PA#1: A Client Program Speaking to a Server

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 patient 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 command `./client`. The client internally calls `fork()` and `exec()` to launch the `./server` process. After that, the client tries to connect to the server process using an IPC method called *named pipe*. Since pipes establish point-to-point connections, whenever either the client or the server exits, the other side will exit as well and you will see an error message along the lines of "Program terminated after receiving a SIGPIPE signal for a broken pipe".

Server Specification

The server supports several functionalities, which the client can request 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

You will see the following in the server main function:

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

which sets up the "named pipe" or "FIFO". Note that the first argument in the channel constructor is the name of the channel. To connect to this server, the client has to instantiate a channel object 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. 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 setting:

- The request type field `REQUEST_TYPE_PREFIX` set to the constant `DATA_REQ_TYPE`. These are defined in the `common.h`. Each request type (e.g., `DATA_REQ_TYPE, FILE_REQ_TYPE, QUIT_REQ_TYPE`) is used as prefix for a particular type of message that is understood and responded uniquely by the server. To make sure every request has a prefix, each request must be a subclass of the base class `Request` that has the request type as the solitary field.

- 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 `DataRequest`. In response to a properly formatted data message, the server replies with the ecg value as a `double`. If the request is not well-formatted or if there are illegal values in the fields, the server returns a `Request` object with value `UNKNOWN_REQ_TYPE`. Please see the given `main.cpp` for examples of this.

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

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.

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

- Message type set to `FILE_REQ_TYPE` indicating that it is a file request. Data type is `REQUEST_TYPE_PREFIX` defined in `common.h`
- Starting offset in the file. Data type is a 64-bit integer.
- 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 `FileRequest` 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 `FileRequest` 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. To avoid the risk of overrunning the buffer, we set the limit of each transfer (i.e., request and response) by the variable called `buffercapacity`. Therefore, instead of requesting the whole file, you must request a portion of the file where the bytes are in range $[offset, offset + buffercapaity]$, put the obtained range into the file, and then request the next range. As a result, your program never takes more than `buffercapaity` bytes while transferring arbtrarily large files.

This `buffercapacity` variable defaults to the constant `MAX_MESSAGE` defined in `common.h`. However, you can override that by providing the optional argument `-m` as follows, which requests the `example.pdf` file in chunks of 5000-bytes:

```
$ ./client -m 5000 -f example.pdf
```

Note that the client and the server must use the same `buffercapacity` value;; otherwise the server would error-out by reporting mismatched buffer sizes. Therefore, if you change the buffer capacity in the client side, make sure to change it on the server side as well by passing an additional argument to the server when you invoke it by `execvp()`. Your program runtime should be a function of `buffercapacity`, because with a higher value, you will neee fewer transfers.

Furthermore, a client must get the length of the file from the server, because we assume that the client does not have to access to the server's filesystem, which in real-life might be situated in a remote machine. To get the file size, the client should first send a special file request by setting `offset` and `length` both to 0. In response, the server just sends back the length of the file as a `int64`. From the file length, the client then calculates knows how many transfers it has to request, because each transfer is limited to `MAX_MESSAGE`.

Also, note that the requested filename is relative to the `BIMDC/` directory meaning that to request the file `BIMDC/1.csv`, the client would put "1.csv" as the file name instead of "BIMDC/1.csv". The client should store the received files under `received/` directory and with the same name (i.e., `received/1.csv`).

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 `NEWCHAN_REQ_TYPE`. In response, the server creates a new request channel object, returns the name back, which the client uses to join into the same channel. Please take a look at the server's `process_new_channel` function to understand the process.

The following is a request a new channel and then do something (i.e., transfer the `5.csv` file) through the newly created channel:

```
$ ./client -c -f 5.csv
```

.

Another instance of creating and using a new channel is the following:

```
$ ./client -c -p 15 -t 10 -e 2
```

, which obtains a datapoint for patient 15 through a new channel.

Your Task

The following are your tasks:

- *Requesting Data Points:* (25 pts) First, request one data point from the server by running the client using the following command line format:

  ```
  $ ./client -p <person 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` either. After demonstrating one data point, request 1000 data points for a person (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 demonstrate that they match. Also, measure the time for collecting data points using `gettimeofday` function, which has microsecond granulity and put the result in the report.

- *Requesting Files:* (40 points) 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 that. The steps for requesting a file 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 linux command `diff` and demonstrate that they are exactly same. Measure the time for the transfer for different file sizes and put the results in the report. A graph is best suited for such presentation.

  In addition to transferring regular text files, your program should be equally capable of handling binary files. For instance, your program should correctly transfer any file type including music files, ppt, pdf, docx etc. Putting the data in a STL string will not work because C++ strings are NULL terminated and thus will truncate the content whenever encountering a NULL. 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. Furthermore, you should try `.pdf` or `.docx` files and `diff` them.

- *Requesting a New Channel:* (15 pts) Ask the server to create a new channel for you by sending a special `NEWCHAN_REQ_TYPE` request and join 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 this resources as long as you send `QUIT_REQ_TYPE` at the end. This part is worth 5 points. Note that the given `client.cpp` already does this for the main control channel.You should simply repeat that for all new channels you create.

- *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 function of varying file sizes and buffer capacity `m`. Follow the grading instruction to demo your work, record that in a youtube video and include the link to the video in the report. However, you can demo your work before the due date to waive the video demo requirement.

2 Submission Instruction

Put everything about this PA in a single directory, do a `make clean` to remove any executable or object files, zip and submit on canvas. Make sure that your directory has everything needed to compile and run your program. Do not include the large test files.