

CSE3100 - Introduction to System Programming Assignment #6
---------------------------------------------------------------

In this assignment, you will complete an FTP-like client for transferring files. The client provides 4 commands: ls, get, put and exit. You will have to finish the implementation of the first three, and some supporting functions. If you open `command.h` with a text editor, you'll notice the following definitions and structures:

```
#define CC_LS 0
#define CC_GET 1
#define CC_PUT 2
#define CC_EXIT 3
#define PL_TXT 0
#define PL_FILE 1

typedef struct CommandTag {
    int code;
    char arg[256];
} Command;

typedef struct PayloadTag {
    int code;
    int length;
} Payload;
```

The client performs ls, get, put or exit by constructing a `Command`, and sending it to the server, and then sending or receiving a `Payload` struct, followed by text or the contents of a file. The `CC_LS`, `CC_PUT`, `CC_GET`, and `CC_EXIT` are used in the code field of the `Command` struct, to indicate what command is being performed. The `Payload` indicates whether it will be followed by text, or the contents of the file. The code field of the `Payload` is either `PL_TXT` or `PL_FILE`, and the length field indicates how many bytes are being sent.

Before I describe how each of these commands work, I need to point out the following fact: when you send an integer between the client and server, you need to convert it to network byte order, and when receiving an integer, you need to convert to host byte order. You should look at the manpage for the `ntohl` and `htonl` functions.

## 1 LS Command

This command tells the server to send a list of files in the current working directory. The protocol works as follows:

1. The client sends a `Command` to the server. The code field should be set to `CC_LS`, and the `arg` field should be an array filled with all zeros.
2. The server prepares a `Payload`, and sends it to the client.
3. The server prepares the list of files, and sends it to the client. Since this is just text, the code field of the `Payload` is set to `PL_TXT`, and the length field is set to the number of characters in the text.
4. The client prints the list of files

You need to finish the `doLSCommand` function in `client.c`

## 2 GET Command

This is when the client requests a file from the server. The communication works as follows:

1. Obtain the name of the file to retrieve over the connection from the end-user (scanf does it and the code is there!)
2. Client prepares a Command, and sends it to the server. The code field should be set to CC GET, and the arg field should be set to the name of the file, including the null terminator.
3. The server prepares a Payload and sends it to the client. The code field should be set to (network byte ordered) PL FILE, and length should be set to the size of the file in bytes (again, in network byte order).
4. The server calls the `sendFileOverSocket` function to send the file contents to the client.
5. The client receives the file, and saves it to disk, appending ".download" to the name of the file.

You need to finish the `doGETCommand` function. Of course, the client will need some way of receiving the file. You need to finish the `receiveFileOverSocket` function in `command.c`, which I will describe here.

### 2.1 receiveFileOverSocket

Here is the signature for this function:

```
void receiveFileOverSocket(int sid, char* fname, char* ext, int fSize);
```

Basically, you pass the file descriptor for the socket, the name of the file, the extension, and the size of the file (in bytes). The function should allocate enough memory to hold the file, and use the `recv` function to read the file contents from the socket, and write the file to disk. In the template code provided, I've given you enough code to open the file for writing on disk.

## 3 PUT Command

This function does the inverse of GET: it sends a file to the server. It works as follows:

1. Client constructs a Command, and sends it to the server. The code is set to the (network byte ordered) CC PUT, while the name of the file is placed in the arg field.
2. The client computes the size of the file using the `getFileSize` function (provided for you). It prepares a Payload, where the code is (network byte ordered) PL FILE, and the length field is the size of the file. The client sends the payload to the server.
3. The client calls the `sendFileOverSocket` function to send the file contents to the server. You need to implement the `sendFileOverSocket` function.

You need to finish the `doPUTCommand` function. Of course, the client will need some way of sending the file. You therefore need to finish `sendFileOverSocket` function in `command.c`, which I will describe below:

### 3.1 sendFileOverSocket

This function has the following signature:

```
void sendFileOverSocket(char* fName, int chatSocket);
```

This should open the file, and send its contents over the chatSocket. It is quite similar to the receiveFileOverSocket as you might have already guessed.

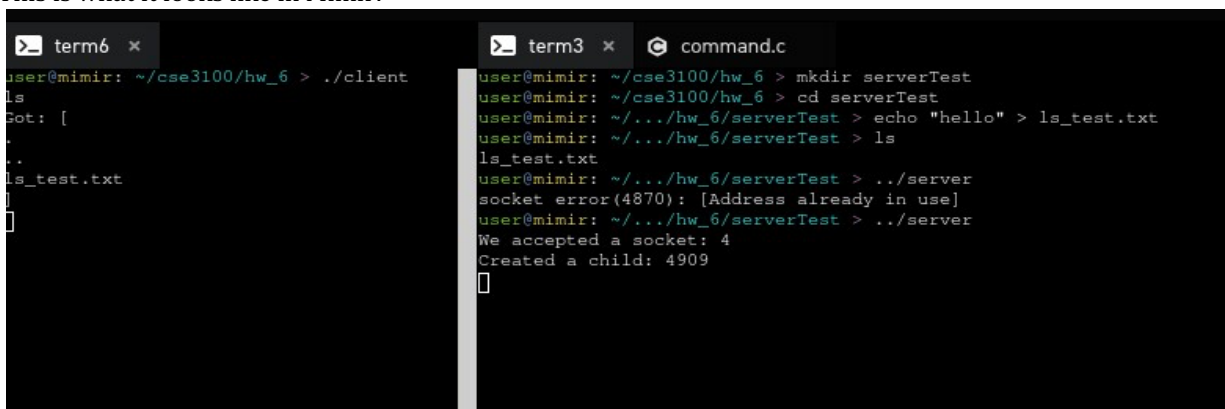
## 4 Compiling and Testing

You can use make to compile both the client and server. However, the compiler may complain about the readdir function being deprecated. Don't worry about this.

You can start by finishing the doLSCommand function. I recommend you test the client and server together as follows:

1. Open two side-by-side tabs in the Mimir IDE.
2. Create a directory called serverTest, and cd into it in one of the tabs.
3. Use echo to create a file with some text in it. I'll call it ls\_test.txt
4. Start the server in the tab that is in the serverTest directory.
5. Start the client in the other tab. Type ls and press enter to get a listing of files on the server side.

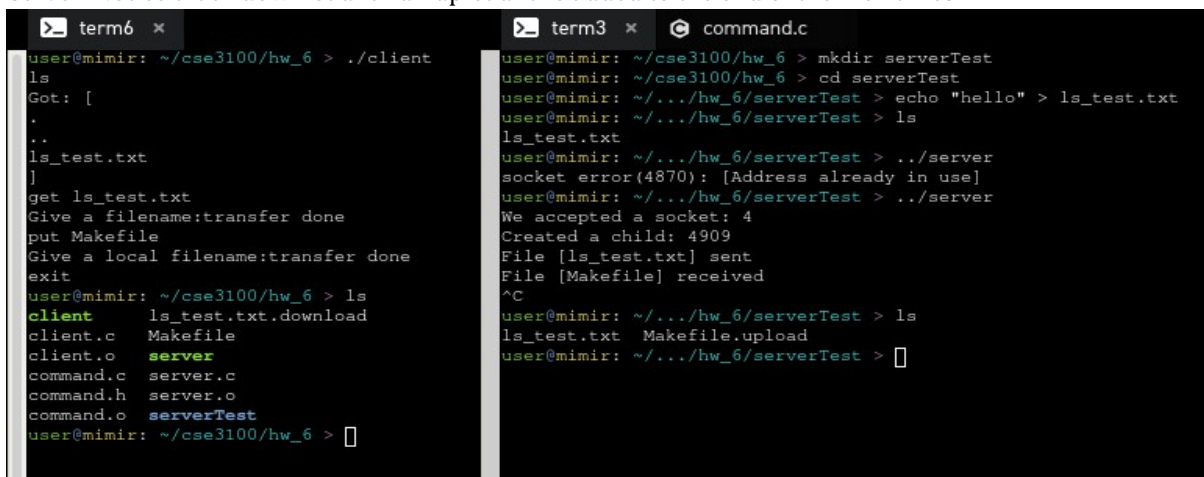
This is what it looks like in Mimir:



```
term6 x
user@mimir: ~/cse3100/hw_6 > ./client
ls
Got: [
..
ls_test.txt
]

term3 x  command.c
user@mimir: ~/cse3100/hw_6 > mkdir serverTest
user@mimir: ~/cse3100/hw_6 > cd serverTest
user@mimir: ~/.../hw_6/serverTest > echo "hello" > ls_test.txt
user@mimir: ~/.../hw_6/serverTest > ls
ls_test.txt
user@mimir: ~/.../hw_6/serverTest > ../server
socket error(4870): [Address already in use]
user@mimir: ~/.../hw_6/serverTest > ../server
We accepted a socket: 4
Created a child: 4909
```

Here is an example of using get to retrieve the ls\_test.txt file from the server, and put to send the Makefile to the server. Notice that ".download" and ".upload" are added to the end of the file names:



```
term6 x
user@mimir: ~/cse3100/hw_6 > ./client
ls
Got: [
..
ls_test.txt
]
get ls_test.txt
Give a filename:transfer done
put Makefile
Give a local filename:transfer done
exit
user@mimir: ~/cse3100/hw_6 > ls
client      ls_test.txt.download
client.c    Makefile
client.o    server
command.c   server.c
command.h   server.o
command.o   serverTest
user@mimir: ~/cse3100/hw_6 >

term3 x  command.c
user@mimir: ~/cse3100/hw_6 > mkdir serverTest
user@mimir: ~/cse3100/hw_6 > cd serverTest
user@mimir: ~/.../hw_6/serverTest > echo "hello" > ls_test.txt
user@mimir: ~/.../hw_6/serverTest > ls
ls_test.txt
user@mimir: ~/.../hw_6/serverTest > ../server
socket error(4870): [Address already in use]
user@mimir: ~/.../hw_6/serverTest > ../server
We accepted a socket: 4
Created a child: 4909
File [ls_test.txt] sent
File [Makefile] received
^C
user@mimir: ~/.../hw_6/serverTest > ls
ls_test.txt  Makefile.upload
user@mimir: ~/.../hw_6/serverTest >
```