Davit Gabrielyan

Project 2 Report

Project Explanation

This project aims to create a shared memory based file compression service (TinyFile) and a client library that can be accessed by any user process in the system. The TinyFile service must be capable of handling any type of file content and must support both synchronous and asynchronous calls, with an optional Quality of Service mechanism. The original file content and compressed file content (and any other service data) must be exchanged between clients and the service via shared memory. The project requires the implementation of a TinyFile service, a TinyFile library for accessing the service, and a sample application that demonstrates the functionalities of the library and service by using library API. The TinyFile library must have an initialization function and functions that allow for blocking (synchronous) and non-blocking (asynchronous) calls to the TinyFile service. The Sample Application must demonstrate all functionalities of the library and service.

Design

I decided to design my implementation using the following C tools.

- Msg queues (System V)
- Shm shared memory (System V)
- Semaphores (POSIX)

I implemented it in the following way.

Firstly, the service (server) needs to be up and running by using ./tinyfile <args> command by specifying the segment size and quantity. I create a shared Msg queue that clients can all add their requests too. After that server picks the request from the queue and processes it. The request is a Request struct described in my code. It contains a unique PID of the client process. The server than uses that ID to establish connection with the client over a separate msg queue for exchanging messages. This is called the unique queue.

Then server and client send several rcv and snd messages as described in the figure 1. They give each other permission to work on the shared memory to not overwrite each other. Client and server also exchange the filesize, mem_size and other metadata with each other to know how

to process the file. If the file is too big for the memory for instance, both client and server run a loop with the number of chunks they split the file into and use semaphores to not overwrite each other. This could also be achieved with just using snd and rcv in the message queue but I strive to learn a little bit about semaphore usage so I implemented semaphores as a learning experience.

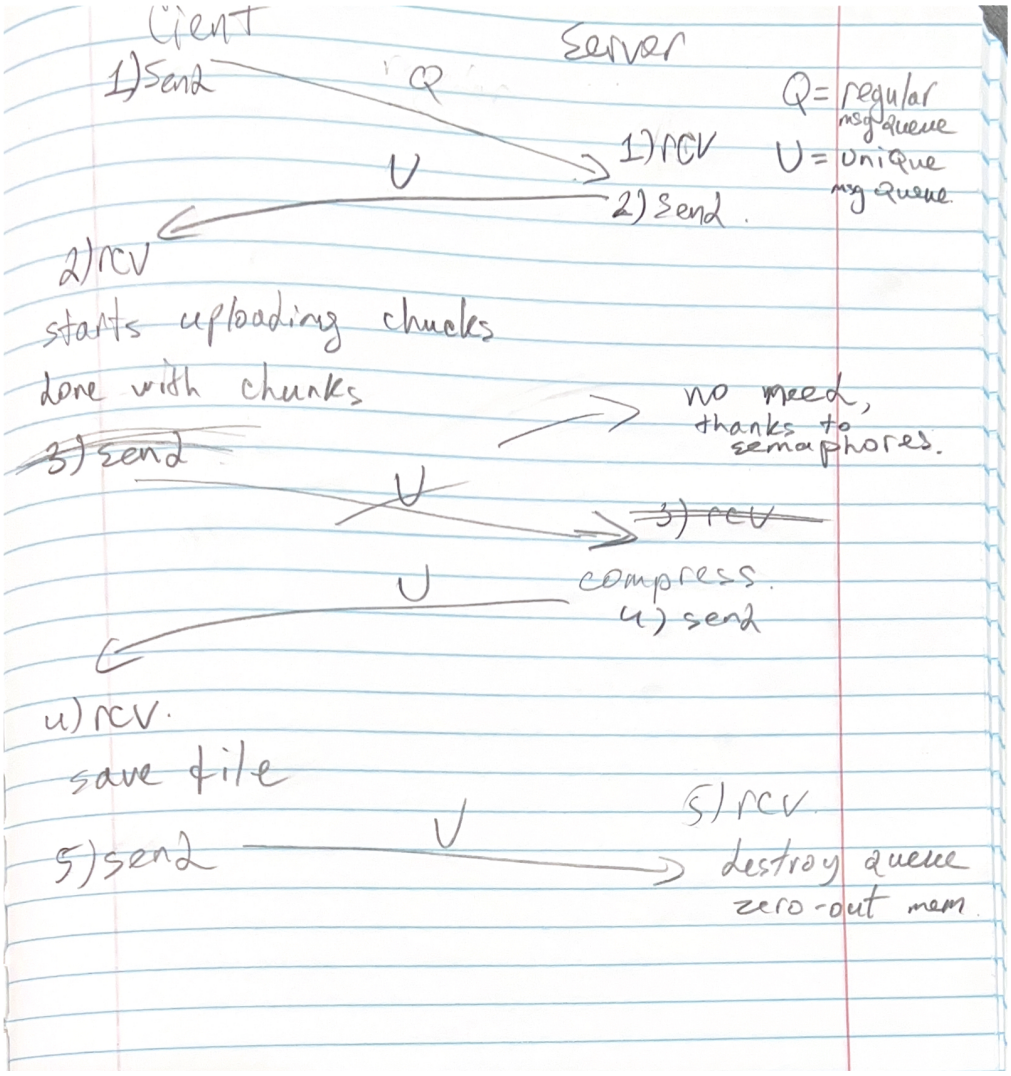


Figure 1, Flow chart illustrating messages sent between client and server.

The server, after receiving all the chunks collects it on a buffer on server side and passes it to snappy-c for compressions. Then it places it in the shared memory and sends a msg to the client that it's ready to download.

Note: There was a lot of discussion on piazza and ultimately it was decided that correctness of the compression doesn't matter since we are using snappy-c. Regardless, I chose to buffer the file on the server side when receiving from client to do accurate compression. But I do not do that for sending it back to the client.

Finally, after client saves file locally, it does a clean-up on both ends. Destroys msg queues, detaches from shared memory etc.

Note: To avoid duplicate code for some of the functionality, I have implemented some of the methods in the shared_mem.c and shared_mem.h to help with this so both library and service can use those methods for the message queues, semaphores, and share memory.

My ASYNC implementation uses multithreading to not block main thread. By TAs suggestion I am printing stuff to illustrate the non-block. Later the threads join and results are printed.

Sample application is very simple, it just uses sys/time to measure the CST time in microseconds and does either SYNC or ASYNC function calls to library.

Experiments and Figures

- sms_size = {32, 64, 128, 256, 512, 1024, 2048, 4096, 8192} bytes
- n_sms = {1,3 and 5}

I ran many experiments. I would primarily execute following commands to test all the input/ files at once.

./sample_app input/Tiny.txt SYNC; ./sample_app input/Large.txt SYNC; ./sample_app input/Small.jpg SYNC; ./sample_app input/Huge.jpg SYNC; ./sample_app input/Medium.pdf SYNC

./sample_app input/Tiny.txt ASYNC; ./sample_app input/Large.txt ASYNC; ./sample_app input/Small.jpg ASYNC; ./sample_app input/Huge.jpg ASYNC; ./sample_app input/Medium.pdf ASYNC

For purposes of this discussion memory size refers to total shared memory size = sms_size * n_sms.

For the purposes of demonstration below I have chosen sms_size and n_sms to be 4096 and 3 respectively.

I observed an obvious trend of CST increasing as the memory size shrank. For both ASYNC and SYNC.

Davit Gabrielyan

```
dgabrielyan3@advos-05:~/proj_2$ ./sample_app input/Tiny.txt SYNC; ./sample_app input/Large.txt SY
NC; ./sample_app input/Small.jpg SYNC; ./sample_app input/Huge.jpg SYNC; ./sample_app input/Mediu
m.pdf SYNC
Starting SYNC CALL for input/Tiny.txt
Size of provided file: 33920
chunk # : 3
compressed_size-> 3057


DONE
===> CST: 1300 microseconds

Starting SYNC CALL for input/Large.txt
Size of provided file: 1048575
chunk # : 86
compressed_size-> 877796


DONE
===> CST: 17220 microseconds

Starting SYNC CALL for input/Small.jpg
Size of provided file: 233024
chunk # : 19
compressed_size-> 14412


DONE
===> CST: 2280 microseconds

Starting SYNC CALL for input/Huge.jpg
Size of provided file: 1936994
chunk # : 158
compressed_size-> 132703


DONE
===> CST: 12017 microseconds

Starting SYNC CALL for input/Medium.pdf
Size of provided file: 532542
chunk # : 44
compressed_size-> 35841
```

Program being ran with 3 and 4096, SYNC

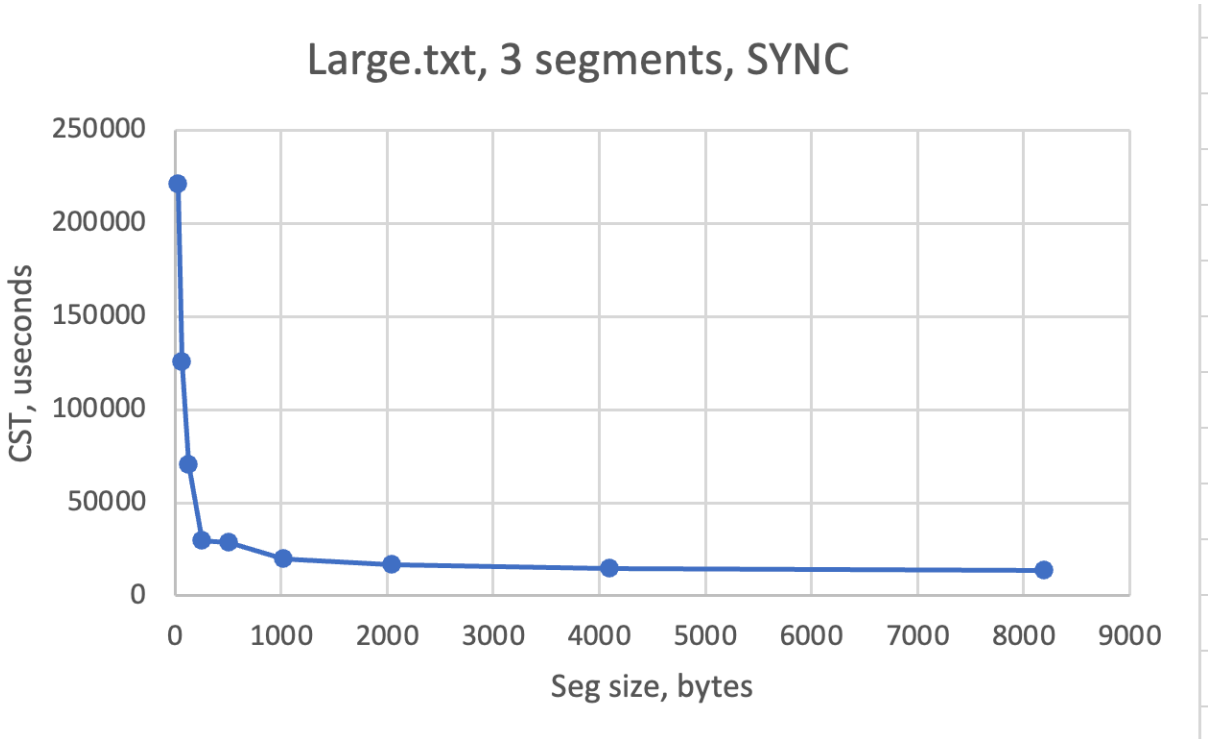
Davit Gabrielyan

```
dgabrielyan3@advos-05:~/proj_2$ ./sample_app input/Tiny.txt ASYNC; ./sample_app input/Large.txt A
SYNC; ./sample_app input/Small.jpg ASYNC; ./sample_app input/Huge.jpg ASYNC; ./sample_app input/M
edium.pdf ASYNC
Starting ASYNC CALL
Size of provided file: 33920
chunk # : 3
compressed file was recieved and saved
compressed_size-> 3057


DONE
===> CST: 3144 microseconds

Starting ASYNC CALL
Size of provided file: 1048575
chunk # : 86
compressed file was recieved and saved
compressed_size-> 877796


DONE
===> CST: 21513 microseconds

Starting ASYNC CALL
Size of provided file: 233024
chunk # : 19
compressed file was recieved and saved
compressed_size-> 14412


DONE
===> CST: 2305 microseconds

Starting ASYNC CALL
Size of provided file: 1936994
chunk # : 158
compressed file was recieved and saved
compressed_size-> 132703


DONE
===> CST: 15753 microseconds

Starting ASYNC CALL
Size of provided file: 532542
chunk # : 44
compressed file was recieved and saved
```

Program being ran with 3 and 4096, ASYNC

There was no significant difference between ASYNC and SYNC for the **same memory** size. TA said this was expected as there is not much other processing going to make a significant dent in CST. However, I did observe that ASYNC was taking slightly longer. This is because of the way I demonstrate the non-blocking by having a for loop printing many lines. So that adds some microseconds by design.

Davit Gabrielyan



Example graph of 3 segments with increasing size

Table of the graph can be seen below. It can easily be observed that segment size improves performance as there are less chucks to deal with for copying back and forth. However, the improvement plateaus with increasing numbers! This is likely because the overhead is a more significant factor than copying time for bigger segment sizes. I can go into more details and show more data during Demo if needed.

Large.txt, 3 segments, SYNC	
Segment size	CST
32	221144
64	125715
128	70501
256	29584
512	28452
1024	19606
2048	16603
4096	14739
8192	13412

Davit Gabrielyan

Final Notes

This was a very scary project for me that took me to very dark places mentally. I am proud of what I accomplished even if it's not perfect. I learned quite a lot during my implementation. The key was to take smaller steps because as a whole it feels overwhelming.

But everything works and I couldn't be happier about that. I have used most if not all tools that were offered/mentioned. Most of them for the first time. I spent a lot of time on this one but once I was getting the hang of things, it was very rewarding. It was a great learning experience.