CS 454/654 Assignments 2 and 3

Deadlines	Assignment 2	Assignment 3
Due	20 June 2018 (4:00pm)	11 July 2018 (4:00pm)
Returned	25 June 2018	26 July 2018
Appeal Deadline	One week from return date	One week from return date

See course webpage for late submission policy
See course webpage for TA contact/office hours
It is imperative that you start these Assignments early!

Section 1) High Level Overview

In these Assignments, you will implement the core components of WatDFS, a simplified distributed file system. WatDFS will act as a transparent layer on top of the local file system to support creating, opening, reading, writing, and closing files on remote machines. You will first implement WatDFS using a remote access model for Assignment 2 (Sections 4-9). You will implement WatDFS using a download/upload model with client-side caching for Assignment 3 (Sections 12-16).

To implement the WatDFS file system, you will integrate with FUSE, a popular library that enables the development of custom file systems. You will be given a library that sets up the FUSE functionality. You will also be given an RPC library that will provide the functionality to make procedure calls on remote servers.

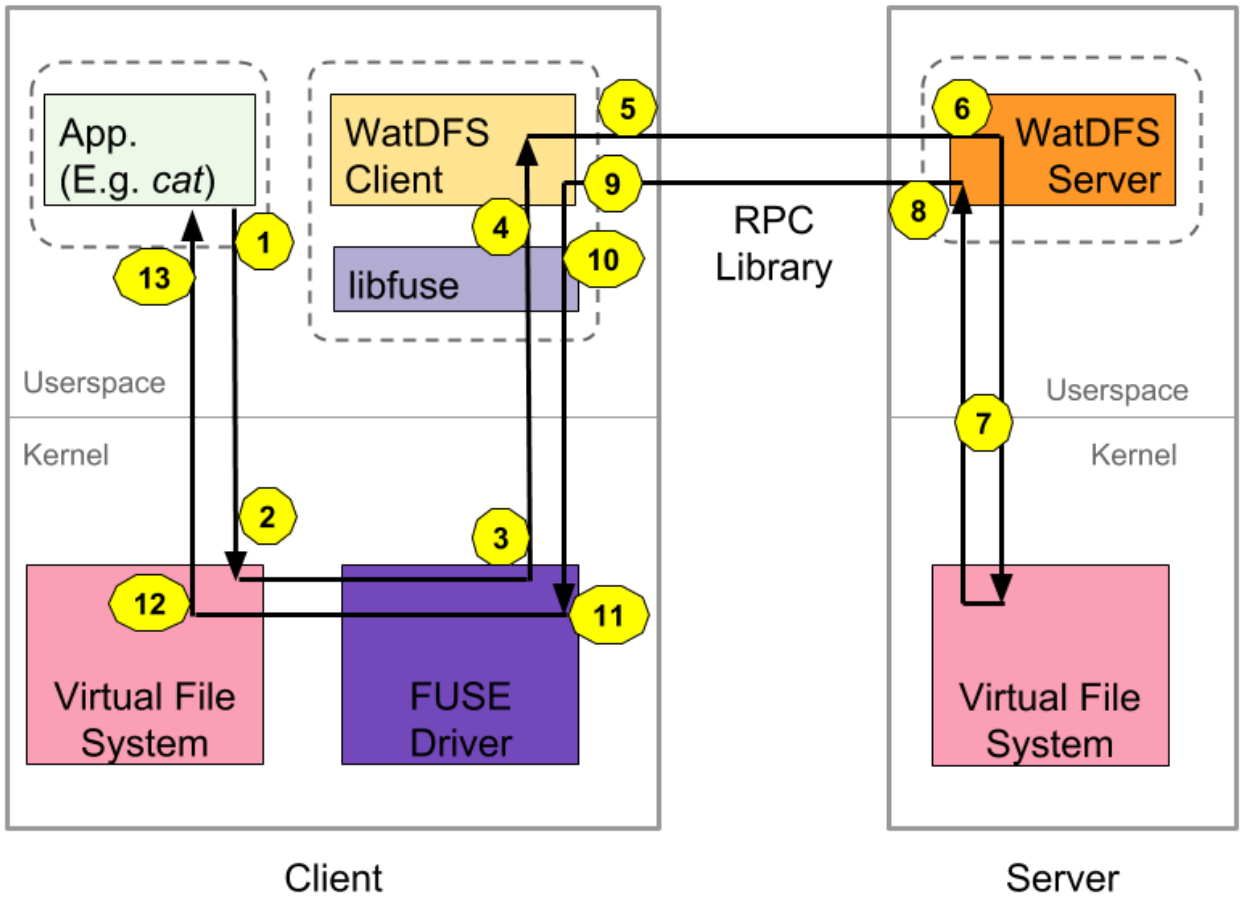
This document presents an overview of FUSE and describes how you should integrate your solution with FUSE (Sections 2-3). It also describes the API of the provided RPC library (Section 10), tips for doing the Assignments, and pointers to more resources (Section 11). ***Ensure that you read the entire document before beginning the Assignments as it contains answers to many frequently asked questions and tips to avoid common pitfalls during implementation.***

Section 2) Distributed File System Overview

Distributed file systems like Coda and NFS allow a user to interact with files that may be located on a remote server. To do so, the distributed file system must present an interface that acts as though remote files are located on the local file system, and therefore accessible through standard system calls like `open`, `close`, `read`, and `write`.

FUSE presents the interface of a standard file system, and therefore maintains access transparency. Additionally, FUSE requires no kernel code modification. FUSE consists of two parts: the FUSE kernel driver, and the library *libfuse* that integrates with the virtual file system and passes calls on to a users pace library, such as the distributed file system that you will implement.

Below is a diagram that traces the steps, with numbers using arrows, that a file system operation invoked by an application will take to reach WatDFS, as well as the actions WatDFS should take. The dashed lines indicate separate processes, the thick grey lines separate machine boundaries, and the thin grey lines separate user space from the kernel. You will be responsible for implementing the WatDFS Client and WatDFS Server components.



Steps

1) An application (e.g., `cat`) performs a file system operation by making a system call (such as `open(), close(), read(), or write()`). This system call is passed to the Linux kernel.
2) The kernel redirects the system call to the virtual file system (VFS).
3) The VFS transforms the call into an appropriate call against the file system the application wishes to access. The VFS detects the file system call is to be served by

WatDFS, a FUSE-based file system. Consequently, the call is passed to the FUSE kernel driver.

4) The FUSE kernel driver delivers the call to the WatDFS client library (*libfuse*/WatDFS client implementation).
5) The WatDFS client library makes an RPC call to the WatDFS server, marshalling and sending appropriate data passed to it by the FUSE kernel driver.
6) The WatDFS server receives the RPC call, unmarshalls the data, and passes it to the WatDFS server implementation.
7) The WatDFS server makes the corresponding system call against the local file system and receives a response from the file system.
8) The WatDFS server responds to the RPC call, sending back any necessary data and a return code from the system call.
9) The RPC response is passed back to the WatDFS client library and the client library generates a response to the FUSE driver's request.
10) The WatDFS client library forwards the FUSE response to the FUSE driver.
11) The FUSE driver passes the response back to the VFS.
12) The VFS translates the file system response back into a system call response for the kernel.
13) The application receives the result of the system call.

To complete Assignments 2 and 3, you must implement steps 5-10 for various functionalities, with the assistance of the provided libraries. To test your system, you should create test applications that perform steps 1 and 13. **We will test the individual components you provide (WatDFS client library and server) in addition to complete end-to-end tests. Most of this testing will be done through automated tests. Consequently, it is imperative that you implement the Assignments *exactly* as this document specifies.**

In these Assignments, there will be only one client application (e.g. `cat`) interacting with a WatDFS client at a time.[1] However, there may be multiple WatDFS client instances that are connected to the same WatDFS server.

Integrating with FUSE requires mapping the FUSE kernel driver's calls to functions in the client WatDFS library. We have implemented this mapping, while filtering out file system calls that are not relevant to the Assignments (e.g. directory operations). Your task is to implement the WatDFS library functions needed to support key FUSE operations, which are detailed below.

Section 3) FUSE Operations to Support

To support opening, reading, writing and closing files, you will be required to support several FUSE functions that are grouped below into categories of related operations. You will need to consult the Linux `man pages` to find the corresponding system call that must be made to

[1] This is a simplifying restriction, because FUSE supports multiple client applications interacting with the same user space *libfuse* library.

support each *libfuse* function. Some pointers on helpful system calls are provided in Section 11. We also provide links to the related *libfuse* calls that provide more details on the specifications of the functions.

Section 3.1) General Notes

In general, the calls made by FUSE should return `0` on success, or `-errno` if an error occurs.[2] If this is not the case then it will be made clear what should be returned.

The first argument to every function (`void *userdata`) is a global state pointer that can be used to store state. The `userdata` pointer is allocated by `watdfs_cli_init` (detailed below), and should allocate and initialize any global structures that you may need.

libfuse always provides `path` names of the file it wishes to operate on, but using names rather than file descriptors results in more expensive lookups and metadata tracking. Therefore, some functions take a `fuse_file_info` structure. This structure has an integer member variable `fh`, which **should** be filled-in during `open`, and used afterwards to identify the file (i.e. act as a file descriptor).[3]

```
struct fuse_file_info {
     int flags; // open flags, available in open and release.
     /* other fields */
     uint64_t fh; // file handle, may be filled in by open
};
```

The `path` provided in calls is relative to the directory where FUSE is mounted.

The exact arguments to the functions have also been altered so as to add additional information to the call (e.g. `userdata`). If you are uncertain as to the types used in the functions (e.g. `mode_t`) additional information can be found by searching the *libfuse* headers or Linux headers.

Section 3.2) Initialization and Destruction

These functions set up the *libfuse* integration and are called on shutdown, respectively.

watdfs_cli_init

```
void *watdfs_cli_init(struct fuse_conn_info *conn,
```

[2] The return values of calls to libfuse are different from system calls. System calls typically return 0 on success and -1 on error. If an error occurs, then system calls set `errno` to a positive number to indicate what went wrong, and `errno` indicates the error code. (https://github.com/libfuse/libfuse/blob/cece24786e7d0a1cd2a2dc7b84ff54225d2b616f/include/fuse.h#L280-L283, http://man7.org/linux/man-pages/man3/errno.3.html)

[3] Although you could implement these Assignments using only file names, and no `fuse_file_info`, you must use the `fuse_file_info` for performance and compatibility with the automated tests.

```
                        const char *path_to_cache,
                        time_t cache_interval);
```

This function is called during the initialization of the file system. The pointer that is returned will be passed to every other function call as `userdata`. This function can be used to perform a one-time set up and initialization.[4]

watdfs_cli_destroy

```
void *watdfs_cli_destroy(void *userdata);
```

This function is called when the file system is being destroyed, and should clean up any allocated structures during initialization.

Section 3.3) File Attributes

Several file system operations require metadata information about a file (e.g. file size, number of blocks) as defined by the `stat` structure. The functions below fill in the the `stat` structure by getting file attributes given in `path`.
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L311

Important Note: *libfuse* checks the output `stat` structure (`statbuf`) to determine information about the existence of files. If the file does not exist, it is important not to fill in the `statbuf` structure.

watdfs_cli_getattr

```
int watdfs_cli_getattr(void *userdata, const char *path,
                       struct stat *statbuf);
```

watdfs_cli_fgetattr

```
int watdfs_cli_fgetattr(void *userdata, const char *path,
                        struct stat  *statbuf,
                        struct fuse_file_info *fi);
```

The `watdfs_cli_fgetattr` function differs from `watdfs_cli_getattr` by passing in the `fuse_file_info` structure, so you can use the `fi->fh` variable when making system calls.

Section 3.4) Opening and Closing

watdfs_cli_mknod

```
int watdfs_cli_mknod(void *userdata, const char *path,
```

[4] `watdfs_cli_init` is **not** responsible for starting the file system. Our provided client code will initialize and mount the FUSE file system, and then `watdfs_cli_init` will be called.

```
                        mode_t mode, dev_t dev);
```

This function is called to create a file if it does not exist.
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L329
Important Note: If an application calls `open` and the file does not exist, `watdfs_cli_mknod` will be called and then `watdfs_cli_open` will be called.

watdfs_cli_open

```
int watdfs_cli_open(void *userdata, const char *path,
                    struct fuse_file_info *fi);
```

This function is called to open a file. As mentioned above, `fi->fh` **must be** filled in with a file handle to identify the file that has been opened. **`watdfs_cli_open` should return `0` or if an error occurs `-errno`. This return value is different from the `open` system call that returns a file descriptor or `-1`.**
We will only use the following flags during `open`: `O_CREAT`, `O_APPEND`, `O_EXCL`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR`.
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L437

Important Note: The `fuse_file_info` structure contains useful information such as the flags that the file is requested to be opened with.
Hint: You can tell whether a file is requested to be opened read only, write only, or read/write by performing a bitwise and (`&`) of the flags and `O_ACCMODE` (https://www.gnu.org/software/libc/manual/html_node/Access-Modes.html)

watdfs_cli_release

```
int watdfs_cli_release(void *userdata, const char *path,
                       struct fuse_file_info *fi);
```

This function is called when FUSE is completely done with a file, generally within the `close` system call. **However, `watdfs_cli_release` is performed asynchronously**; that is, the `close` system call may complete before `watdfs_cli_release` completes. There is exactly one `watdfs_cli_release` per `open`, with the same file name, flags and file handle (`fi->fh`).
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L504

Section 3.5) Reading and Writing

The following calls correspond to systems calls that support reading and writing data.

watdfs_cli_read

```
int watdfs_cli_read(void *userdata, const char *path, char *buf,
```

```
                    size_t size, off_t offset,
                    struct fuse_file_info *fi);
```

This function reads into `buf` at most `size` bytes **from `offset`** of the file. It should return the number of bytes requested to be read, except on EOF (return the number of bytes actually read), or error (return `-errno`).
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L448
Hint: The amount of data requested to be read may be larger then the maximum amount of data that can be used as an array argument in our RPC library (Section 10.1). You need to handle this case.

watdfs_cli_write

```
int watdfs_cli_write(void *userdata, const char *path,
                     const char *buf, size_t size, off_t offset,
                     struct fuse_file_info *fi);
```

This function writes into `buf` `size` bytes **at `offset`** of the file. It should return the number of bytes requested to be written, except on error (`-errno`).
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L460
Hint: The amount of data requested to be written, may be larger than the maximum amount of data that can be used as an array argument in our RPC library (Section 10.1). You need to handle this case.

watdfs_cli_truncate

```
int watdfs_cli_truncate(void *userdata, const char *path,
                        off_t newsize);
```

This function changes the size of the file to `newsize`. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part is filled in with null bytes ('\0').
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L387
Important Note: `truncate` can be called without opening the file, but should succeed if the file has been created with write permissions.

watdfs_cli_fsync

```
int watdfs_cli_fsync(void *userdata, const char *path,
                     struct fuse_file_info *fi);
```

This function should flush the file data specified by the `path` and `fi`.
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L511

Section 3.6) Changing Metadata

watdfs_cli_utimens

```
int watdfs_cli_utimens(void *userdata, const char *path,
                       const struct timespec ts[2]);
```

This will change the file access and modification time with nanosecond resolution.
https://github.com/libfuse/libfuse/blob/da29b950bc9d204a93706985255a299f1dfea561/include/fuse.h#L650

Important note: There are different semantics based on the `ts` argument; you should read the documentation of `utimensat` http://man7.org/linux/man-pages/man2/utimensat.2.html

Assignment 2

Section 4) Remote Access Model (Assignment 2)

In the remote access model, every request is forwarded from the client to the server. A remote file is never stored at the client.[5]

To implement the remote access model in WatDFS, you will transform each of the described FUSE functions (Section 3) into an RPC call from the client to the server. The server will transform the RPC call into a system call and return the result to the client. The server should perform all operations on the files using appropriate system calls and should not buffer writes or cache files in memory.[6]That is, if you receive a write call, the write should be performed on the remote file server and should not be performed by keeping a copy of the file in local memory and writing to it.

In Assignment 2, the client and server should be stateless as all the information needed to perform the required operation is provided by the calling function.

The remote access model will serve as a warm up to familiarize you with FUSE, RPC calls, and the required system calls.

Section 4.1) RPC Protocol

To aid in your testing and development, the RPC calls that you should implement to satisfy the client requests are listed below with their argument types. **Each RPC requires an integer (`retcode`) as an output argument, which is the return code of the function call on the WatDFS server, which you must set.**

You must exactly follow this protocol specification, as the automated tests will verify it.

[5] The file will also be uncached at the client because we will be disabling client-side kernel caches. (`-o direct_io`).
[6] You do not need to worry about kernel buffering/caching at the server as part of this restriction. Your server code, however, should not buffer or cache files in memory.

Important note: rather than serializing each argument in a structure `T (e.g. fuse_file_info)`, you should simply serialize the entire structure as a character array of `sizeof(T)`.

Function Name
`getattr`
Function types

Arg label	is_input	is_output	is_array	type
`path`	yes	no	yes	ARG_CHAR
`statbuf`	no	yes	yes	ARG_CHAR
`retcode`	no	yes	no	ARG_INT

Function Name
`fgetattr`
Function types

Arg label	is_input	is_output	is_array	type
`path`	yes	no	yes	ARG_CHAR
`statbuf`	no	yes	yes	ARG_CHAR
`fi`	yes	no	yes	ARG_CHAR
`retcode`	no	yes	no	ARG_INT

Function Name
`mknod`
Function types

Arg label	is_input	is_output	is_array	type
`path`	yes	no	yes	ARG_CHAR
`mode`	yes	no	no	ARG_INT
`dev`	yes	no	no	ARG_LONG
`retcode`	no	yes	no	ARG_INT

Function Name
`open`
Function types

Arg label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
fi	yes	yes	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Important Note: The `retcode` for the `open` RPC call should be `0`, or `-errno`. This return value matches the expected return value of `watdfs_cli_open`, but is different from the system call `open`, which returns a file descriptor.

Function Name
`release`
Function types

Arg label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Function Name
`read`
Function types

Arg label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
buf	no	yes	yes	ARG_CHAR
size	yes	no	no	ARG_LONG
offset	yes	no	no	ARG_LONG
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Function Name
`write`
Function types

Arg label	is_input	is_output	is_array	type

path	yes	no	yes	ARG_CHAR
buf	yes	no	yes	ARG_CHAR
size	yes	no	no	ARG_LONG
offset	yes	no	no	ARG_LONG
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Function Name
truncate
Function types

Arg label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
newsize	yes	no	no	ARG_LONG
retcode	no	yes	no	ARG_INT

Function Name
fsync
Function types

Arg label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Function Name
utimens
Function types

Arg label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
ts	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Section 4.2) Notes

Recall from Section 3.2 the definition of `watdfs_cli_init`:

```
void *watdfs_cli_init(struct fuse_conn_info *conn,
                      const char *path_to_cache,
                       time_t cache_interval);
```

In Assignment 2, `path_to_cache` and `cache_interval` are not used, and should be ignored.

Section 5) Tips

You should implement functions in the following order: `getattr, mknod, open, release, fgetattr.` These will allow you to support opening and creating a file. Next you should implement: `write, read, truncate`. These will allow you to read and write data to a file. Finally implement: `utimens, fsync.`

To assist you in seeing when calls into your library have been made, we log before your functions are called, and log the return value of your calls. The logfile is stored as watdfs.log in the directory where you start your client.

Recall from Section 3.1 that *libfuse* expects functions to return `0` or `-errno`. However, most system calls (e.g. `stat`) return `0` or `-1`, and set `errno`. If a system call returns `-1`, your WatDFS server should set the retcode in the RPC call to `-errno`, and return that error to libfuse from the WatDFS client.

Section 6) Requirements (Assignment 2)

Code with Makefile and README

You are required to implement WatDFS using the Remote/Access Model as described in this specification (Section 4). In particular, you are required to implement a client library, and the server. **You must use C++ to implement your solution.**

You should write a `Makefile` that produces:

1. `libwatdfs.a:` a library containing the implementation of your client side library.
    a. Important Note: you should not have a `main` routine in your library; we provide it in `libwatdfsmain.a`
2. `watdfs_server:` an executable of your server
    a. Your server executable should have a main function.
    b. Your server must parse the command line arguments to read the directory of where to persist data (`persist_path` in the examples in Section 6.1).
    c. Your server should register and execute RPC functions.

We will provide:

1. `rpc.h, watdfs_client.h`

    a. Important Note: you must not change these headers
2. `librpc.a`: which contains the RPC library
3. `libwatdfsmain.a`: which implements the client main and sets up FUSE

We will also provide starter code to assist you in beginning the Assignment:

1. `watdfs_client.c`: with function definitions for the required methods, and some comments.
2. `watdfs_server.c`: with a starter `main` method, and some comments
3. `Makefile` that will compile the starter code

Section 6.1) Example Test

To compile your code for a test, we will:

```
# Compile libwatdfs.a and watdfs_server
$ make
# Compile watdfs_client
$ g++ -o watdfs_client -L. -lwatdfsmain -lwatdfs -lrpc -lfuse -pthread
```

To run your code for a test, we will:

Run your server:

```
$ ./watdfs_server <server_persist_dir>
# The persist_path should be where you persist file data, this
# directory should not be located in your home directory which is
# replicated. Instead use a directory local to the machine, e.g.
# /tmp/$USER/server
# when you call rpcExecute it should print
export SERVER_ADDRESS=ubuntu1604-006
export SERVER_PORT=12345
```

Run your client:

```
$ export SERVER_ADDRESS=ubuntu1604-006
$ export SERVER_PORT=12345
$ ./watdfs_client -s -f -o direct_io <path_to_cache>
<client_mount_dir>
# path_to_cache is the directory you should cache files (A3 only)
# it is passed as path_to_cache in watdfs_cli_init.
# client_mount_dir is the directory that the system is mounted as
# all operations to this directory will flow through your library.
# As with the server, these directories should be local to the
# machine, e.g. /tmp/$USER/cache /tmp/$USER/mount
```

You can test your client with bash commands like:

```
$ touch client_mount_dir/454.txt
$ echo “454 is great!” > client_mount_dir/454.txt
$ cat client_mount_dir/454.txt # should print “454 is great!”
$ stat client_mount_dir/test.txt # should print info about the file
```

Important Note: FUSE can run a client multi-threaded. We will always run your client single threaded (-s) and with the -o direct_io flag (to disable kernel caching). Run `./watdfs_client` (no args) to see more options. For example without the (-f) flag, the client will run in the background.

Important Note: If you cannot start your client and get a mount-point error, execute the following:

```
$ fusermount -u <client_mount_dir>
```

Section 7) Submission

You should submit your Assignment on Marmoset as a single zip file for automated testing. Your Assignment zip should contain your code, Makefile and a README file. You can submit to Marmoset using the `marmoset_submit` command.

Marmoset’s build scripts require that your Makefile be titled *Makefile* and that it is placed in the root directory of your submission zip file. If the Makefile is not located in the right location, you will get a compilation error indicating that the Makefile could not be found --- **be sure when you unzip your code that the Makefile is not in a folder!**

```
$ ls
watdfs_client.c
watdfs_client.h
watdfs_server.c
rpc.h
Makefile
$ zip a2_solution.zip *
  adding: watdfs_client.c (deflated 17%)
...
# Submit a2_solution.zip to Marmoset
```

Section 7.1) More details about Marmoset

If your submitted program does not compile or run successfully on its own, your submission will receive a result of "did not compile" and the detailed test results will contain something similar to the error message you get if you ran your program yourself. In this case, your submission will not be tested with any of the tests.

If your submitted program runs successfully on its own, it will be tested with all of the public tests. If it fails any of the public tests, the detailed test results will display an error message for that public test. In this case, your submission will not be tested with any of the release tests.

If it passes all of the public tests, you will have the option to see information for the release tests. If you do so, you will use up one of your "release tokens" for that Assignment. Normally, for every Assignment, you will be initially given 5 release tokens. If you use up one or more of them, one release token will regenerate once every 24 hours. If the deadline will expire before your token regenerates, you can still submit, though you will not be able to tell how your submission did on the tests.

Marmoset automatically tests each submission with all of the release tests, in some order specified by the course staff. If your submission fails a release test and you use a token to see the results, you will see only that test and one more test in the detailed test results. If your submission passes all the release tests, you will not see any release tests in the detailed test results. If you fail a release test, you will get a very small amount of information about what went wrong. You will not be given details of the test case that you failed. **Please do not attempt to guess what that test case might be; do not ask about it on piazza, and do not speculate about test cases on Piazza. The correct action when failing a release test is to re-examine your own test suite and redesign it to find the error in your code or your assumptions.**

Important Note: `recv()` socket calls used the RPC library will hang if they anticipate receiving more data than the other side sends. If a Marmoset test “times out” it is quite likely that you are either not following the protocol for Assignment 2, or you have incorrectly used the RPC library. Be sure to check that you set the length of arguments correctly on both the client and server side!

Section 8) Evaluation (Assignment 2)

While developing your system, you should be aware that we shall be evaluating your system for the following:

1. Code compiling on linux.student.cs.uwaterloo.ca
2. README file
    a. containing build instructions or any other instructions
    b. name and student ID
3. We will be using a number of tests cases to test the following functionality (though we may add or remove particular tests):
    a. Opening or creating a file
        i. In particular we will only use the following flags: `O_CREAT`, `O_APPEND`, `O_EXCL`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
    b. Writing to a file
    c. Reading from a file
    d. Getting file attributes, and updating file modification times

e. Synchronizing a file
f. Truncating a file
g. Closing a file
h. Using appropriate error codes for different error scenarios

4. We will test that Assignment 2 follows the specified protocol for RPC calls.

Section 9) Specific Don'ts

- Do not use any public or web-based source code repository hosting service other than https://git.uwaterloo.ca
- The interface of the WatDFS Client cannot change. We will be using these interfaces to run the FUSE client.
- Do not modify rpc.h, or watdfs_client.h
- Do not assume that the code will compile or run on linux.student.cs without actually compiling and running it on that environment.

It is important that you start this Assignment early!

Section 10) RPC Library Documentation

To ease implementation, an RPC library is provided. The RPC library supports making remote procedure calls from a client to a server. To do so, function handlers and their expected types must be defined and registered before the WatDFS server accepts connections.

Section 10.1) rpcRegister

```
int rpcRegister(char *name, int *argTypes, skeleton f);
//Where skeleton is defined as
typedef int (*skeleton)(int *, void **);
```

Function calls must be registered with the `rpcRegister` call before they can be used by remote clients. This function tells the RPC library what function to call at the server when a client makes an `rpcCall` with a specified named and arguments.

The `skeleton f` is the address of the server function, which corresponds with the server function being registered. Skeletons return an integer error code indicating success with a zero, or failure with a negative error code. **Values that you wish to return should be provided as an argument to the skeleton and indicated as outputs via the `argTypes` array.** `rpcRegister` will return an integer indicating success (`0`) or failure (a negative number).

The `argTypes` array specifies the types of the argument, and whether the argument is an input, output, or both an input and output argument for the function call. Each argument has an integer to encode the type information (described in the following paragraphs), which will collectively form the `argTypes` array. Thus `argTypes[0]` specifies the type information for `args[0]`.

Since it is not known how many arguments there are, the last value in the `argTypes` array must be 0. Consequently, the size of `argTypes` is 1 greater than the size of `args`. The args array is an array of pointers to the different arguments, which will be discussed later. For array arguments, they are specified by pointers in C++, and can be used directly, instead of the addresses of the pointers.
The number of output arguments is arbitrary and they can be positioned anywhere within the args vector.

The argument type integer (of size 4 bytes) will be broken down as follows:
(a) The first byte will specify the input/output nature of the argument. Specifically, if the first bit is set then the argument is input to the server. If the second bit is set the argument is output from the server. The third bit indicates whether the argument is an array type. The remaining 5 bits of this byte are currently undefined and must be set to 0.

(b) The second byte contains argument type information.

```
#define ARG_CHAR    1
#define ARG_SHORT   2
#define ARG_INT     3
#define ARG_LONG    4
#define ARG_DOUBLE  5
#define ARG_FLOAT   6
```

(c) The last two bytes of the argument type integer specify the length of the array. Arrays are limited to a length of 2^16 - 1 (defined as `MAX_ARRAY_LEN`). If the argument is not an array, then these last two bytes must be 0. **It is expected that the client programmer will have reserved sufficient space for any output arrays.**
Important Note: When the function is registered with `rpcRegister`, the server may not know the length of the array; it is therefore acceptable to specify the length of the array as `1` in the call to `rpcRegister`.

For convenience, the following definitions are provided:

```
#define ARG_INPUT      31
#define ARG_OUTPUT     30
#define ARG_ARRAY      29
```

For example, "`(1 << ARG_INPUT) | (1 << ARG_ARRAY) | (ARG_INT << 16) | 20`" represents an array of 20 integers being sent to the server.

We show an example of registering the `sum` function used in the example below.

```
int sumFunction(int *argTypes, void** args) {
      // Expected args are output int and input array of ints
      // get lowest 2 bytes (array len)
      int len = argTypes[1] & ((1 << 16) - 1);
      int total = 0;
      int* to_sum = (int*) args[1];
      for (int pos = 0; pos < len; pos++) {
            total += to_sum[pos];
      }
      *((int *)(args[0])) = total;
      return 0;
}

// result = sum(vector);
#define PARAMETER_COUNT 2       // Number of RPC arguments
#define LENGTH 1                // it is an array


int argTypes[PARAMETER_COUNT+1];

argTypes[0] = (1 << ARG_OUTPUT) | (ARG_INT << 16);// result
// input vector to sum
argTypes[1] = (1 << ARG_INPUT)  | (1 << ARG_ARRAY) |
              (ARG_INT << 16) | LENGTH;

argTypes[2] = 0;                 // Terminator


rpcRegister("sum", argTypes, sumFunction);
```

Important note: The RPC server may call your registered function from any thread that it has available so you should not expect to have any thread-local state around on subsequent function executions!

Section 10.2) rpcCall

```
int rpcCall(char *name, int *argTypes, void **args);
```

The client executes an RPC by calling the `rpcCall` function. The integer returned is the result of executing the `rpcCall`, **not the result of the procedure that the `rpcCall` was executing.** That is, if the `rpcCall` failed, that would be indicated by a negative number and success is indicated by a 0. It is expected that you check the return value for errors. If your function is

expected to return a value, it should be specified as an output argument in the same way as defined in `rpcRegister`.

The `name` argument is the name of the remote procedure to be executed, a procedure that matches the `name` with the same `argTypes` must have been registered at the server. If no such procedure exists, an error code will be returned indicating so.
Thus, if the client wished to execute `result = sum(int vect[LENGTH])`, the code would be:

```
// result = sum(vector);
#define PARAMETER_COUNT 2       // Number of RPC arguments
#define LENGTH 23               // Vector length


int argTypes[PARAMETER_COUNT+1];
void **args = (void **)malloc(PARAMETER_COUNT  * sizeof(void *));


argTypes[0] = (1 << ARG_OUTPUT) | (ARG_INT << 16);      // result
// vector
argTypes[1] = (1 << ARG_INPUT) | (1 << ARG_ARRAY) |
              (ARG_INT << 16) | LENGTH;

argTypes[2] = 0;                 // Terminator

int result;
int vector[LENGTH]; // suppose this is filled in with values
args[0] = (void *)&result;
args[1] = (void *)vector;

int ret = rpcCall("sum", argTypes, args);
// if ret < 0, there is an rpcCall error
// do something with result
free(args); // clean up args
```

Section 10.3) Other RPC library calls

There are 4 other calls that must be made to the RPC library. All these calls return `0` on success, and a negative number for failure.

rpcServerInit

Before registering any functions the server must first call `rpcServerInit`, this will initialize any server state that the RPC library must maintain.

rpcExecute
When the server is ready to serve calls, it should call `rpcExecute`. This will transfer control to the RPC library, and when an RPC call is received the registered functions should be called. `rpcServerInit` must be called before `rpcExecute` is called. When `rpcExecute` is called the server will print the address and port that the server is listening on, which must be set by the client, as described in `rpcClientInit`.

```
export SERVER_ADDRESS=address
export SERVER_PORT=port
```

rpcClientInit
The client should call `rpcClientInit` to initialize a connection with the server. `rpcClientInit` should be called before any calls to `rpcCall` are made. To connect with the server the environment variables `SERVER_ADDRESS` and `SERVER_PORT` must be set before the client binary is executed. For example:

```
$ export SERVER_ADDRESS=ubuntu1604-006
$ export SERVER_PORT=12345
$ ./client <args>
```

rpcClientDestroy
The client should call `rpcClientDestroy` when they are finished interacting with the server. This will terminate connections with the server.

RPC Library Tips
The RPC library logs each call, the argument types, and the return value in rpc_client.log and rpc_server.log respectively. Consult these logs if your RPC calls do not succeed.

Section 11) Other Resources

Section 11.1) Resources for working with FUSE
The FUSE headers are available online:
https://github.com/libfuse/libfuse/blob/master/include/fuse.h

A general tutorial for the FUSE functions are available online at:
https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/html/index.html

Some pointers on the purpose of each FUSE call:
https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html

Section 11.2) Resources for system calls

You will have to make many system calls. To understand their function, you should read the man pages. Some system calls you might need are listed below

System Calls	Description	Links
open, creat	Opening a file	http://man7.org/linux/man-pages/man2/open.2.html
close	Closing a file	http://man7.org/linux/man-pages/man2/close.2.html
mknod	Creating a file	http://man7.org/linux/man-pages/man2/mknod.2.html
read/write	Reading and writing to a file	http://man7.org/linux/man-pages/man2/read.2.html
pread, pwrite	Reading and writing a file to offsets	http://man7.org/linux/man-pages/man2/pread.2.html
stat, lstat, fstat	Getting file attributes	http://man7.org/linux/man-pages/man2/fstat.2.html
truncate ftruncate	Truncating a file to a length	http://man7.org/linux/man-pages/man2/truncate.2.html
fsync	Sync a file to storage	http://man7.org/linux/man-pages/man2/fsync.2.html
futimens, utimensat	Change file modified/access times	http://man7.org/linux/man-pages/man2/utimensat.2.html

Section 11.3) General Resources

For those who are not familiar with the process of creating a static library, consult the following link: http://tldp.org/HOWTO/Program-Library-HOWTO/introduction.html

In coding this Assignment, you may find helpful to use a debugger, e.g., gdb is available on CSCF environment; a gdb tutorial is available at: https://beej.us/guide/bggdb/

POSIX threads (pthreads) is a standardized interface for threads on UNIX systems and a great tutorial with code examples can be found here: https://computing.llnl.gov/tutorials/pthreads/
A more general tutorial on threads including details of different threading interfaces is given in: http://www.cs.cf.ac.uk/Dave/C/node29.html

Assignment 3

Section 12) Upload Download Model (Assignment 3)

In Assignment 3, you will implement WatDFS using the upload-download model.

Section 12.1) Objective

In the upload-download model, the file is copied from the server to the client, and the client performs all operations locally. In this Assignment, you will use a timeout-based caching method and mutual exclusion of writes to ensure consistency.

In Assignment 3, you should modify your Assignment 2 solution and implement new RPC calls to support the upload-download model. In Assignment 3, you are free to use any protocol that you wish, although we make some suggestions below (Section 13). Consequently, we will be testing for end-to-end system correctness.

Hint: You should review Sections 3 and 10 to remind yourself of the FUSE operations and the RPC library functions.

Section 12.2) Mutual Exclusion

To ensure mutual exclusion of writes, the server should return an error (`-EACCES`) if a client attempts to `open` a file (with write mode) and the file is already opened (in write mode). The file can be opened again by another client once the file has been closed (`watdfs_cli_release`). Any number of concurrent readers (from multiple FUSE instances) to a file are allowed.

It is prohibited for the same client to open the same file multiple times. If a client attempts to `open` a file that is already open, the client should return an error (`-EMFILE`). `-EMFILE` has precedence over `-EACCES`, i.e., the `-EMFILE` error condition should be checked first.

In summary, you are required to support multiple clients that can read a file concurrently but only one client may be a writer of the file.

Section 12.2) Timeout-based Caching

You will implement a variant of NFS caching semantics using timeouts. Under this caching scheme, a cached object may be used to satisfy a read request if it satisfies the **freshness condition** at time *T* for some freshness interval *t.*

Let *Tc* be the time the cache entry was last validated by the client. *T_client* represents the time that the file was last modified as recorded by the client. *T_server* represents the time that the file was last modified as recorded by the server.

The freshness condition at time *T* for a file that is opened for only reads is defined as follows. If at time T: *[(T - Tc) < t]* or *[T_client == T_server]* then the file can be served from the locally cached file copy. Otherwise, a new copy of the file must be retrieved from the server and cached at the client. Note that *[T - Tc) < t]* **should** be determined without accessing the server.

You should determine *T_server* by consulting the server when necessary. If *T_server* is retrieved and is equal to *T_client* then *Tc* should be updated to the current time. Otherwise, the file has changed at the server and a fresh copy should be fetched from the server to the client. After fetching, you should update *T_client* to be equal to *T_server,* and update *Tc* to the current time.

The client should also execute writes against the locally cached file according to the freshness condition. Writes by a client should be applied to the local copy of the file, and periodically written back to the server. You will do this by checking the freshness condition for writes at the end of a `write/truncate` call. If at that time *T*, *[(T - Tc) < t]* or *[T_client == T_server]*, then you will return immediately without writing back the file Otherwise, you must write client's copy of the file back to the server and *T_server* must be updated to *T_client*. When the client is done with the file, as indicated by `watdfs_cli_release`, the file should be written back to the server.[7]

Note, that you do not need to write a file back to the server asynchronously (or read a file from the server asynchronously), simply check if the freshness condition holds anytime the file is accessed, and write the file to the server (or read the file from the server). However, **if a client application issues `fsync` (only possible if the file is opened in write mode), then the client's copy of the file must be written to the server immediately and *T_server* and *Tc* should be updated.**

In Assignment 3, copying files to and from the server must be performed atomically. As an example, if the file *F* is being written to the server by client *C1*, and *F* is being read from the server by a second client *C2*, then *C2* should see all or none of the updates made to *F* by *C1*.

Section 12.3) Client side cache

Recall the definition of `watdfs_cli_init` from Section 3.2:

```
void *watdfs_cli_init(struct fuse_conn_info *conn,
                      const char *path_to_cache,
                      time_t cache_interval);
```

[7] During `watdfs_cli_release` the client copy of the file should be closed (`close` system call), but it should remain in the cache directory. The cached copy of the file should not be deleted.

The client should persist data (cached copies of files) in the `path_to_cache` directory. All client operations should be performed on its copy of the file, using the local file system. The client **should not** store its local copies of files in memory. That is, when a client performs a write, make a write system call against the client's copy of the file on disk --- simply writing to an in-memory buffer is incorrect!
The cache interval *t*, in seconds, is passed as `cache_interval`. `cache_interval` is determined by setting the environment variable `CACHE_INTERVAL_SEC` before executing the client.

Section 12.4) Server side
As in the remote access model, the server should perform operations on the file system and not buffer writes or cache the file in memory.

Section 13) Suggested Implementation Strategy (Assignment 3)
We now define a suggested method for implementing the upload-download model in WatDFS. Strictly speaking, this is hidden behind the API and so you may choose to implement Assignment 3 any way that you wish. However, it is recommended that you at least try and understand this method before developing your own.

Section 13.1) Opening a file
When a file is opened, it is opened with a file access mode (i.e. `O_RDONLY`, `O_WRONLY`, `O_RDWR`). However, since files opened on the server are always read and files opened on the client are always written (to create local cached copies), these modes cannot be passed directly to `open` on the client or server file system. Consequently, the file access mode should be adjusted to support copying the file. On subsequent operations, the client should ensure that the **original** access mode passed to `open` allows the operation! Otherwise, the client should return `-EPERM`.

When `watdfs_cli_open` is called, you should copy the server's copy of the file to the client so the client can apply operations locally. Therefore, `open` should be called at both the client and the server as part of caching the file locally, resulting in two different file descriptors which you should track at the client. As part of these `open` calls, you should satisfy the mutual exclusion requirements (suggestions in Section 13.2).

Opening a file should also initialize metadata at the client that is needed to check the freshness condition for the file (*Tc*). You can use the file modification times of the underlying file systems to track *T_client* and *T_server*.

Section 13.2) Enforcing mutual exclusion
To ensure that a file has only a single open writer at a time, the server should keep track of open files. The server should maintain a thread synchronized data structure that maps filenames to their status (open for write, open for read, etc.). If the server receives an open

request for write to a file that has already been opened in write mode, the server will use this data structure to discover the conflict and return `-EACCES`. When the server receives a message to close a file it has opened in write mode, the data structure should be modified to indicate that the file is now available to a writer.

To ensure that the same client does not open the same file multiple times, you must track current open files at the client. You should use your already tracked metadata to determine if the file has already been opened; if it has return `-EMFILE`.

Section 13.3) Releasing a file

If the file was opened in write mode, the file should be flushed from the client to the server (described below). The file should then be closed at both the client and the server. The *`release`* RPC call from Assignment 2 can be modified to incorporate the mutual exclusion protocol. The file metadata that is tracked at the client should also be released.

Section 13.4) Steps for transferring the file from the server to the client

To transfer a file from the server to the client you should: truncate the file at the client, get file attributes from the server, read the file from the server, write the file to the client, and then update the file metadata at the client. You should reuse RPC calls from Assignment 2 to complete this task.

Section 13.5) Ensuring file transfers are atomic

To ensure that file transfers are atomic, you should implement two RPC calls to indicate that a file is in transfer either to or from the server. A file can be transferred from a server to multiple clients in parallel. However, writing back the file to the server should be mutually exclusive to these reads and other write backs. To achieve these atomic transfers you should mark a file as in transfer for reads or in transfer for writes.

To assist you with this task, we have provided an implementation of the reader-writer locks, in the files: `rw_lock.h` and `rw_lock.c` which define the struct `rw_lock_t`, and methods to acquire and release locks in read or write mode.

The API of the `rw_lock_t` is similar to that of `pthread_mutex_t`, and is defined as follows (more details are in `rw_lock.h`). All functions return `0` on success, and `-errno` (`< 0`) on failure.

To setup and tear down the lock, use the following functions:

```
int rw_lock_init(rw_lock_t* lock); // initialize the lock
int rw_lock_destroy(rw_lock_t* lock); // destroy the lock
```

To acquire and release the lock, use the following functions:

```
// acquire the lock in mode  (RW_READ_LOCK, RW_WRITE_LOCK)
int rw_lock_lock(rw_lock_t* lock, rw_lock_mode_t mode);

// release the lock from mode  (RW_READ_LOCK, RW_WRITE_LOCK),
// you must have been an owner of the lock to release it
int rw_lock_unlock(rw_lock_t* lock, rw_lock_mode_t mode);
```

Given the reader-writer lock definition, the following two RPC calls should be implemented:

1. `lock(path, mode)`
2. `unlock(path, mode)`

These calls lock and unlock the file (located at `path`) for transfer using the read or write `lock_mode`, respectively.

Section 13.6) Calls on files that have not been opened

Recall that some calls may take place on files that have not been opened, for instance `watdfs_cli_truncate` and `watdfs_cli_getattr`. To implement these calls, you should check if the file has been opened at the client. If the file has not been opened, you should open the file, transfer the file, perform the operation, transfer the file to server (if appropriate), and then close the file.

Section 13.7) The Effects of Asynchronous `release`

Recall from the description of the *libfuse* functions that `watdfs_cli_release` is called asynchronously after a file is closed. Specifically, when `close` is called on a file, *libfuse* will call `watdfs_cli_release` but will not wait for `watdfs_cli_release` to return before `close` returns. However, because `watdfs_cli_release` is responsible for transferring a writable file from the client to server and unlocking it, subsequent `open`/`close` calls to the same file may not succeed because `watdfs_cli_release` has not completed. It is for this reason that we **require** that if `release` has not yet completed on the same file, `open` should fail with `EMFILE`. The automated tests will look for this and loop over `open`() with `EMFILE` to open a file that has just been closed. When you are writing your own test applications, you should look at the `errno` when `open` calls fail on subsequent open/close operations, and if it is `EMFILE`, retry the `open` operation.[8]

Section 14) More Tips

Note, that the flags that are passed with `open` are set in the `fi->flags` structure in both the `open` and `release` calls.

[8] This looping behaviour should only be done in your testing applications, and not in your *libfuse* code! Since we are running FUSE in single-threaded mode, looping in your library code on `EMFILE` will result in a deadlock!

When a file is opened, it may exist at the server, but not at the client. The file should be created at the client which may necessitate additional changes to the flags passed to `open`.

Section 15) System Manual (Assignment 3)

For Assignment 3, you must submit a system manual as a PDF file with your code. The system manual should include at least the following items (you can add other details if you wish but be concise)

1. You should discuss your design choices so that we can understand how key functionality was implemented. For instance, how the server ensures a single writer to a file happens at a time, the steps you take to copy a file from the client to server and vice-versa, and the sequence of RPC calls that are made to check the invalidation state.
2. List all error codes that you returned (outside of error codes returned from system calls directly)
3. You should discuss how you have tested your Assignment (e.g., describe a series of system calls that you have made to ensure your Assignment works)
4. Clearly identify any functionality of the Assignment has not been implemented.

Section 16) Requirements, Evaluation and Submission (Assignment 3)

Section 16.1) Requirements

You are required to implement WatDFS using the Upload/Download model as described in this specification. In particular, you are required to implement the client library and the server.

The code and Makefile requirements for submission of Assignment 3 are the same as Assignment 2 (Section 6). In addition to your code, README, and Makefile, you must also submit your system manual (Section 15).

In addition to the files provided in Assignment 2, we provide:

1. `rw_lock.h, rw_lock.c` these contain an implementation of a reader-writer lock, as defined in Section 13.5.

To run your client and server we will run the commands listed in Section 6.1, but will modify the execution of the client as follows:

```
$ export SERVER_ADDRESS=ubuntu1604-006
$ export SERVER_PORT=12345
# New for A3:
$ export CACHE_INTERVAL_SEC=5 # or any integer value >= 0
$ ./watdfs_client -s -f -o direct_io <path_to_cache>
<client_mount_dir>
# path_to_cache is the directory you should cache files
```

```
# it is passed as path_to_cache in watdfs_cli_init
# client_mount_dir is the directory that the system is mounted as
# all operations to this directory will flow through your library.
# As with the server, these directories should be local to the
# machine, e.g. /tmp/$USER/cache /tmp/$USER/mount
```

Section 16.2) Submission

You should submit your Assignment on Marmoset as a single zip file for automated testing. Your Assignment zip should contain your code, Makefile, README and system manual. More details about Marmoset and submission are found in Section 7.

Section 16.3) Evaluation

While developing your system, you should be aware that we shall be evaluating your system for the following:

1. Items 1 and 2 in Section 8 (Code compiling and README)
2. Detailed documentation in the system manual.
3. We will use a number of test cases to test the following functionality in addition to Item 3 in Section 8:
    a. Mutual exclusion on opens (write mode), and multiple opens by the same client to the same file that have not yet been closed
    b. Concurrent readers and single writer to the same file
    c. Accesses to the server version of the file are atomic
    d. Client caches the file and respects the freshness condition
    e. `fsync` triggers a flush of the file to the server

Section 16.4) Specific Do's and Don'ts

You should follow the guidelines in Section 9. **While doing Assignment 3, you can re-use your code from Assignment 2. Do not use code from a source other than your own Assignment 2.**

It is important that you start this Assignment early!