Programming assignment 1

Server client communication using named pipes

Table of content

Academic Integrity

The following actions are strictly prohibited and violate the honor code. **The minimum penalty for plagiarism is a grade of zero and a report to the Aggie honor system office.**

- Uploading assignments to external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from a classmate or unauthorized sources and submitting it as your work.

- Sharing your code with a classmate.

Keywords

Processes, files, named pipes, FIFO

Introduction

Write a client program that connects to a server. The server hosts electrocardiogram (ECG) data points of 15 patients suffering from cardiac diseases. The client communicates with the server to complete two tasks:

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.

Starter Code

You are given a source directory with the following files:

- makefile (makefile)

This file compiles and builds the source files when you type the make command in the terminal.

- FIFORequestChannel class (FIFORequestChannel.cpp/.h)

These files implement 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. **Do not modify this class.**

- Server program (server.cpp)

This file contains the server logic. When compiled with the makefile, an executable server is created. Run this executable to start the server. Refer to the server to understand the server protocol and then implement the client functionality based on that. **Do not modify this program**.

- Client program (client.cpp)

This file contains the client logic. The starter code can connect 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 client is created. Run this executable to start the client. **You will make most of your changes in the client program.**

- Utilities (common.h and common.cpp)

These files contain 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. **Do not modify this class**.

Server Specifications

The client requests the server some functionality by sending the appropriate message to the server through a FIFORequestChannel class. 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);
```

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);
```

The two lines above belong to the FIFORequestChannel class, and set up a communication channel over an OS-provided IPC mechanism called a FIFO or a named pipe. Named pipes are created by the system called **mkfifo**.

They are used by processes to receive (read system call) and send (write system call) information to one another. 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. The client and the server can connect with each other using several channels.

Data Point Requests

The server contains the BIMDC directory which includes **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:

- -p 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].
- -t Time in seconds. The type is a double with range [0.00,59.996].
- -e ECG record: 1 or 2, indicating which record (ecg1 or ecg2) the client should be sent. The data type is an integer.

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
```

An appropriate datamsg object constructed would therefore be:

```
datamsg dmsg(10, 59.004, 2);
```

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 transfer a large file of 20 GB in one transaction, the message sent across the channel or the physical memory required will also be 20 GB. This will slow down the server.

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.

The user can change this value by providing the optional argument -m to any command. In the example below, the buffer capacity is changed to 5000 bytes.

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

The change must be done for both the 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 a 32-bit integer is not sufficient to represent large files.
- How many bytes to transfer beginning from the starting offset. The data type is int. To transfer 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/

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. You can 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. To use a data type, you 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, 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.

Consider this 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. 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. __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 with any other command. All communication for that execution will occur over the new channel. This feature will be implemented in this assignment 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 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 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.
- 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 call the wait(...) function to wait for its end.

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 works; 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 number>
```

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, r they match.

For collecting the first 1000 data points of a given patequest the first 1000 data points for a patient (both ecg1 and ecg2), collect the responses, and put them in a file named x1.csv. Compare the file against the corresponding data points in the original file and check thatient, use the following command line format:

```
$ ./client -p <patient number>
```

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>
```

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.

1. First, send a file message to get its length.
2. Next, send a series of file messages to get the content of the file.
3. Put the received file under the received/ directory with the same name as the original file.
4. Compare the received file against the original file using the Linux command diff and demonstrate that they are identical.

5. 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 files, 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 that they are identical using the diff command.

(5 pts) Experiment with transferring a large file (100MB), and document the required time. What is the main bottleneck here? Submit your answer in the repository in a file **answer.txt**

Requesting a New Channel: (15 pts)

Ask the server to create a new channel 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, you need to process the request the user passed in through the CLI options.

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 should clean up these resources as long as you send QUIT_MSG at the end for the new channels created. The given client.cpp already does this for the control channel.

Getting started

1. Go to the assignment's GitHub classroom: https://classroom.github.com/a/XHbjTopT
2. Clone your assignment repository on the Linux machine
3. Open a terminal, navigate to the directory, and then build using the make command.
4. Run the executable **./server** to start the server.
5. 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. Starter video (credit to Mary Julian): https://www.youtube.com/watch?v=HRSoiUXPdHA

Q&A

1. How do I verify whether the file transferred from the BIMDC directory is identical to the one in the received directory?

You can use the diff Linux command to check the difference between 2 files.

For example, to check the difference between file “9.csv” that exists under both the received and the BIMDC directory, you would run the following command in the shell:

```
diff received/9.csv BIMDC/9.csv
```

The absence of an output upon running this command means that the files are identical. An output indicates the difference between the two files. Additionally, to compare the first 1000 lines of the two files, include the head command as follows:

```
diff -sqwB <(head -n 1000 BIMDC/9.csv) received/9.csv
```

2. Do I need to manually create the received directory to store my files?

You can create this folder manually or during program execution.

3. When requesting a new channel, the document says to use the “-c” flag. Does this “-c” flag have any value associated with it?

No, the c flag does not take in any value associated with it. It is an indication for the client to request the server for a new channel. Upon a successful new channel request, the server creates a new channel and returns its name. The client can join this new channel using this returned name.

4. I don’t understand this piece of the starter code well. What does it mean?

```
filemsg fm(0, 0);
string fname = "teslkansdlkjflasjdf.dat";
int len = sizeof(filemsg) + fname.size() + 1;
char buf2[len];
memcpy(buf2, &fm, sizeof(filemsg));
strcpy(buf2 + sizeof(filemsg), fname.c_str());
chan.cwrite(buf2, len);
```

The objective of this code is to send a message to the server asking for the size of the file “teslkansdlkjflasjdf.dat”. To do this - we construct a filemsg object with offset and length parameters set to 0 and append the filename to this filemsg object; we then send this (filemsg object + filename) across a character buffer to the server. The size of the buffer (len) is sizeof(filemsg) + fname.size() + 1; we add 1 for the ‘\0’ character which indicates the end of the character array (this is implicitly added for you, you must not explicitly add this character to the end of the filename).

The purpose of memcpy is to copy the filemsg object to the character buffer such that it occupies the first (sizeof(filemsg)) bytes in the buffer. We then use strcpy to copy over our filename to occupy the rest of the buffer. Note the buf2 + sizeof(filemsg)parameter in strcpy - this indicates that we want to copy our string starting from the byte after the end of the filemsg object (so that we do not overwrite the filemsg object present in the buffer itself).

5. I see strange/random characters in the file transferred to the received/ directory. What could be the reason for this?

It's likely that you're writing the received data as a string rather than using the fwrite(...) system call to transfer over binary files. Another reason could be that you are putting in an incorrect offset or length parameter.

6. Should I handle incorrect / invalid inputs in my code?

No, you can assume all inputs are valid.

7. When running the server as a child process of the client, how do I pass the command line arguments (specifically -m for the common buffer capacity) to the server?

You must use execvp(...) for this functionality.

8. My autograder passes on one type of test case but fails on another similar one. The error message is exit with error code: 1 and signal null. What could be the reason for this?

There was a "nonsense" filemsg implemented for you in the starter code. That file doesn't exist in the BIMDC directory, and it may lead to issues when you try using the channel to send and read some other message after this section of the code. Try commenting this "nonsense" section out and rerun your autograder.

Another reason could be that you cloned your repo directly to Windows instead of Linux - this causes the encoding of files to change from LF to CRLF, meaning that your files are most likely locally stored in NTFS. The UNIX commands don't recognize '\r' (carriage return or CR) as a delimiter and your comparison ends up being "-0.685\r" to "-0.685", hence the discrepancy. The fix to the following problem: Open all CSV files in VSCode, click CRLF in the bottom right, and choose LF, instead.

9. My multiple data point transfer is working fine, but it is not passing the autograder script. Am I missing out on something?

You're likely saving it to a file called x<person>.csv - you should be saving it to a file called x1.csv regardless of the person you're transferring the data for. Only in the case of a file transfer should you be writing the contents of a file called <person>.csv.

10. What functionality must the new channel implement?

The new channel must be used in place of the control channel to send messages to the server to fulfill the command line arguments passed to the client.

11. When transferring over multiple data points to x1.csv, what format do we use to write the values?

You must match the format of the associated file in the BIMDC directory. You can run the diff command mentioned previously to ensure that your format is correct.

12. The "Client-side is done and exited" message appears in different formats each time a new channel is created. What could be the reason for this?

Each channel outputs a "Client-side is done and exited" message upon quitting. The new channel created is threaded with the control channel for concurrency (we'll learn about this later in class - this topic is relevant for PA 3). This means that the output is non-deterministic so each time you run it, there is a chance that you will see different outputs (i.e. sometimes the message appears twice on the same line, sometimes it appears on separate lines).

13. Why does my program say that I need an argument for the '-c' flag whereas the document says it should not take in any?

You're likely adding a ':' after the c in the getopt(...) function. You would only add a ':' after a character if it has an argument (for example in -m 512, you would say 'm:' in getopt because the argument for m is 512). But the c flag does not have an argument, so in getopt you would not put a ':' after the 'c' character.

14. Why do we add a NULL to the end of the arguments passed into execvp for the server? Is the first element of the argv array always the process name?

The NULL character in the array indicates to execvp that the command line arguments are complete and it can stop looking for them (NULL termination).

Also note the argument array consists of the program name in index 0, after which the "real" command line arguments begin in subsequent indices in the array. The variable (int argc) passed into main indicates the number of command line arguments + 1 (program name in argv[0]).

For example, if ./client -f 10.csv was our command:

argc: 3 (does not include NULL)

argv[0]: "./client"

argv[1]: "-f"

argv[2] : “10.csv”

argv[3]: NULL

15. After running the server as a child process, I get the following message printed out: ./client: invalid option -- 'p'. I don’t get why this is happening?

The invalid option messages are due to the way execvp is set up - sending argv to the server directly (all of the client's command line arguments). But the server is only equipped to handle the m flag in it's own getopt(...) parser, so it doesn't recognize command line arguments like p, t, e. Ideally, you should create a separate argument array where you only specify the m flag and the buffer capacity to pass it to the server.

16. When I run the client, I keep getting the error message: “fifo_control1: no such file or directory”. How do I resolve this?

If you’re using WSL, make sure you’re running the code in your /home directory and not your /mnt directory. To switch from mnt to home, use the command: “cd/home/${USER}”, then you can git clone the local repo into there or use: “cp -r /mnt/<path goes here> ./” to move it .