**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

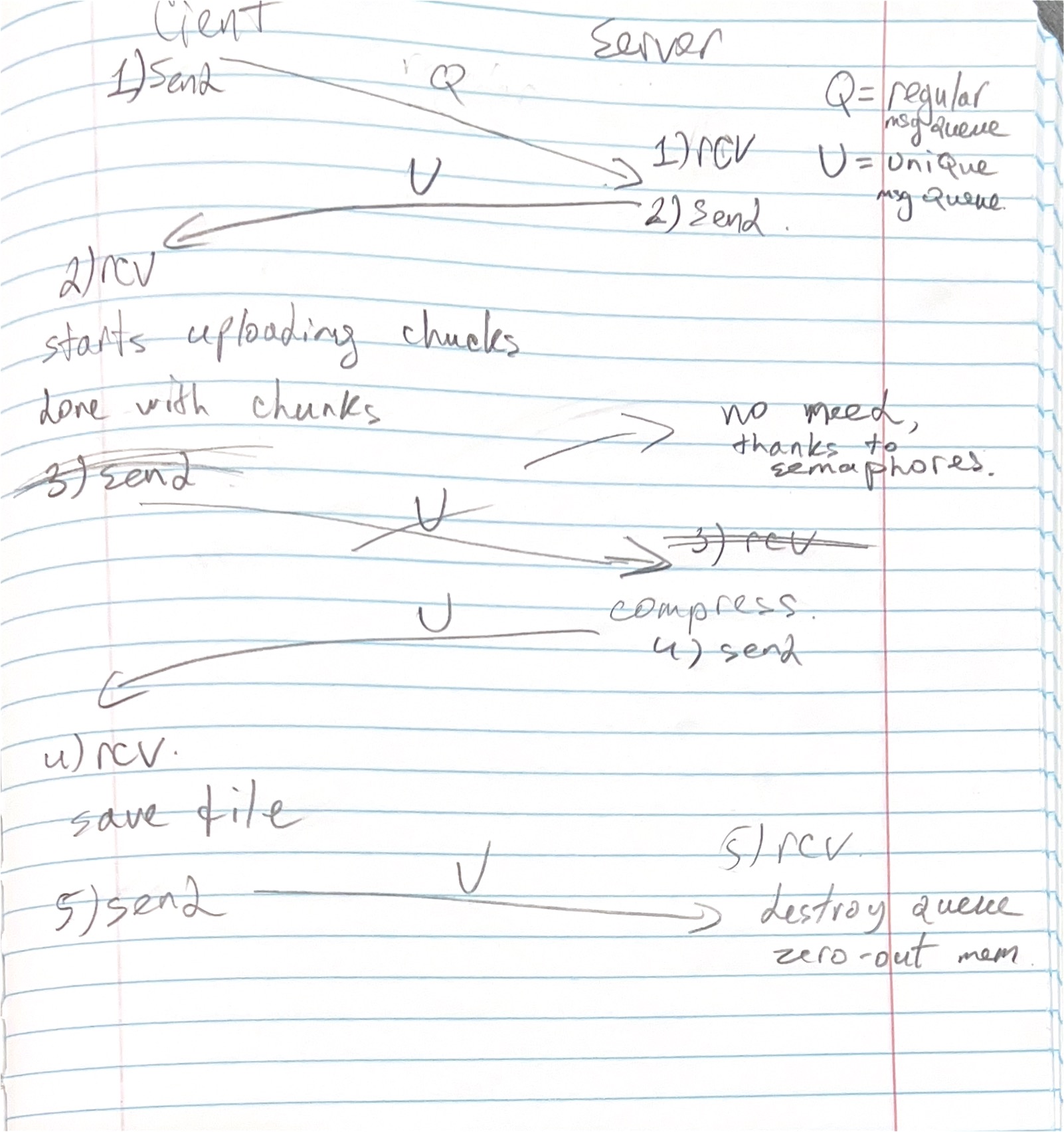
I decided to design my implementation using the following C tools.

* 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.

# Credit Scheduler Implementation

# Experiments and Figures