B06902102
羅費南

Computer Networks Homework Report

The objective of this project is to create a server program and client program that communicate with each other by sending and receiving packets of information. To achieve this, we implement four instructions for the programs to follow: ls, put, get and play. They all work similar in principle, and I will explain how I achieved the results of my program.

Part 1 – File transferring

Transferring files works pretty similar to how you would transfer a normal string over a network. In my program, I transferred strings of characters after reading them from a file in binary mode and sending them using send(), then the other end will receive them using recv(). Since the TCP protocol doesn't guarantee that an entire message will be sent as a whole or that the receiving part will read the entire packet we intended to send, I wrote my own functions, saved in an auxiliary program called "common.cpp", to guarantee that I send and receive all that I want to send and receive, and that both client and server use the same functions. I use the normal send and receive for small instructions and numbers, since the package size is very small; it is almost guaranteed that it will arrive completely to the other end.

Function 1 – sendall():

```
int sendall(int s, char *buf, int *len)
{
    int total = 0;        // how many bytes we've sent
    int bytesleft = *len; // how many we have left to send
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) {
            perror("Send error ");
            break;
        }
        total += n;
        bytesleft -= n;
    }

    *len = total; // return number actually sent here

    return n==-1?-1:0; // return -1 on failure, 0 on success
}
```

Function 2 – rec_all():

```c
int rec_all(int s, char *buf, int *len) {
    int total = 0;  //how many bytes we've read
    int bytesleft = *len;    //how many we have left to receive
    int n;
    while (total < *len) {
        if ((n = recv(s, buf+total, sizeof(char)*bytesleft,0)) <= 0) {
            if (n == 0)
                fprintf(stderr, "Connection closed\n");
            if (n < 0)
                perror("Recv error ");
            break;
        }
        total += n;
        bytesleft -= n;
    }

    *len = total;   // return number actually sent here

    return n<=0?-1:0; // return -1 on failure, 0 on success
}
```

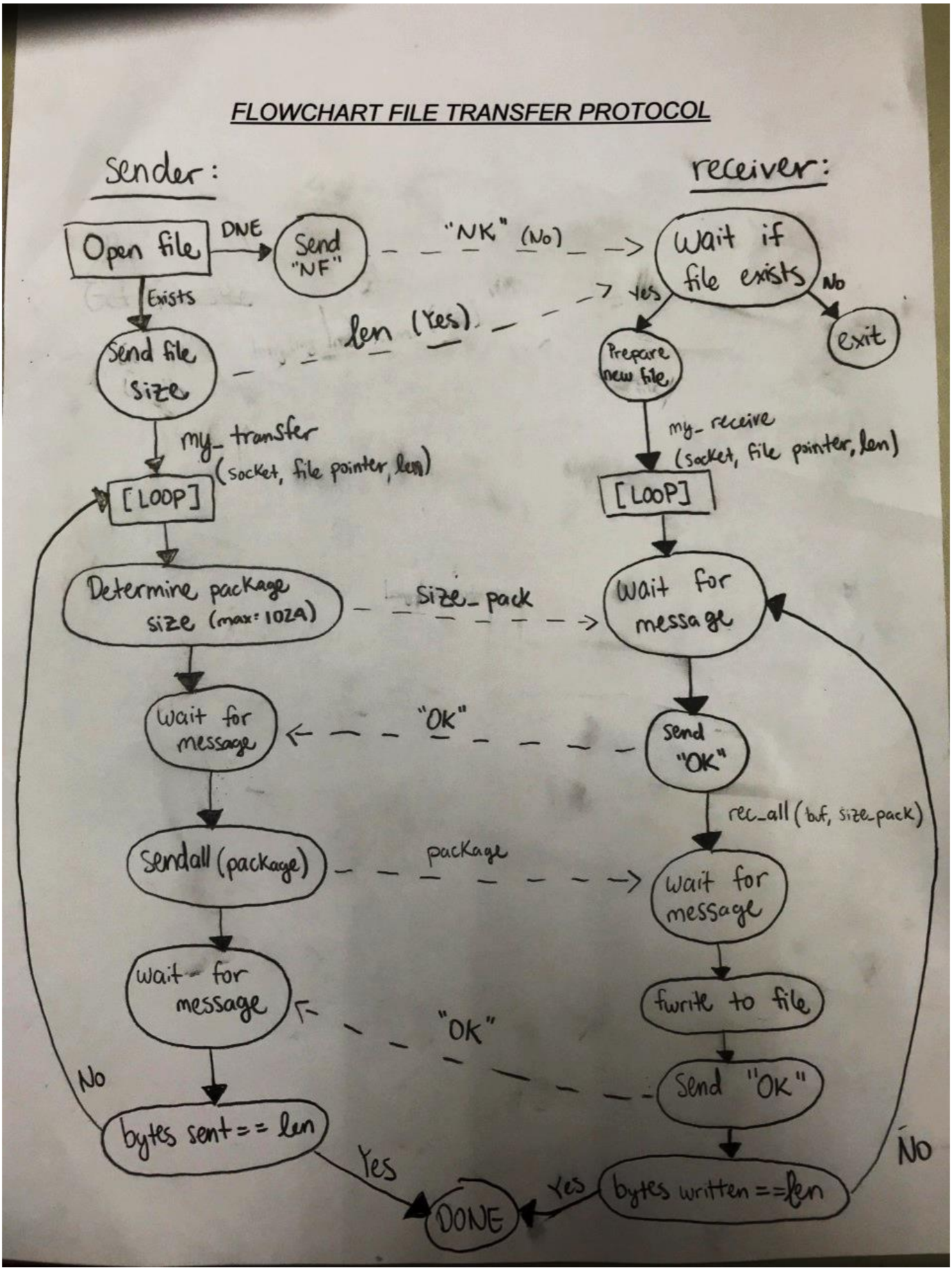
Both client and server use the same functions to transfer and receive the files, called my_transfer() and my_receive() accordingly. I won't paste the code for them in here, but they are also found in "common.cpp".

The only difference between the put and get instructions is who is sending and who is receiving; basically get does the same as put, but it has to first search for the file in server and then send the info of the file to client. The info they send each other before agreeing to transfer the files is the file name and the file size, then they proceed to send the file.

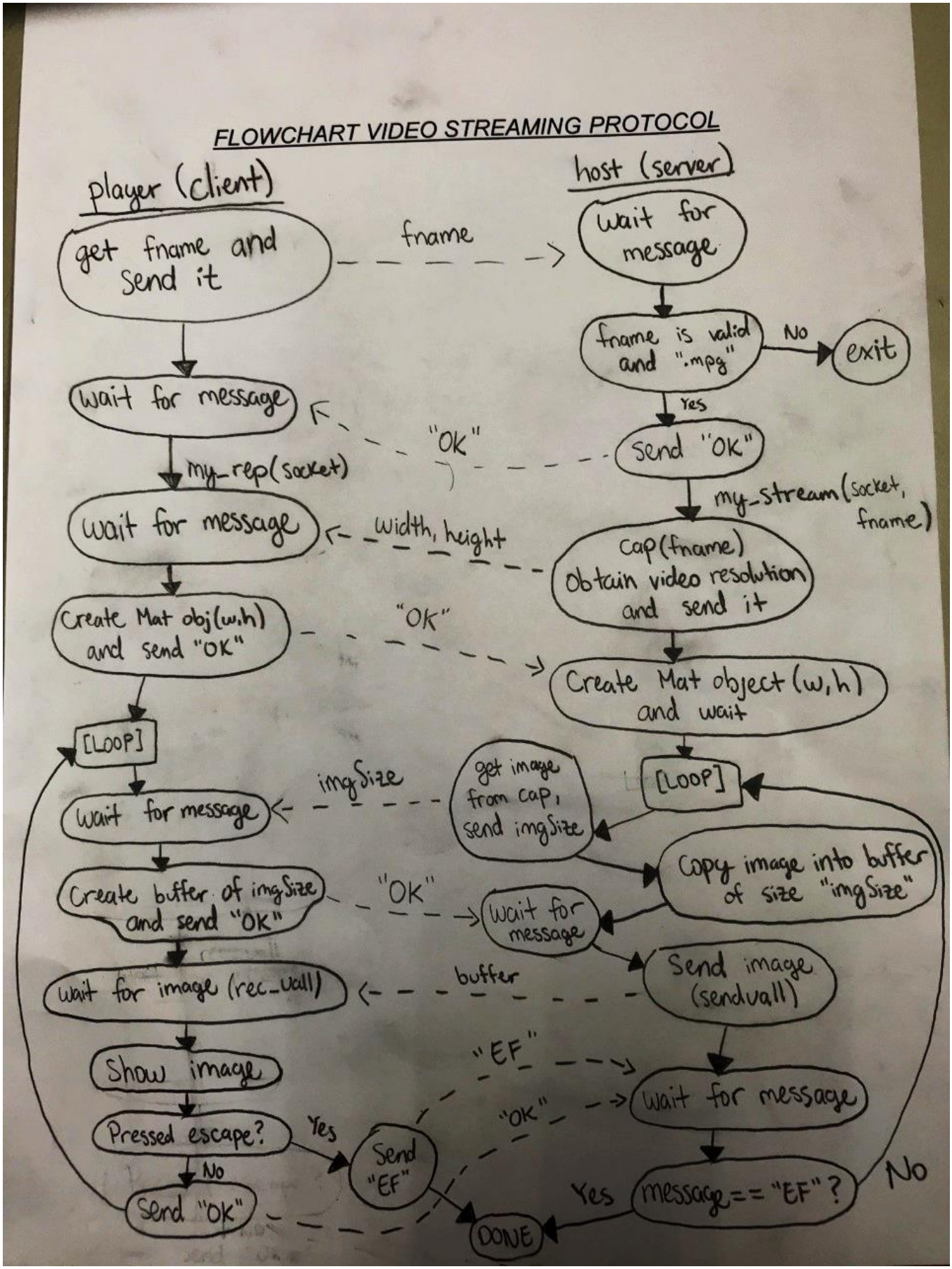
To implement the video streaming protocol, I used different functions for the client and server. In both cases, the protocol is designed so that after sending and receiving images (as arrays of ubites, with predetermined size by the protocol), they send each other confirmation messages and proceed. To send and receive all of them, I used a modified version of my sendall() and rec_all(), called senduall() and rec_uall().

To demonstrate how they work, though, here are the flowcharts of how I achieved my file transferring protocol and video streaming protocol:

B06902102
羅費南



B06902102
羅費南



B06902102
羅費南

Observations:

- SIGPIPE is a signal that the system or program sends when a pipe is closed and one tries to do I/O on it, or when one tries to write to a socket that is no longer available for reading on the other end. It is possible that SIGPIPE is raised in my code, in case that either client or server dies without finishing the operations they were supposed to be currently doing. This can happen by pressing Ctrl-C, sending a kill signal, or any other reason like losing the connection unexpectedly. My program ignores any SIGPIPE (both client and server), and the way it handles the usual errors that would arise from them, I just see if it raises any of those errors when trying to write into a closed socket, then print the error accordingly. If it's the reading sign, recv knows if the socket closes if it reads a 0 value from it, so it handles it as well by seeing if it returns 0.

- Blocking I/O is not equivalent to synchronized I/O. Blocking I/O means if the process halts its operations to wait for an input signal, whereas synchronized I/O means that the process can only focus on one source for I/O operations at a time or many. For example, blocking synchronous I/O (normal read/write) is the traditional I/O system, where the program stops each time it wants to perform an I/O operation, and waits for a response to proceed. Non-blocking synchronous I/O (read/write with the O_NONBLOCK flag activated) means that the process has to ask multiple times for the same I/O operation, but it can do other operations after it fails to perform one of them if the operation is not available. Blocking asynchronous I/O (select or poll) means that the process waits for multiple sources of information and then when one of them is ready, it performs I/O operations on it. Asynchronous non-blocking I/O then, is when a process performs an I/O operation, it does the request only, continues working on other things, and returns to fetch the I/O result when the data is ready to fetch or write; it also means the process can work on other things while slow I/O operations are being performed.