Proj-1: Client-Server File Transfer with TCP/IP

Course: **CSCI-3550/8550, Communication Networks (Spring 2025)**

*Written by **J. Santos**, Department of Computer Science*
College of Information Science and Technology (IS&T)
University of Nebraska at Omaha

Created: Jan. 2021

Last Revised: Apr. 11, 2024

DUE: *See Canvas for Due Date!*

About

This assignment goes over your first *major* programming assignment, which asks that you write a client/server application to transfer a list of files from the client application to the server application. This project significantly builds upon the work you did for *Pre-project 4: Read/Write Files*. The focus is on writing **TCP-oriented client/server applications** from a Network Application programmer's perspective using the *BSD Sockets Interface* on a GNU/Linux system.

Useful References

- J. Santos, "*Client-Server Programming with BSD Sockets*". (*Should be accessible to you in Canvas*)

- J. Santos' Video Lecture on TCP/IP Programming using BSD Sockets. (*Should also be accessible to you in Canvas*)

Before moving on it is VERY important that you know that this document assumes you've fully read "*Client-Server Programming with BSD Sockets*", above.

> ***ALWAYS READ THE HANDOUT IN ITS ENTIRETY FIRST BEFORE CARRYING OUT THE ASSIGNMENT'S INSTRUCTIONS.***

Assignment Overview

The purpose of this assignment is to write a simple **TCP-oriented client/server application** from a Network Application programmer's perspective using the "sockets interface", otherwise known as *BSD Sockets Interface* that is commonly available in UNIX-like and GNU/Linux systems. For this programming assignment you will need to produce *two* separate program executables:

- The first serving as the **client** program.
- The second serving as the **server** program.

Let's discuss each in turn.

The *Client* Program

The **client** program will be fed a list of file names at the command line when it is invoked to start, specifying the files on disk with any arbitrary data (up to a maximum size of 10MiB). The client will then proceed through the list of files, and for each such file, it will:

1. Open one TCP connection with the server.
2. Send ONE file to the server to completion.
3. Close the connection.

Again, the above three steps are conducted for *each* file specified to the client program at the command line. So, if there are three files provided as command-line arguments, then a TCP connection will be established, used to transfer the file, and then torn down with the server three times. Once ALL files have been transferred, the client will clean up and swiftly terminate. The following shows what the expected behavior (and output) should be under normal circumstances:

```
$ ./client 127.0.0.1 5000 shopping.txt notes.txt assignments.txt
[Enter] client: Connecting to 127.0.0.1:5000... client: Success!
client: Sending: "shopping.txt"... client: Done. client: Connecting to
127.0.0.1:5000... client: Success! client: Sending: "notes.txt"...
client: Done. client: Connecting to 127.0.0.1:5000... client: Success!
client: Sending: "assignments.txt"... client: Done. client: File
transfer(s) complete. client: Goodbye!
```

Example behavior from the client program, including command-line arguments, which include the IP address and port number to connect with along with an arbitrary list of files to transfer to the server (three in this example). The corresponding output while transfers are taking place is also shown. Your version must replicate this behavior exactly as shown.

Note—as is evident from the above example—that the client program must (*as a minimum*) take two mandatory command-line arguments: (1) the first being the *IP address* of the server in dotted-decimal notation; and (2) the second being the *port number* of the server. Again, these are MANDATORY. Any arguments following these two REQUIRED arguments are assumed to be the list of files on disk to be transferred to the server. Thus, the client command-line specification is:

`client <server_IP> <server_Port> file1 file2 ...`

where:

- `<server_IP>` is the IP address in dotted-decimal notation of the server you want the client to connect with.
- `<server_Port>` is the port number in the server you want the client to connect with.
- `file1 file2 ...` list of space-delimited files to transfer in sequential order to the server.

Note also in the above output that all messages printed by the client application are prefixed with "`client:`". Make sure your program follows this convention as well because you will run *both* the client application and server application on the *same* terminal shell, and you'll want to be able to determine *which* of your programs (client or server) the messages are coming from!

Once the TCP connection is open, **you must ensure that ONLY THE BYTES corresponding to the current file being transmitted are *written* into the socket. That is, do NOT build a custom protocol into your program that adds additional bytes for some 'special meaning'**. The client must treat the TCP connection (and its socket) as if you were simply writing a file to disk via a file descriptor (*just as you did in **Pre-project 04***) and close the connection once the file's bytes have all be fully transmitted to the server. The server will automatically detect the 'end-of-file' condition when it attempts to read more bytes and its corresponding `recv()` function returns a value of 0.

The *Server* Program

The **server** program, in turn, and once started, will be 'listening' (awaiting) for TCP connection requests from the client. Obviously, the server would have to be started first. The server will read the file data from a socket and store it into a buffer (that *you* would pre-allocate). Once the client terminates the connection (by closing it once it has finished sending the file to the server) the server, in turn, will take all the buffered data and dump the ENTIRE contents received onto a new file on disk. Each time a new connection (from the client) is 'accepted', a *single* file will be received and written to disk. This means the server must always be running, checking and waiting for new TCP connection requests from any clients that want to transmit another file.

Now, each time a new file is saved to disk, it must be saved using sequential file numbers with a name of the form:

`file-XX.dat`,

where `XX` is a two-digit number that starts from "01" (e.g. `file-01.dat`). Each time the server is terminated and re-started, the file numbering will *still* begin from "01". This also means that if such a file already exists locally on the system, it will simply be overwritten completely with the new data.

Note that the server (unlike the *client*, which automatically self-terminates once ALL files have been transferred) will run forever, always waiting for new connection requests. The ONLY way to shut down the server will be by sending it a **SIGINT** signal (aka. [CTRL]+[C] at the keyboard), at which point the server will release its resources, close all file handles and sockets, memory, and shut down nice and clean.

When starting the server, it will only have ONE REQUIRED argument being the *port number* that it will use and assigned to its 'listening' socket for connection purposes.

The following example illustrates the general behavior of the server and its expected output under normal circumstances:

```
$ ./server 5000 [Enter] server: Awaiting TCP connections over port
5000... server: Connection accepted! server: Receiving file... server:
Connection closed. server: Saving file: "file-01.dat"... server: Done.
server: Awaiting TCP connections over port 5000... server: Connection
accepted! server: Receiving file... server: Connection closed. server:
Saving file: "file-02.dat"... server: Done. server: Awaiting TCP
connections over port 5000... server: Connection accepted! server:
Receiving file... server: Connection closed. server: Saving file:
"file-03.dat"... server: Done. server: Awaiting TCP connections over
port 5000... ^C ((User hits [CTRL]+[C] on the keyboard...)) server:
Server interrupted. Shutting down. server: Goodbye!
```

Example output behavior for the *server* program. It takes one parameter and that is the port number it will 'listen' for connection requests. The output shows three files being received and their assigned names once saved to disk. Upon hitting [CTRL]+[C] the program is forced to terminate, but with the help of a signal handler, we are able to 'clean up' nicely. Your program must produce the same output and behave in the same manner.

> **Note**: In the above example the lines "`server: Connection closed.`" and "`server: Saving file: "file-XX.dat"`" may be *swapped,* depending on how you attack the problem. If you use `recv()` with the `MSG_WAITALL` option then you know you have EVERYTHING, so you can *save* the file first and then close. But, if on the other hand, you keep using `recv()` without this option then you must wait until you detect the "EOF" condition (`recv()` returns 0) to know that the client has sent all the data and at that point you *could* close the socket first, and *then* save. Up to you.

In any case, the above example output shows how the server would behave if the client had sent us three files given in the previous example for the client. You can also gather the command-line specification for the server must be:

```
server <listening_port>
```

Where

- `<listening_port>` is the *unprivileged* port number to listen for connection requests.

Once again, **the server is hard-coded internally** to IP address `127.0.0.1`.

You can see above the ONLY required command-line argument for the server is the *listening port* number. Naturally, since ports 0 through 1023 (*first 1024 ports*) are 'special' (i.e. *privileged*) ports, we need to ensure that we allow only port numbers 1024 up to 65535. **Any port specification between 0 and 1023 should cause the server to issue an error to the user and immediately terminate**. Specifics are given in the *Instructions* section further below.

This also means that **the server is hard-coded internally** to IP address `127.0.0.1`.

Note also—once again—that the server messages are also prefixed with the string "`server:`". Again, this is necessary so that we can determine which program is producing the messages that we see since you will run both, the client *and* server programs on the same terminal shell. It will be VERY difficult to do so otherwise!

That's it. This basically explains—at 30,000 feet—what you are expected to do for this assignment.

Let's discuss specifics.

Instructions

I. Write a Client/Server TCP Program(s)

As described in the *Assignment Overview* section, for this assignment you will produce two programs:

1. A *client* program, whose source code you will save in `client.c`.
2. A *server* program, whose source code you will save in `server.c`.

Requirements & Specifications for the Client

The command-line specification for the client program *must* be as follows:

```
client <server_IP> <server_Port> file1 file2 ...
```

Where (and once again):

- `<server_IP>` is the IP address in **dotted-decimal notation** of the server you want the client to connect with.
- `<server_Port>` is the (*non-privileged*) port number in the server you want the client to connect with.
- `file1 file2 ...` list of space-delimited files to transfer in sequential order to the server.

EXCLUDING the program name itself, if two arguments or less (i.e. *no files*) are given, your program must issue the following message as illustrated in the following example:

```
$ ./client 127.0.0.1 5000 [Enter] client: USAGE: client <server_IP>
<server_Port> file1 file2 ...
```

Example of what happens on the client when the user does not specify any files.

In the case where a file does NOT exist the expected behavior is that your program should simply continue to the next file in the list. For example:

```
$ ./client 127.0.0.1 5000 shopping.txt BooBooTheCat.txt assignments.txt
[Enter] client: Connecting to 127.0.0.1:5000... client: Success!
client: Sending: "shopping.txt"... client: Done. client: Connecting to
127.0.0.1:5000... client: Success! client: Sending:
"BooBooTheCat.txt"... client: ERROR: Failed to open: "BooBooTheCat.txt"
client: Connecting to 127.0.0.1:5000... client: Success! client:
Sending: "assignments.txt"... client: Done. client: File transfer(s)
complete. client: Goodbye!
```

Example of client behavior when it encounters a non-existent file. The precise order of the messages that appear when an ERROR is encountered while attempting to transfer a file does NOT have to be EXACTLY as shown above, as it all depends when you choose to issue messages. In the example above the program attempts to make a connection first before determining whether it can even open the file, or whether it exists, which explains why some of the messages appear above even though we eventually encounter an ERROR.

> **Note**: How you handle the above particular case on the SERVER side is up to you. If you open the connection (as shown above), then the server WILL create an empty `*.dat` file still. That's fine. However, if *your* version of the CLIENT first checks whether the file exists BEFORE it opens a connection with the server, you could opt to NOT open a connection (since the file doesn't exist) preventing the server from creating a 0-byte file for a non-existent file in the first case. **Either case is fine**. Just be consistent.

Your client program must also be ready to provide or output the following error messages where appropriate in your program:

- `"client: ERROR: Failed to allocate memory."`, when `malloc()` fails.
- `"client: ERROR: Failed to create socket."`, when `socket()` fails.
- `"client: ERROR: Failed to open: <filename>"`, when `open()` fails to open `<filename>` for any reason.
- `"client: ERROR: Unable to read: <filename>"`, when `read()` fails to read from the file via its descriptor for any reason.
- `"client: ERROR: connecting to <server_IP>:<server_Port>"`, when `connect()` fails to connect to the given server specified by the given IP and port number at the command line by the user.

- `"client: ERROR: Port number is privileged."`, when the port number is between 0 and 1023 (inclusive).
- `"client: ERROR: While sending data."`, when `send()` gives an error.

ANY condition that triggers the above errors should cause the client to terminate (and *clean up after itself*).
(*The only exception is when you fail to open a file. As was demonstrated earlier, the client should simply proceed to the next file*)

Your client must also present the ordinary messages shown in examples given in the *Assignment Overview* section under normal operating conditions.

Furthermore, the **SIGINT** signal handler for the client *must* contain the following code ONLY, and EXACTLY as given. NO OTHER CODE should be placed inside of the handler:

```
/* SIGINT handler for the client */ void SIGINT_handler( int sig ) { /*
Issue an error */ fprintf( stderr, "client: Client interrupted.
Shutting down.\n" ); /* Cleanup after yourself */ cleanup(); /* Exit
for 'reals' */ exit( EXIT_FAILURE ); } /* end SIGINT_handler() */
```

The philosophy behind signal handlers and the `cleanup()` function has been explored in previous pre-projects in this course, so these will not be repeated here.

Requirements & Specifications for the Server

The command-line specification for the *server* program must be as follows:

`server <listening_port>`

Where:

- `<listening_port>` is the *unprivileged* port number to listen for connection requests from clients.

EXCLUDING the program name itself, if NO arguments (i.e. the *port number*) is given, then the server should present a 'usage' message as shown below:

```
$ ./server [Enter] server: USAGE: server <listen_Port>
```

Example of server behavior when the user omits the *port* argument at the command line which is required.

Your server program must also output the following error messages at the appropriate locations in your code:

- `"server: ERROR: Failed to allocate memory."`, when `malloc()` fails.
- `"server: ERROR: Failed to create socket."`, when `socket()` fails.
- `"server: ERROR: Unable to create: <filename>."`, when `open()` fails to create `<filename>` (e.g. `file-01.dat`).
- `"server: ERROR: Unable to write: <filename>."`, when `write()` fails to write any data to `<filename>` (e.g. `file-01.dat`).
- `"server: ERROR: Failed to bind socket."`, when `bind()` fails for any reason.
- `"server: ERROR: listen(): Failed."`, when `listen()` fails.
- `"server: ERROR: While attempting to accept a connection."`, when `accept()` fails.
- `"server: ERROR: Port number is privileged."`, when the user specifies a *privileged* port number at the command line.
  (*Recall a privileged port is in the range between 0 to 1023, inclusive*)
- `"server: ERROR: Reading from socket."`, when `recv()` gives an error.
  (*Recall a return value of 0 is NOT considered an error. That tells you the 'other side' has closed the connection.*)

Note that it may not be possible for you to test your program to see if we can 'hit' all the errors. All you are responsible is in making sure you *are* accounting for these events in your code.

Your server must also present all the informational messages in the examples given in the *Assignment Overview* section under normal operating conditions.

Furthermore, the **SIGINT** handler for the server must also contain the following code (without ANY modifications whatsoever):

```
/* SIGINT handler for the client */ void SIGINT_handler( int sig ) { /*
Issue an error */ fprintf( stderr, "server: Server interrupted.
Shutting down.\n" ); /* Cleanup after yourself */ cleanup(); /* Exit
for 'reals' */ exit( EXIT_FAILURE ); } /* end SIGINT_handler() */
```

Once again, the philosophy behind signal handlers and the `cleanup()` function has been explored in previous pre-projects in this course, so these will not be repeated here.

II. Compiling

Your `client.c` and `server.c` should be in the same folder. You *must* compile your code using the following EXACTLY:

```
gcc -Wall -Wstrict-prototypes -Wmissing-prototypes -ansi -pedantic-
errors -D_DEFAULT_SOURCE -o client client.c
```

and...

```
gcc -Wall -Wstrict-prototypes -Wmissing-prototypes -ansi -pedantic-
errors -D_DEFAULT_SOURCE -o server server.c
```

> **Warning**: Your programs MUST compile SPECIFICALLY with the above options. If I cannot compile your code without ERRORS your program will NOT be graded, even if it magically generates an executable still.

> **Warning**: Do NOT ignore compiler warnings. These will be HEAVILY penalized.

> **Warning**: You *must* include with your submission the **Odin** executable(s) to receive credit. If you do not include the Odin executable(s) you will not receive credit.

III. Test your Program and Make Sure it Works!

You MUST make sure your program works. Your strategy should be to tackle the development of the *server* first. This way, you can use the TELNET command-line application to test your server. For example, if your *server* is already running and listening on over port 5000, you can start a telnet session as follows:

```
$ telnet 127.0.0.1 5000 [Enter] Trying 127.0.0.1... Connected to
127.0.0.1. Escape Character is '^]' ((This means [CTRL] + ] )) ((Type
something)) Hello Dolly! [Enter] ((Type [CTRL] + ] to bring the telnet
prompt)) ^] telnet> close [Enter] Connection Closed. telnet> quit
[Enter] $
```

Example of using the *telnet* command-line application to test your server.

Then, if you're confident that the server is ready to go, you can start working on the client, and—having the server completed—you'll have a way to test the client itself. If you start the other way around you'll just quickly get stuck with no way to test your client. You can't use telnet to test the client.

> **Note**: The corresponding discussion video supplement created by the instructor should also walk you through this process as well as how to work both, the client and the server from a *single* command-line window shell.

> **Note: While all examples in this handout suggest you use port 5000, please pick a RANDOM port number and STICK to it**. Remember other students will be testing *their* programs in the same system as well and if a port is being used up, they won't be accessible to your program. Again, **pick a RANDOM port number**, and STICK with it.

Other Ways to Test: Instructor-provided Odin Executables

There are also sample *Odin* executables that should be available to you in Canvas that you can upload to your Odin account. These are called:

- `client_jsantos-odin-XXXXXXXX`, which you can use to test *your* version of your server.
- `server_jsantos-odin-XXXXXXXX`, which you can use to test *your* version of your client.

You can use the client executable to test your server implementation, and vice versa. The `XXXXXXXX` usually refers to a date when the executable was last generated.

These programs are written so that the arbitrary files to transfer read and transferred 'as is' without adding any additional custom protocol or special 'messages' to the transmission. This is an important bit of detail because if you add additional special 'messaging protocols' (*which you're not supposed to do*), you can't expect the above executables to play nice with *your* versions of the same.

In any case, these are there for you to utilize for testing purposes only.

> Keep in mind the output messages may slightly deviate from what is asked of you in this handout, but their behavior should be relatively faithful to what is requested here.

For example, say you want to test *your* version of the client program. The you would want to use `server_jsantos-odin-XXXXXXXX` to test your code by starting the server and then placing in in the 'background' so that you can still manipulate the shell and start *your* version of the client:

```
$ ./server_jsantos-odin-XXXXXXXX 5000 [Enter] server: Awaiting TCP
connections over port 5000... ^Z ((<-- What[CTRL]+[Z] looks like when
you press it)) bg [Enter] ((Keep the server running in the background))
Send job 1 "./server 5000" to background $ ./client 127.0.0.1 5000
sometestfile.txt client: Connecting to 127.0.0.1:5000... server:
Connection accepted! client: Success! client: Sending:
"sometestfile.txt"... server: Receiving file... client: Done. client:
File transfer(s) complete. server: Saving file: "file-01.dat"...
client: Goodbye! server: Done. fg [Enter] ((Bring server back to
foreground)) Send job 1, "./server 5000" to foreground ^C ((<-- What
[CTRL]+[C] looks like when you press it)) server: Server interrupted.
Shutting down. $ ((We get the prompt back, so we're now in control))
```

A hypothetical example of how to use the instructor's provided *server* executable to test a student's custom *client* using the same terminal shell, which requires that we push the server to the background so that we may gain control of the shell and start the client software, and then bring the background server BACK to the foreground so that we can kill it (via [CTRL]+[C]) once we're done testing to terminate the server as well. Note how the prompt `$` appears and disappears, indicating points at which we have control of the shell—thereby allowing us to enter commands—versus when we do not have control because the server and/or client do.

Note this interaction interleaves *server* messages with *client* messages.

> Keep in mind that *your* corresponding experience doing the same might result in messages appearing in different order. *That is okay.*

To test *your* version of the server, you would pretty much repeat the above, except now you would invoke your *own* server, and then use the instructor's *client* program instead.

> **Note**: Feel free to employ both techniques to test your work. The instructor will test your server with his client program and your client with his server program and vice versa, and then test both your client and server at the same time.

> **Note**: Please do NOT submit non-working programs.

IV. Packaging for Submissions

Once you've compiled your executable and you are ready to submit your project to Canvas, you'll need to package the source code for your client and server, as well as include the compiled executable images from *Odin*: that is, your *tarball* should only contain `client.c`, `server.c`, `client` (Odin-executable) and `server` (Odin-executable) and NOTHING ELSE.

> **Warning**: You *must* include with your submission the **Odin** executable(s) to receive credit. If you do not include the Odin executable(s) you will not receive credit.

Let's suppose (*as an example*) your current working directory is `~/Documents/csci3550/Project-01/`, and your files are *inside* the `Project-01/` folder. *That's* the folder you'll want to create a 'tarball' out of. So we would:

1. Back up ONE directory so that you are OUTSIDE of this folder and NOT inside.
2. Invoke the `tar` command to create the `*.tar` archive.
3. Invoke the `gzip` command to compress the archive and obtain `*.tar.gz` file.

These steps are illustrated below:

```
$ pwd [Enter] ~/Documents/csci3550/Project-01/ $ cd .. [Enter] $ pwd
[Enter] ~/Documents/csci3550/ $ tar -cvf JDoe_CSCI3550_Project01.tar
Project-01/ [Enter] $ ls -l JDoe_CSCI3550_Project01.tar $ gzip
JDoe_CSCI3550_Project01.tar [Enter] $ ls -l
JDoe_CSCI3550_Project01.tar.gz
```

Hypothetical example of how to create a 'tarball' out of the contents of a folder. You need to know where you are (that's what the `pwd` command is for); you need to know *what* is contained where you are (that's what the `ls -l` is for); and you need to know how to move where you want to go (that's what the `cd` is for).

Once you have your tarball (`*.tar.gz` file), you MUST verify that it actually has *something* in it! You can do a quick by taking a peek at what's inside your tarball in the following way:

```
$ gzip -dc JDoe_CSCI3550_Project01.tar.gz | tar -tvf - [Enter] Project-
01/client.c Project-01/client Project-01/server.c Project-01/server
```

Example on how to verify that the tarball *actually* contains something in it!

Once you've confirmed that your tarball is good to go, you need to retrieve it from Odin and submit it to Canvas.

> **Please make sure your tarball doesn't accidentally include 10MB files you use for testing purposes! These should NOT be included in the tarball submission or the submission file itself will be huge!**

> **Warning**: Please do NOT leave 10MB test files in your tarballs. You will be HEAVILY penalized for doing so! 😡

Other Details

This project builds on some of the foundational concepts you've learned over several *pre-project* assignments in this course. Most notable is all of the things you've learned in ***Pre-project 04***. Recall that in that assignment you learned several concepts that you will need to apply here as well. Some of these tasks include:

- How to read and parse *command-line arguments*.
- How to direct error messages to the *standard error* by calling `fprintf( stderr, ... )` instead of just `printf()`. It should be standard practice that when you issue error messages that they be directed to this particular *stream* instead of the *standard output* stream (what's what `printf()` does).
- How to allocate memory using `malloc()` whenever you need to do so, and how to [eventually] deallocate it using `free()`.

- How to open files for reading and/or also create new files using the `open()` system call.
- How to read from files, and write to files, using the `read()` and `write()` system calls, respectively.
- How to close files using `close()` as well as sockets when these are no longer needed so that their corresponding operating-system resources are released and reclaimed by it.
- How to install *signal handlers* to catch [CTRL]+[C] at the keyboard.
- How to write and compartmentalize 'clean-up' code in a *single* function called `cleanup()`, whose purpose is to release any resources you may have allocated (memory, file descriptors, and/or socket descriptors) whenever you need to 'close shop' for ANY reason. Furthermore, 'resources' should be carefully released and/or closed only one time. For example, you should not call `free()` to release memory you've already released, nor call `close()` on a file or socket you've already closed. The way to ensure this doesn't happen is to make your pointers `NULL` and your file descriptors `-1` so that you can check whether these things have already been released and/or closed before you 'blindly' attempt to do so in multiple parts of your program. The below example illustrates the *proper* way to do such things:

```
/* Free ONLY if it's safe to do so... */ if( buf != NULL ) { free( buf ); /* Release the memory */ buf = NULL; /* Mark the pointer so that we know we have already released it and avoid a 'dangling pointer. */ } /* Free a file or socket descriptor */ if ( fd > -1 ) { close( fd ); /* Close the file or socket. */ fd = -1; /* Mark it as such. */ }
```

  Note that the idea above is that if you place any such code inside of your *one* `cleanup()` function, then calling `cleanup()` multiple times for *whatever* reason will be 'safe' to do because the code inside of your `cleanup()` takes precautions to only release resources that have NOT YET been released and will not attempt to do so if it has already done so.

Other Restrictions and Requirements

This programming assignment MUST be strictly written using ANSI/ISO C.

- You must NOT use any C++ or any of its features.
- You cannot use ANY support libraries of ANY kind outside what is already provided by the default GCC compiler itself and its Standard C library.
- You must use the *system calls* `open()`, `read()`, `write()` and `close()` for file access and not the *similar* functions provided by the Standard C library.

Final Words

This programming assignment is a **high-priority assignment**.

- It is imperative that you start tackling the assignment at your earliest convenience.