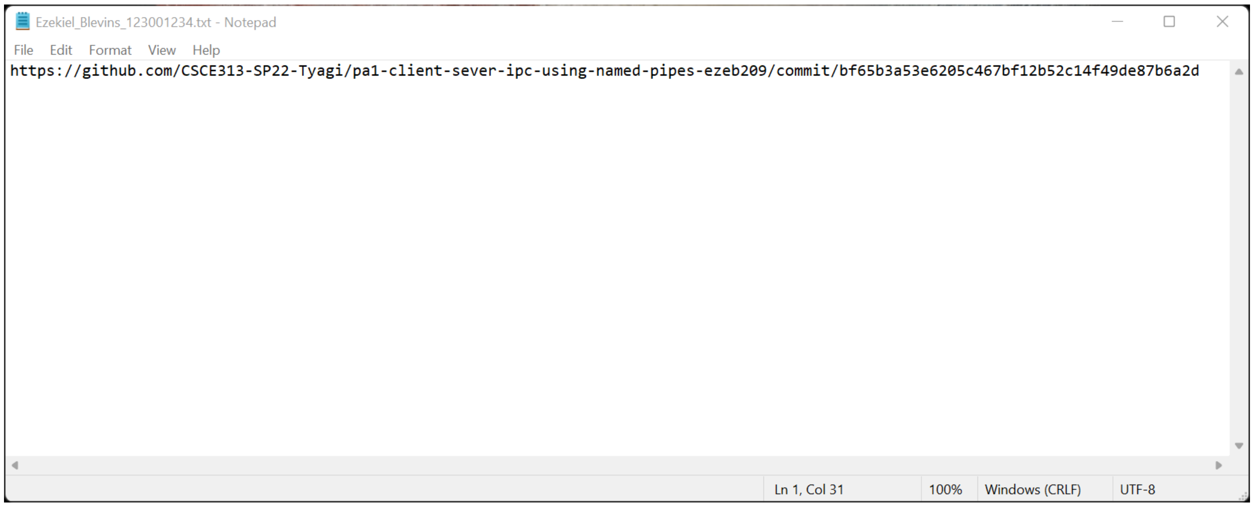
This assignment will be hosted using the **GitHub Classroom** platform. You must visit https://classroom.github.com/a/IeMKck9Z to create a repository for your project.

How to begin

Start this programming assignment by cloning the repo created for you by GitHub classroom. Open a terminal, navigate to the directory, and then build using the make command. After that, run the executable `./server` to start the server. Now, open another terminal, navigate to the same directory, and run the executable `./client`. At this point, the client will connect to the server, exchange a simple message, and then the client will exit. Since the pipe is a point-to-point connection, when either the client or the server exits, the other side will exit as well after receiving the SIGPIPE signal for “broken pipe”. Some of these terms will be introduced in the lectures later in this semester, but for now, these are used for reference only.

What to do when done

Once you have completed the assignment and submitted your final code to GitHub Classroom, on Canvas, turn in a text document named *FirstName_LastName_UIN.txt* containing the link to the commit hash that you are submitting (example below) and your report named *PA1_FirstName_LastName_UIN.pdf*.


Ezekiel_Blevins_123001234.txt - Notepad
File Edit Format View Help
Ln 1, Col 31
100%
Windows (CRLF)
UTF-8

```
https://github.com/CSCE313-SP22-Tyagi/pa1-client-sever-ipc-using-named-pipes-ezeb209/commit/bf65b3a53e6205c467bf12b52c14f49de87b6a2d
```