

ecet4640-lab5

1.0

Generated by Doxygen 1.9.8

1 ecet4640-lab5	1
1.1 Intro	1
1.2 Contributions	1
1.3 Overview	2
2 Compilation	3
2.1 Compilation Pipelines	3
2.2 Compiling and running	3
2.3 Screenshot of Compilation	4
2.4 Cleaning	4
3 Topic Index	5
3.1 Topics	5
4 Data Structure Index	7
4.1 Data Structures	7
5 File Index	9
5.1 File List	9
6 Topic Documentation	11
6.1 Build	11
6.1.1 Detailed Description	11
6.1.2 Function Documentation	11
6.1.2.1 CreateUsersArray()	11
6.1.2.2 CreateUsersMap()	12
6.2 Connection	12
6.2.1 Detailed Description	13
6.2.2 Function Documentation	13
6.2.2.1 InitializeShared()	13
6.2.2.2 StartUpdateThread()	14
6.2.2.3 StartConnectionThread()	14
6.2.2.4 MessageOrClose()	16
6.2.2.5 MessageAndClose()	17
6.2.2.6 _help()	17
6.2.2.7 _register()	18
6.2.2.8 _myinfo()	19
6.2.2.9 _who()	19
6.2.2.10 _rand_gpa()	20
6.2.2.11 _rand_age()	21
6.2.2.12 _advertisement()	22
6.3 Data	22
6.3.1 Detailed Description	23
6.3.2 Macro Definition Documentation	23

6.3.2.1 RECORD_COUNT	23
6.3.2.2 ID_MAX_LENGTH	23
6.3.2.3 NAME_MAX_LENGTH	23
6.3.2.4 IP_LENGTH	24
6.3.3 Variable Documentation	24
6.3.3.1 accepted_userIDs [1/2]	24
6.3.3.2 userFullNames [1/2]	24
6.3.3.3 accepted_userIDs [2/2]	25
6.3.3.4 userFullNames [2/2]	25
6.4 Files	25
6.4.1 Detailed Description	26
6.4.2 Macro Definition Documentation	26
6.4.2.1 LOCKFILE	26
6.4.2.2 REGISTERED_FILE	26
6.4.2.3 SERVER_SETTINGS_FILE	26
6.4.2.4 ADS_DIR	27
6.4.3 Function Documentation	27
6.4.3.1 FileStatus()	27
6.4.3.2 CreateOrOpenFileVerbose()	27
6.4.3.3 ReadRegisteredFileIntoUsersMap()	28
6.4.3.4 UpdateRegisteredFileFromUsersMap()	29
6.4.3.5 NumberOfFilesInDirectory()	30
6.4.3.6 GetRandomFileNameFromDir()	31
6.4.3.7 ReadSettingsFileIntoSettingsMap()	31
6.4.3.8 CatFileToBuffer()	32
6.5 Log	33
6.5.1 Detailed Description	34
6.5.2 Function Documentation	34
6.5.2.1 LogfFatal()	34
6.5.2.2 LogfError()	34
6.5.2.3 LogfWarning()	35
6.5.2.4 LogfInfo()	35
6.5.2.5 LogfDebug()	35
6.5.2.6 LogfTrace()	36
6.5.2.7 InitializeLogger()	36
6.6 Map	37
6.6.1 Detailed Description	37
6.6.2 Function Documentation	38
6.6.2.1 NewMap()	38
6.6.2.2 Map_Set()	38
6.6.2.3 Map_Get()	39
6.6.2.4 Map_Delete()	39

6.7 Server	40
6.7.1 Detailed Description	40
6.7.2 Function Documentation	40
6.7.2.1 StartServer()	40
6.7.2.2 NextAvailableConnection()	41
6.7.2.3 InitializeServer()	42
6.8 Util	42
6.8.1 Detailed Description	42
6.8.2 Macro Definition Documentation	42
6.8.2.1 COLOR_RED	42
6.8.2.2 COLOR_GREEN	43
6.8.2.3 COLOR_YELLOW	43
6.8.2.4 COLOR_BLUE	43
6.8.2.5 COLOR_RESET	43
6.8.3 Function Documentation	43
6.8.3.1 printRed()	43
6.8.3.2 printGreen()	44
6.8.3.3 printYellow()	44
6.8.3.4 printBlue()	45
6.8.3.5 RandomInteger()	46
6.8.3.6 RandomFloat()	46
6.8.3.7 RandomFlag()	47
6.9 Process	47
6.9.1 Detailed Description	48
6.9.2 Function Documentation	48
6.9.2.1 RunHeadless()	48
6.9.3 Variable Documentation	48
6.9.3.1 users_array	48
6.9.3.2 users_map	49
6.9.3.3 settings_map	49
6.9.3.4 default_settings	49
6.9.3.5 active_clients	49
7 Data Structure Documentation	51
7.1 _map_bucket Struct Reference	51
7.1.1 Detailed Description	51
7.2 ClientShared Struct Reference	51
7.2.1 Detailed Description	52
7.2.2 Field Documentation	52
7.2.2.1 users	52
7.2.2.2 mutex	52
7.2.2.3 dirty	52

7.2.2.4 shutting_down	52
7.2.2.5 send_buffer_size	53
7.2.2.6 receive_buffer_size	53
7.3 Connection Struct Reference	53
7.3.1 Detailed Description	54
7.3.2 Field Documentation	54
7.3.2.1 status	54
7.3.2.2 socket	54
7.3.2.3 address	54
7.3.2.4 address_length	54
7.3.2.5 thread_id	54
7.3.2.6 time_connected	55
7.3.2.7 state	55
7.3.2.8 user	55
7.4 LogSettings Struct Reference	55
7.4.1 Detailed Description	55
7.5 map Struct Reference	56
7.5.1 Detailed Description	56
7.6 map_result Struct Reference	56
7.6.1 Detailed Description	56
7.7 ServerProperties Struct Reference	57
7.7.1 Detailed Description	57
7.7.2 Field Documentation	57
7.7.2.1 port	57
7.7.2.2 send_buffer_size	57
7.7.2.3 receive_buffer_size	57
7.7.2.4 backlog	58
7.7.2.5 active_connections	58
7.7.2.6 max_connections	58
7.7.2.7 time_started	58
7.8 User Struct Reference	58
7.8.1 Detailed Description	59
8 File Documentation	61
8.1 Build.c	61
8.2 Build.h	61
8.3 Connection.c	61
8.4 Connection.h	65
8.5 Data.c	66
8.6 Data.h	67
8.7 File.c	67
8.8 File.h	70

8.9 Log.c	70
8.10 Log.h	71
8.11 main.c	72
8.12 map.c	72
8.13 map.h	74
8.14 Process.c	75
8.15 Process.h	78
8.16 Server.c	78
8.17 Server.h	81
8.18 Util.c	81
8.19 Util.h	82
Index	83

Chapter 1

ecet4640-lab5

1.1 Intro

This program starts a Ipv4 server that listens on a port for incoming connections. For each new connection, it starts a thread. Connecting users send strings to execute various actions on the server.

1.2 Contributions

- On 9/29, Christian made the initial repository.
- On 9/29, Karl made the Makefile, copied the example client and server, created the Build, Data, File, Process, Util, and map modules.
- On 9/30, Karl added logic for reading the registered users file into a map and initializing/binding the server socket; created the Server and [Connection](#) modules.
- On 10/1, Paul and Karl did work on reading the server-settings.txt file, and creating a new thread for a client connection.
- On 10/2, Christian added a Log module and added some functionality to support various log levels.
- On 10/5, Paul, Karl, and Christian began adding different command line arguments the server could take.
- On 10/5, Karl and Paul fixed some bugs related to the multithreading.
- On 10/5, Paul added the myinfo and register commands and fixed typos.
- On 10/8, Christian added features for initializing the logger and fixed a bug related to disconnects.
- On 10/12, Karl, Paul and Christian changed the message format to the current format, set commands to be lowercase, and improved the messages sent to users.
- On 10/16, Christian added features to the `_rand_age` function.
- On 10/18, Karl fixed a segfault bug that occurred when users entered an invalid ID.
- On 10/19, Christian added the call to updating the registered file as users register
- On 10/19, Christian, Karl, and Paul added more command line arguments to the server.
- On 10/21, Christian implemented the advertisement feature and removed debug prints.

1.3 Overview

When the program runs, it reads from server-settings.txt to determine how it should be configured. As users register, they are added to a registered.txt file. The Users data structure has a dirty flag and a mutex that indicates when user data has been changed. A separate cleanup thread checks this dirty flag every few seconds to see if the registered.txt file should be updated.

Only one server process should be running at a given time. To that end, a running server creates a lockfile in the /tmp folder and deletes the lockfile when it is done. New servers will not be started if a lockfile exists, but the running server can be stopped by passing the command line argument 'stop' to the binary. There are other command line arguments available, as detailed below.

Argument	Description	Calls
none	Defaults to RunCommand; runs server attached to terminal	RunCommand()
headless	Runs the server with .nohup, as a background process.	RunHeadless()
stop	Stops an existing server process if it is running.	StopCommand()

Author

Karl Miller
Paul Shriner
Christian Messmer

Chapter 2

Compilation

2.1 Compilation Pipelines

There are several compilation pipelines, which are described in more detail in the Makefile comments.

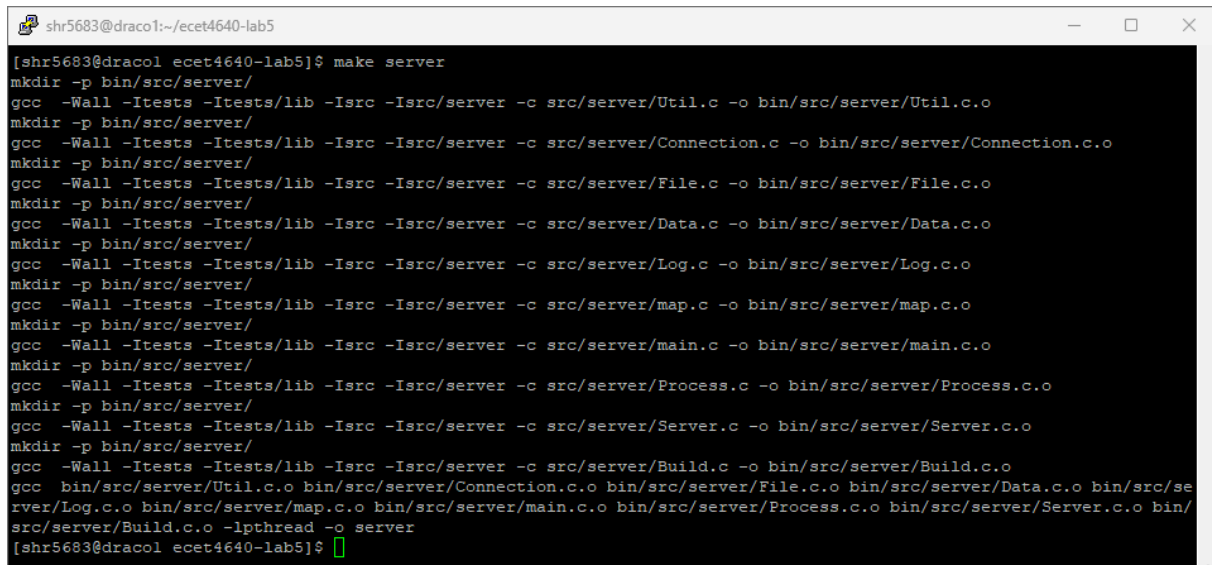
The first is for making and running the regular server process. Calling `make` executes this. It uses the files in `src/server` to generate the binary and runs it. This will run the binary after it is built, and the default command will cause it to run in the server. Executing `make server` will make the server binary without running it.

The second is for making the test binary. This compiles the files in `tests` and the files in `src/server`, but excludes `src/main.c` so that `tests/main_test.c` will be the program entry point instead. The tests use `CuTest`. The tests are not documented here in order to not inflate the documentation size any further.

2.2 Compiling and running

1. Copy the .zip file to the server.
2. Extract the zip file.
3. Enter the unzipped folder.
4. Run `make server`
5. Run `./server` to run the server attached to the shell.
6. Press `ctrl+c` to exit and close the server.
7. Run `./server headless` to run the server headlessly.
8. Run `./server stop` to stop the headless server.
9. If a better client is not available, you can use the example client to connect.
10. `cd` into the `/example` folder
11. run `gcc client.c`
12. run `./a.out` and input '3000' as the port.

2.3 Screenshot of Compilation



```
[shr5683@draco1:~/ecet4640-lab5]$ make server
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Util.c -o bin/src/server/Util.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Connection.c -o bin/src/server/Connection.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/File.c -o bin/src/server/File.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Data.c -o bin/src/server/Data.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Log.c -o bin/src/server/Log.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/map.c -o bin/src/server/map.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/main.c -o bin/src/server/main.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Process.c -o bin/src/server/Process.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Server.c -o bin/src/server/Server.c.o
mkdir -p bin/src/server/
gcc -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Build.c -o bin/src/server/Build.c.o
gcc bin/src/server/Util.c.o bin/src/server/Connection.c.o bin/src/server/File.c.o bin/src/server/Data.c.o bin/src/se
rver/Log.c.o bin/src/server/map.c.o bin/src/server/main.c.o bin/src/server/Process.c.o bin/src/server/Server.c.o bin/
src/server/Build.c.o -lpthread -o server
[shr5683@draco1:~/ecet4640-lab5]$
```

Figure 2.1 Compiling on draco1

2.4 Cleaning

`make clean` will clean all `.o` files and binaries.

Chapter 3

Topic Index

3.1 Topics

Here is a list of all topics with brief descriptions:

Build	11
Connection	12
Data	22
Files	25
Log	33
Map	37
Server	40
Util	42
Process	47

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

_map_bucket	Map_bucket is an endpoint in the map. It is also a node in a linked list; if there were collisions, then the buckets are appended to the linked list at that location, then traversed until the matching key is found	51
ClientShared	51
Connection	53
LogSettings	55
map	A map. Stores key-value pairs for near constant lookup and insertion time	56
map_result	The result of a map retrieval	56
ServerProperties	57
User	58

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

src/server/Build.c	61
src/server/Build.h	61
src/server/Connection.c	61
src/server/Connection.h	65
src/server/Data.c	66
src/server/Data.h	67
src/server/File.c	67
src/server/File.h	70
src/server/Log.c	70
src/server/Log.h	71
src/server/main.c	72
src/server/map.c	72
src/server/map.h	74
src/server/Process.c	75
src/server/Process.h	78
src/server/Server.c	78
src/server/Server.h	81
src/server/Util.c	81
src/server/Util.h	82

Chapter 6

Topic Documentation

6.1 Build

Functions for creating and populating data structures.

Functions

- [User](#) * [CreateUsersArray](#) (char **userIDs, char **userNames, int recordsCount)
- [map](#) * [CreateUsersMap](#) ([User](#) *usersArray, int recordsCount)

6.1.1 Detailed Description

6.1.2 Function Documentation

6.1.2.1 CreateUsersArray()

```
User * CreateUsersArray (  
    char ** userIDs,  
    char ** userNames,  
    int recordsCount )
```

Mallocs a new array of [User](#) structs.

Parameters

<i>userIDs</i>	An array of userIDs to set.
<i>userNames</i>	An array of userNames corresponding to the userIDs.
<i>recordsCount</i>	The number of records in userIDs and userNames, and the size of the created an array.

Returns

A malloced array of user structs.

Definition at line 9 of file [Build.c](#).

6.1.2.2 CreateUsersMap()

```
map * CreateUsersMap (
    User * usersArray,
    int recordsCount )
```

Given a user's array, initializes a new map that points to the underlying data in the array, using the user's ID as a key.

Parameters

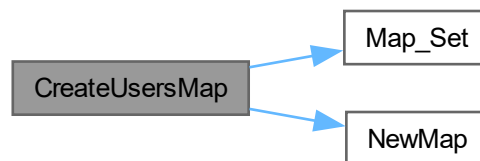
<i>usersArray</i>	The array used to build the user's map.
<i>recordsCount</i>	The number of records in the user's array.

Returns

A map

Definition at line 23 of file [Build.c](#).

Here is the call graph for this function:



6.2 Connection

This module handles an individual user's active connection.

Data Structures

- struct [ClientShared](#)
- struct [Connection](#)

Functions

- `ClientShared * InitializeShared (map *users_map, size_t send_buffer_size, size_t receive_buffer_size)`
- `void * StartUpdateThread (void *parameter)`
- `void * StartConnectionThread (void *p_connection)`
- `int MessageOrClose (char *send_buffer, char *receive_buffer, Connection *connection)`
- `void MessageAndClose (char *send_buffer, Connection *connection)`
- `void _help (Connection *connection, char *response)`
- `int _register (Connection *connection, char *response)`
- `int _myinfo (Connection *connection, char *response)`
- `void _who (char *response)`
- `void _rand_gpa (Connection *connection, char *response)`
- `void _rand_age (Connection *connection, char *response)`
- `void _advertisement (Connection *connection, char *response)`

responds with a random ascii art

6.2.1 Detailed Description

6.2.2 Function Documentation

6.2.2.1 InitializeShared()

```
ClientShared * InitializeShared (
    map * users_map,
    size_t send_buffer_size,
    size_t receive_buffer_size )
```

Initializes the structure that shares data between connections and the server.

Parameters

<code>users_map</code>	The map of <code>User</code> structs.
------------------------	---------------------------------------

Returns

A pointer to the same `ClientShared` object seen by the connection threads.

Definition at line 19 of file `Connection.c`.

Here is the caller graph for this function:



6.2.2.2 StartUpdateThread()

```
void * StartUpdateThread (
    void * parameter )
```

Starts an update thread. This thread is responsible for checking `shared.dirty`. If it is, it writes the user's data to a file and sets dirty to 0.

Parameters

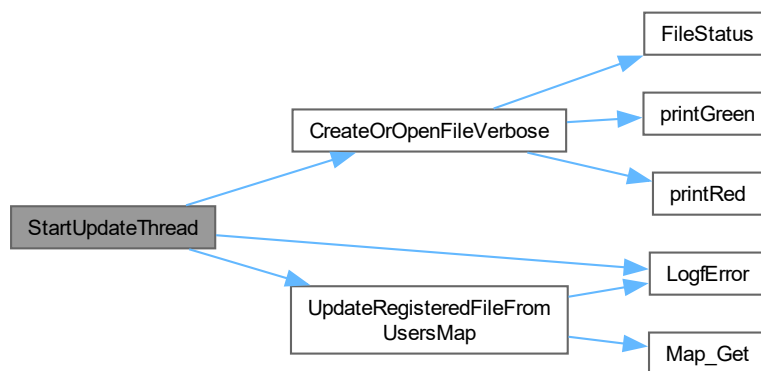
<i>parameter</i>	None.
------------------	-------

Returns

NULL

Definition at line 30 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.3 StartConnectionThread()

```
void * StartConnectionThread (
    void * connection )
```

Starts a connection thread

Parameters

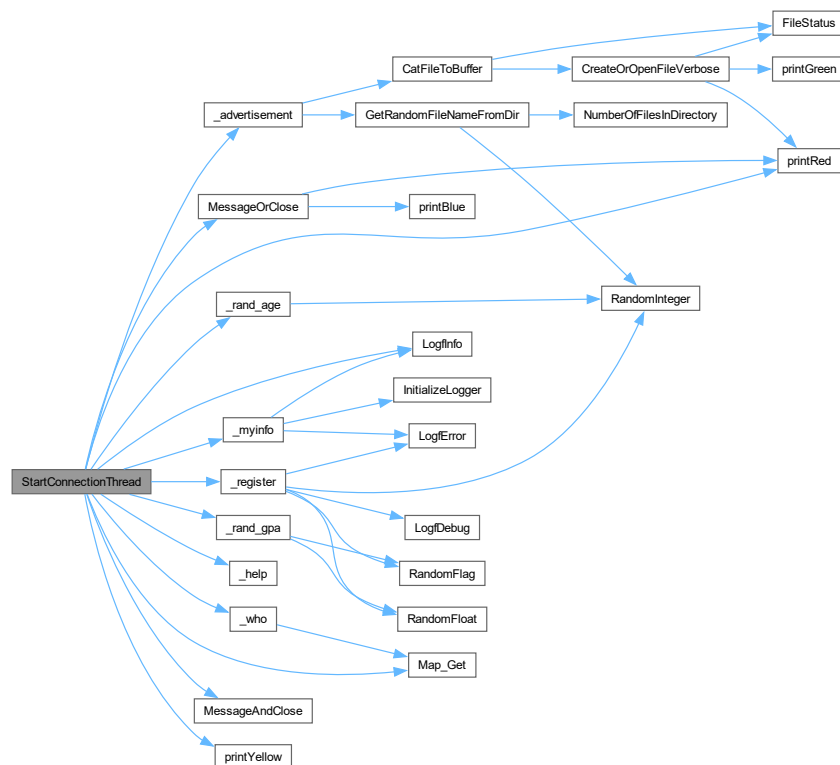
<i>connection</i>	A pointer to a Connection structure from the server's connections array.
-------------------	--

Returns

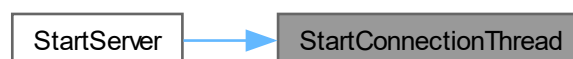
NULL

Definition at line 52 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.4 MessageOrClose()

```
int MessageOrClose (
    char * send_buffer,
    char * receive_buffer,
    Connection * connection )
```

Sends `send_buffer` to the socket referenced by `connection`, then memsets `send_buffer` to 0. Memsets `receive_buffer` to 0, then receives a message from the client. If this length is 0, assumes the connection was closed and sets `connection->active` to 0.

Warning

`send_buffer` and `receive_buffer` must be the size specified in `shared`.

Parameters

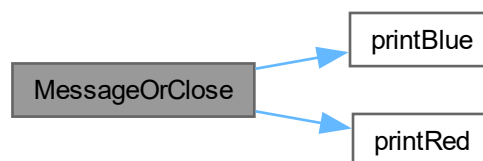
<code>send_buffer</code>	A message to send to the client.
<code>receive_buffer</code>	The message received by the client.
<code>connection</code>	The socket's Connection

Returns

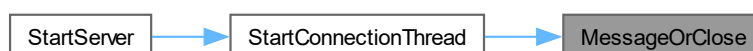
The number of bytes read into `receive_buffer`, or 0 if the connection closed.

Definition at line 163 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.5 MessageAndClose()

```
void MessageAndClose (
    char * send_buffer,
    Connection * connection )
```

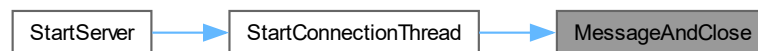
Sends send_buffer to the socket referenced by connection, then sets connection.active to 0.

Parameters

<i>send_buffer</i>	The send buffer. Should be shared.send_length in size.
<i>connection</i>	The socket's Connection .

Definition at line 191 of file [Connection.c](#).

Here is the caller graph for this function:



6.2.2.6 _help()

```
void _help (
    Connection * connection,
    char * response )
```

Returns the functions available to the user

Parameters

<i>connection</i>	connection the user is on
<i>response</i>	fills the response buffer with what to send to the client

Definition at line 200 of file [Connection.c](#).

Here is the caller graph for this function:



6.2.2.7 _register()

```
int _register (
    Connection * connection,
    char * response )
```

Registers the user from connection

Parameters

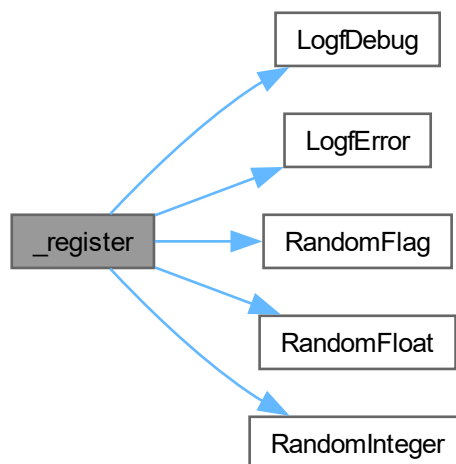
<i>connection</i>	connection the users is on
<i>response</i>	fills the response buffer with what to send to the client

Returns

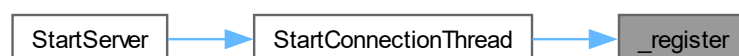
int 1 if successful, 0 if not

Definition at line 216 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.8 _myinfo()

```
int _myinfo (
    Connection * connection,
    char * response )
```

Returns the info of the user to the client

Parameters

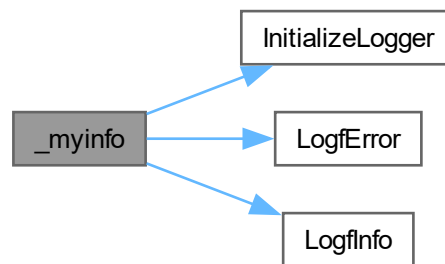
<i>connection</i>	connection the user is on
<i>response</i>	fills the response buffer with what to send to the client

Returns

int 1 if successful, 0 if not

Definition at line 252 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.9 _who()

```
void _who (
    char * response )
```

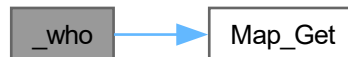
Sets response buffer to be a list a userIDs that are connected.

Parameters

<i>response</i>	fills the response buffer with what to send to the client
-----------------	---

Definition at line 273 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.10 _rand_gpa()

```

void _rand_gpa (
    Connection * connection,
    char * response )
  
```

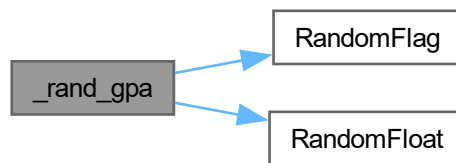
Randomly changes the gpa of the user

Parameters

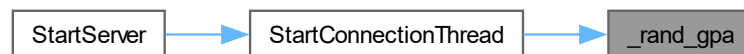
<i>connection</i>	connection the user is on
<i>response</i>	fills the response buffer with what to send to the client

Definition at line 288 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.11 _rand_age()

```

void _rand_age (
    Connection * connection,
    char * response )
  
```

Randomly changes the age of the user

Parameters

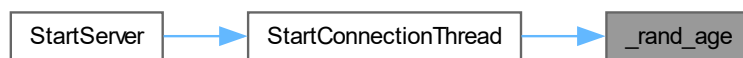
<i>connection</i>	connection the user is on
<i>response</i>	fills the response buffer with what to send to the client

Definition at line 303 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.12 _advertisement()

```

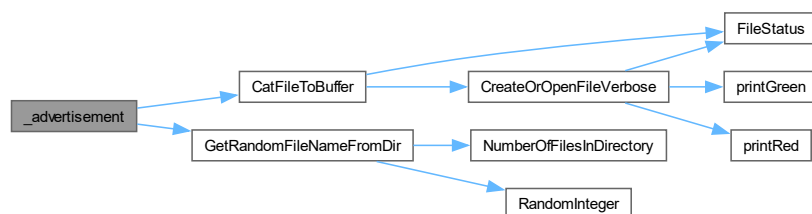
void _advertisement (
    Connection * connection,
    char * response )
  
```

Parameters

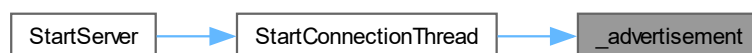
<i>connection</i>	connection the user is on
<i>response</i>	fills the response buffer with what to send to the client

Definition at line 315 of file [Connection.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.3 Data

This module describes structures used in this program.

Data Structures

- struct [User](#)

Macros

- `#define` [RECORD_COUNT](#) 17
- `#define` [ID_MAX_LENGTH](#) 9
- `#define` [NAME_MAX_LENGTH](#) 21
- `#define` [IP_LENGTH](#) 16

Variables

- `char *` [accepted_userIDs](#) []
- `char *` [userFullNames](#) []
- `char *` [accepted_userIDs](#) []
- `char *` [userFullNames](#) []

6.3.1 Detailed Description

6.3.2 Macro Definition Documentation

6.3.2.1 RECORD_COUNT

```
#define RECORD_COUNT 17
```

The total count of records.

Definition at line 12 of file [Data.h](#).

6.3.2.2 ID_MAX_LENGTH

```
#define ID_MAX_LENGTH 9
```

The amount of memory (bytes) required to be allocated for the ID field. Equal to the longest name in `Data_IDs`, "mes08346", plus the null terminator

Definition at line 17 of file [Data.h](#).

6.3.2.3 NAME_MAX_LENGTH

```
#define NAME_MAX_LENGTH 21
```

The amount of memory (bytes) required to be allocated for the Name field. Equal to the longest name in `Data_↔Names`, "Assefa Ayalew Yoseph", plus the null terminator

Definition at line 22 of file [Data.h](#).

6.3.2.4 IP_LENGTH

```
#define IP_LENGTH 16
```

The amount of memory (bytes) required to be allocated for the IP field. Large enough to store '111.111.111.111' plus the null terminator.

Definition at line 28 of file [Data.h](#).

6.3.3 Variable Documentation

6.3.3.1 accepted_userIDs [1/2]

```
char* accepted_userIDs[ ]
```

Initial value:

```
= {
    "chen",
    "beal389",
    "bol4559",
    "cal6258",
    "kre5277",
    "lon1150",
    "mas9309",
    "mes08346",
    "mil7233",
    "nef9476",
    "nov7488",
    "pan9725",
    "rac3146",
    "rub4133",
    "shr5683",
    "vay3083",
    "yos2327"}
```

An array of the accepted userIDs.

Definition at line 7 of file [Data.c](#).

6.3.3.2 userFullNames [1/2]

```
char* userFullNames[ ]
```

Initial value:

```
= {
    "Weifeng Chen",
    "Christian Beatty",
    "Emily Bolles",
    "Cameron Calhoun",
    "Ty Kress",
    "Cody Long",
    "Caleb Massey",
    "Christian Messmer",
    "Karl Miller",
    "Jeremiah Neff",
    "Kaitlyn Novacek",
    "Joshua Panaro",
    "Caleb Rachocki",
    "Caleb Ruby",
    "Paul Shriner",
    "Alan Vayansky",
    "Assefa Ayalew Yoseph"}
```

An array of the full names, where the index of the name corresponds to the id in accepted_userIDs.

Definition at line 26 of file [Data.c](#).

6.3.3.3 accepted_userIDs [2/2]

```
char* accepted_userIDs[ ] [extern]
```

An array of the accepted userIDs.

Definition at line 7 of file [Data.c](#).

6.3.3.4 userFullNames [2/2]

```
char* userFullNames[ ] [extern]
```

An array of the full names, where the index of the name corresponds to the id in accepted_userIDs.

Definition at line 26 of file [Data.c](#).

6.4 Files

This module contains functions that interact with files.

Macros

- #define [LOCKFILE](#) "/tmp/lab5.lock"
- #define [REGISTERED_FILE](#) "registered.txt"
- #define [SERVER_SETTINGS_FILE](#) "server-settings.txt"
- #define [ADS_DIR](#) "ads"

Functions

- short [FileStatus](#) (char *filename)
- FILE * [CreateOrOpenFileVerbose](#) (char *filename, char *defaultContents)
- int [ReadRegisteredFileIntoUsersMap](#) (FILE *reg_file, map *users_map)
- void [UpdateRegisteredFileFromUsersMap](#) (FILE *reg_file, map *users_map)
Updates the registered file with of all users from user map that are marked as registered.
- int [NumberOfFilesInDirectory](#) (char *dir_name)
Finds the number of files/directories in a given directory.
- void [GetRandomFileNameFromDir](#) (char *dir_name, char *file_name)
Get the Random File Name From Dir object.
- int [ReadSettingsFileIntoSettingsMap](#) (FILE *settings_file, map *settings_map)
- void [CatFileToBuffer](#) (char *file_name, char *buffer, size_t buffer_size)
Concatinates the contents of file_name into the buffer string.

6.4.1 Detailed Description

6.4.2 Macro Definition Documentation

6.4.2.1 LOCKFILE

```
#define LOCKFILE "/tmp/lab5.lock"
```

The presence of a lockfile indicates that a server process is already running. The lockfile contains the process ID of the running process.

Definition at line 16 of file [File.h](#).

6.4.2.2 REGISTERED_FILE

```
#define REGISTERED_FILE "registered.txt"
```

This file contains a list of registered users and their data, with fields tab-delimited.

Note

- (1) The userID of the user.
- (2) The age of the user.
- (3) The GPA of the user.
- (4) The IP address of the user.
- (5) The last connection time of a user.

Definition at line 26 of file [File.h](#).

6.4.2.3 SERVER_SETTINGS_FILE

```
#define SERVER_SETTINGS_FILE "server-settings.txt"
```

Contains settings for the server. Each setting row contains a key, 0 or more space, an '=' symbol, and a value. Valid keys:

Note

- port; the port the server will listen on.
- send_buffer_size; the size of the send buffer
- receive_buffer_size; the size of the receive buffer
- backlog; the quantity of allowed backlogged unprocessed connections.

Definition at line 38 of file [File.h](#).

6.4.2.4 ADS_DIR

```
#define ADS_DIR "ads"
```

Contains files of ascii art to sent to clients.

Note

should be the relative directory to the file the ads are in

Definition at line 45 of file [File.h](#).

6.4.3 Function Documentation

6.4.3.1 FileStatus()

```
short FileStatus (
    char * filename )
```

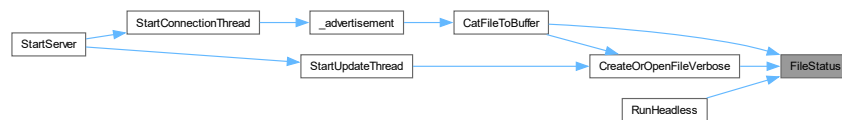
Determines if a file indicated by filename exists and is accesible by the user.

Returns

0 if the file does not exist. 1 if the file exists and the user has access. 2 if the file exists and the user does not have read and write permissions.

Definition at line 17 of file [File.c](#).

Here is the caller graph for this function:



6.4.3.2 CreateOrOpenFileVerbose()

```
FILE * CreateOrOpenFileVerbose (
    char * filename,
    char * defaultContents )
```

Will call fopen() on a file and put default data inside, or nothing if defaultContents is NULL. Will print the results of its attempt.

Warning

Does not close the file; returns the open file.

Note

Prints successes and errors.

Parameters

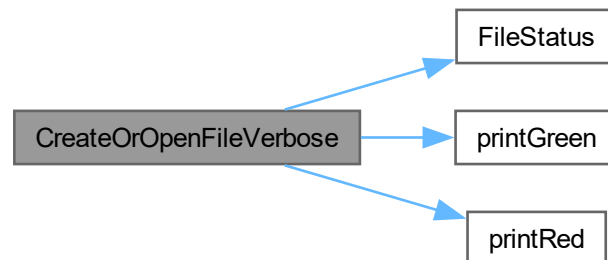
<i>filename</i>	The file name to create or open.
<i>defaultContents</i>	The contents to put in the file, if creating a default file, or NULL if no contents should be added.

Returns

The opened file, or NULL on failure.

Definition at line 29 of file [File.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**6.4.3.3 ReadRegisteredFileIntoUsersMap()**

```

int ReadRegisteredFileIntoUsersMap (
    FILE * reg_file,
    map * users_map )
  
```

Reads the registered file into the user's map, by checking the IDs in the first field and setting the data at that location.

Note

Prints warnings and errors.

Parameters

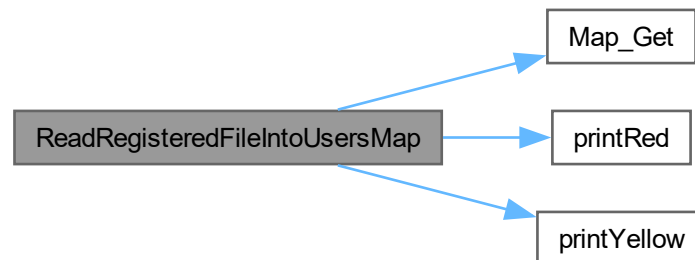
<i>reg_file</i>	The registered users file, open for reading.
<i>users_map</i>	The user's map to read into.

Returns

0 if success, error code if there was an error.

Definition at line 78 of file [File.c](#).

Here is the call graph for this function:



6.4.3.4 UpdateRegisteredFileFromUsersMap()

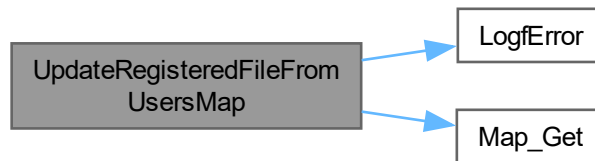
```
void UpdateRegisteredFileFromUsersMap (
    FILE * reg_file,
    map * users_map )
```

Parameters

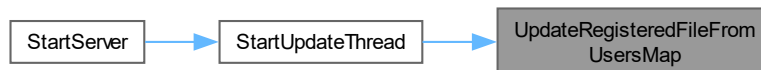
<i>reg_file</i>	file to update to
<i>users_map</i>	the map of users to use to update

Definition at line 111 of file [File.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.4.3.5 NumberOfFilesInDirectory()

```
int NumberOfFilesInDirectory (
    char * dir_name )
```

Parameters

<i>dir_name</i>	directory to count files from
-----------------	-------------------------------

Returns

int number of files in the directory

Definition at line 127 of file [File.c](#).

Here is the caller graph for this function:



6.4.3.6 GetRandomFileNameFromDir()

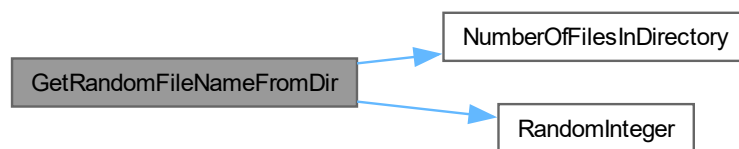
```
void GetRandomFileNameFromDir (
    char * dir_name,
    char * file_name )
```

Parameters

<i>dir_name</i>	name of the director to get a file name of
<i>file_name</i>	sets the name of the file into file_name

Definition at line 146 of file [File.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.4.3.7 ReadSettingsFileIntoSettingsMap()

```
int ReadSettingsFileIntoSettingsMap (
    FILE * settings_file,
    map * settings_map )
```

Reads the settings file into the settings map, by checking each line for a key value pair separated by a "=". It mallocs each key and value string it finds.

Note

Prints warnings and errors.

Parameters

<i>settings_file</i>	The settings file, open for reading.
<i>users_map</i>	The settings_map to read into.

Returns

0 if success, an error code if there was an error.

Definition at line 165 of file [File.c](#).

Here is the call graph for this function:

**6.4.3.8 CatFileToBuffer()**

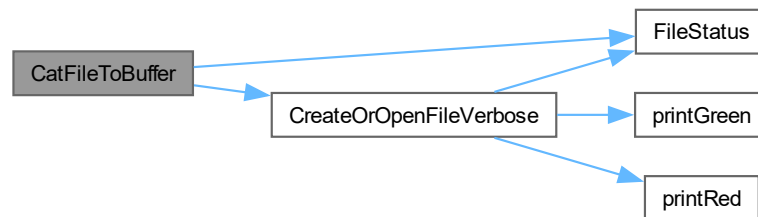
```
void CatFileToBuffer (
    char * file_name,
    char * buffer,
    size_t buffer_size )
```

Parameters

<i>file_name</i>	file to concatenate
<i>buffer</i>	string to copy it to
<i>buffer_size</i>	max size of buffer

Definition at line 190 of file [File.c](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.5 Log

Handles logging; can be to the console or a file.

Data Structures

- struct [LogSettings](#)

Macros

- `#define TRACE 0`
These define the levels that a log can be printed as.

Functions

- void [LogfFatal](#) (const char *format,...)
Logs at a FATAL level the formatted string and parameters of the string.
- void [LogfError](#) (const char *format,...)
Logs at an ERROR level the formatted string and parameters of the string.
- void [LogfWarning](#) (const char *format,...)
Logs at a WARNING level the formatted string and parameters of the string.
- void [LogfInfo](#) (const char *format,...)
Logs at a INFO level the formatted string and parameters of the string.
- void [LogfDebug](#) (const char *format,...)
Logs at a DEBUG level the formatted string and parameters of the string.
- void [LogfTrace](#) (const char *format,...)
Logs at a TRACE level the formatted string and parameters of the string.
- void [InitializeLogger](#) (FILE *_printStream, char printLevel, char logLevel, char printAllToStdOut)
Instantiates the logger settings to run off of, must be called before logging can occur.

6.5.1 Detailed Description

Log was initially going to also maintain a log.txt file that recorded user interactions with the server, but this feature was not finished. For now it just logs to the console.

6.5.2 Function Documentation

6.5.2.1 LogfFatal()

```
void LogfFatal (
    const char * format,
    ... )
```

Parameters

<i>format</i>	formatted string
...	what to put in the formatted string

Definition at line 28 of file [Log.c](#).

6.5.2.2 LogfError()

```
void LogfError (
    const char * format,
    ... )
```

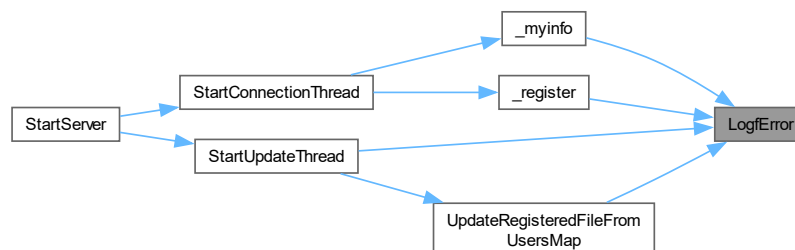
Is used

Parameters

<i>format</i>	formatted string
...	what to put in the formatted string

Definition at line 35 of file [Log.c](#).

Here is the caller graph for this function:



6.5.2.3 LogfWarning()

```
void LogfWarning (
    const char * format,
    ... )
```

Parameters

<i>format</i>	formatted string
...	what to put in the formatted string

Definition at line 42 of file [Log.c](#).

6.5.2.4 LogfInfo()

```
void LogfInfo (
    const char * format,
    ... )
```

Parameters

<i>format</i>	formatted string
...	what to put in the formatted string

Definition at line 49 of file [Log.c](#).

Here is the caller graph for this function:



6.5.2.5 LogfDebug()

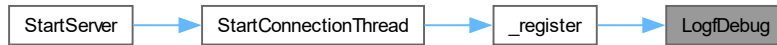
```
void LogfDebug (
    const char * format,
    ... )
```

Parameters

<i>format</i>	formatted string
...	what to put in the formatted string

Definition at line 56 of file [Log.c](#).

Here is the caller graph for this function:



6.5.2.6 LogfTrace()

```
void LogfTrace (
    const char * format,
    ... )
```

Parameters

<i>format</i>	formatted string
...	what to put in the formatted string

Definition at line 63 of file [Log.c](#).

6.5.2.7 InitializeLogger()

```
void InitializeLogger (
    FILE * _printStream,
    char printLevel,
    char logLevel,
    char printAllToStdOut )
```

Parameters

<i>_printStream</i>	output stream to print at
<i>outputLevel</i>	minimum level needed to print to the printstream
<i>logLevel</i>	minimum level needed to print to log file
<i>printAllToStdOut</i>	will print everything to stdout if not set to 0 as well as _printStream

Definition at line 9 of file [Log.c](#).

Here is the caller graph for this function:



6.6 Map

Functions that implement a hash map data structure.

Data Structures

- struct `_map_bucket`
map_bucket is an endpoint in the map. It is also a node in a linked list; if there were collisions, then the buckets are appended to the linked list at that location, then traversed until the matching key is found.
- struct `map`
A map. Stores key-value pairs for near constant lookup and insertion time.
- struct `map_result`
The result of a map retrieval.

Functions

- `map * NewMap` (int capacity)
- void `Map_Set` (`map *a_map`, char *key, void *value)
Sets a value in the map.
- `map_result Map_Get` (`map *a_map`, char *key)
Gets a value from the map. It will return a map_get_result describing whether it was succesful, and possibly containing the data sought, or NULL if it was unsuccessful.
- `map_result Map_Delete` (`map *a_map`, char *key, short free_it)
Deletes a key from the map. Returns a map_get_result describing whether the delete was succesful and containing the removed data, if extant.

6.6.1 Detailed Description

Karl's take on a simple hash map structure, which maps strings to void pointers. You can use casting to convert the void pointers into most of whatever else is needed.

Example usage, casting an int into the data part of the map.

```
int myfunc() {
    map *mymap = NewMap(100);
    Map_Set(mymap, "age", (void*)55);
    map_result result = Map_Get(mymap, "age");
    int age;
    if(result.found) {
        age = (int) map_result.data;
    }
}
```

Note, with this simple implementation, the map cannot change its capacity. A change to its capacity would change the hashing.

Ultimately there are really only three things you need to do with the map.

Initialize it, with some capacity larger than you will use. EG `map * mymap = NewMap(100)`. The bigger it is, the fewer collisions (which are pretty rare anyway).

Set some values in it. Eg `Map_Set(mymap, "key", &value)`;

You can cast numbers to void pointers to put them in the map, or you can use the pointers as references to, for example, strings malloced somewhere.

Get some values from it. Eg `void* myval = Map_Get(mymap, "key")`;

Delete some values from it. For example `Map_Delete(mymap, "key", 0)`;

Note that the last parameter, 'free it', tells the map whether it should call 'free' on the underlying data in memory. If this is 1, and the underlying data is not a reference to a malloced part of the heap, errors will result.

6.6.2 Function Documentation

6.6.2.1 NewMap()

```
map * NewMap (
    int capacity )
```

Creates a new map. The map capacity will be a power of 2 that is large enough to contain the estimated size.

Parameters

<i>capacity</i>	The estimated required capacity of the map.
-----------------	---

Returns

A pointer to the heap allocated map.

Definition at line 49 of file [map.c](#).

Here is the caller graph for this function:



6.6.2.2 Map_Set()

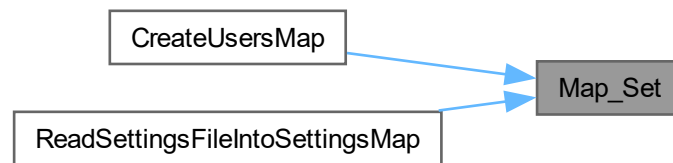
```
void Map_Set (
    map * a_map,
    char * key,
    void * value )
```

Parameters

<i>map</i>	The map to set a key in.
<i>key</i>	The key to use.
<i>keylen</i>	The length of the key.
<i>value</i>	The pointer to the data stored at that location.

Definition at line 89 of file [map.c](#).

Here is the caller graph for this function:



6.6.2.3 Map_Get()

```
map_result Map_Get (
    map * a_map,
    char * key )
```

Parameters

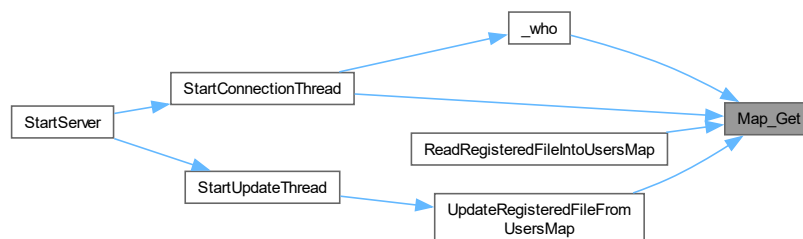
<i>map</i>	The map to retrieve from.
<i>key</i>	The key of the item.

Returns

A map_get_result containing the sought data.

Definition at line 119 of file [map.c](#).

Here is the caller graph for this function:



6.6.2.4 Map_Delete()

```
map_result Map_Delete (
    map * a_map,
    char * key,
    short free_it )
```

Parameters

<i>map</i>	The map to delete the key from.
<i>key</i>	The key to delete.
<i>free↔ _it</i>	Whether to call free() on the underlying data.

Returns

A `map_get_result` with the data that was removed.

Definition at line 154 of file [map.c](#).

6.7 Server

Functions for running the server.

Data Structures

- struct [ServerProperties](#)

Functions

- int [StartServer](#) ([map](#) *[users_map](#))
- [Connection](#) * [NextAvailableConnection](#) ()
- int [InitializeServer](#) ()

6.7.1 Detailed Description

6.7.2 Function Documentation

6.7.2.1 StartServer()

```
int StartServer (  
    map * users\_map )
```

Starts the server.

Note

This is a blocking call that will start a loop until SIGINT is received.

Parameters

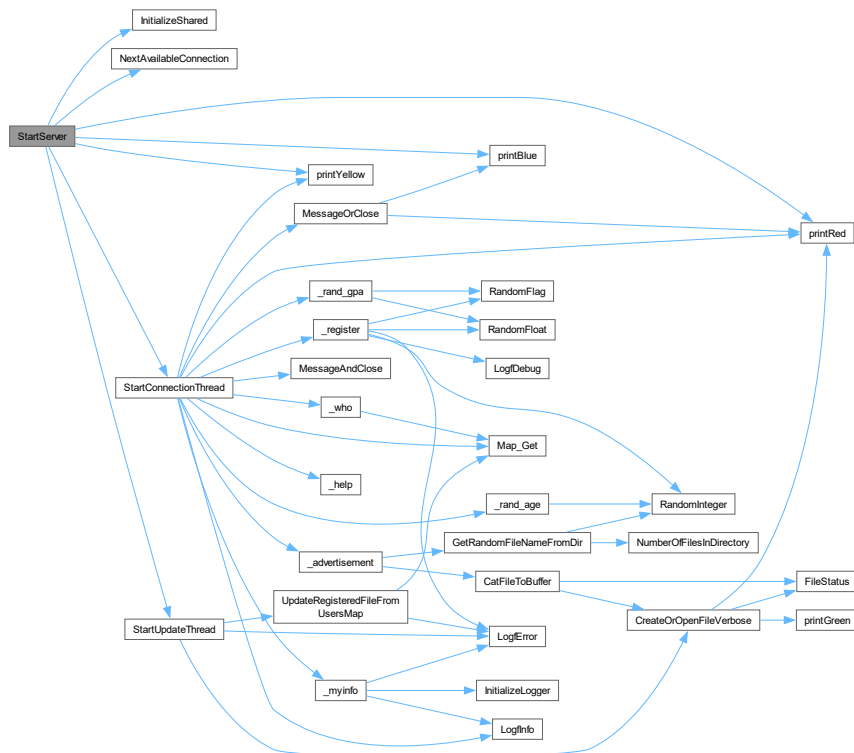
<i>users_map</i>	The user's map.
------------------	-----------------

Returns

1 if the server ran and shutdown gracefully, 0 if there was an error during setup.

Definition at line 103 of file Server.c.

Here is the call graph for this function:



6.7.2.2 NextAvailableConnection()

```
Connection * NextAvailableConnection ( )
```

Iterates through the Connections array until it finds one whose 'active' field is false and returns it. If it iterates through the array and fails to find a connection, it returns NULL.

Returns

A **Connection** struct or null.

Definition at line 156 of file Server.c.

Here is the caller graph for this function:



6.7.2.3 InitializeServer()

```
int InitializeServer ( )
```

Initializes the server properties structure and the structures for holding [Connection](#) objects.

Note

Prints initialization status.

Returns

1 if it was able to initialize, otherwise 0.

6.8 Util

Utility functions used by various modules but not dependent on any other modules.

Macros

- `#define COLOR_RED "\e[38;2;255;75;75m"`
- `#define COLOR_GREEN "\e[38;2;0;240;0m"`
- `#define COLOR_YELLOW "\e[38;2;255;255;0m"`
- `#define COLOR_BLUE "\e[38;2;0;240;240m"`
- `#define COLOR_RESET "\e[0m"`

Functions

- void [printRed](#) (const char *format,...)
- void [printGreen](#) (const char *format,...)
- void [printYellow](#) (const char *format,...)
- void [printBlue](#) (const char *format,...)
- int [RandomInteger](#) (int min, int max)
- float [RandomFloat](#) (float min, float max)
- short [RandomFlag](#) (float percentage_chance)

6.8.1 Detailed Description

6.8.2 Macro Definition Documentation

6.8.2.1 COLOR_RED

```
#define COLOR_RED "\e[38;2;255;75;75m"
```

A virtual terminal escape sequence to print foreground red.

Definition at line 10 of file [Util.h](#).

6.8.2.2 COLOR_GREEN

```
#define COLOR_GREEN "\e[38;2;0;240;0m"
```

A VTE for green.

Definition at line 12 of file [Util.h](#).

6.8.2.3 COLOR_YELLOW

```
#define COLOR_YELLOW "\e[38;2;255;255;0m"
```

A VTE for yellow.

Definition at line 14 of file [Util.h](#).

6.8.2.4 COLOR_BLUE

```
#define COLOR_BLUE "\e[38;2;0;240;240m"
```

A VTE for blue.

Definition at line 16 of file [Util.h](#).

6.8.2.5 COLOR_RESET

```
#define COLOR_RESET "\e[0m"
```

A VTE to reset the printing color.

Definition at line 18 of file [Util.h](#).

6.8.3 Function Documentation

6.8.3.1 printRed()

```
void printRed (  
    const char * format,  
    ... )
```

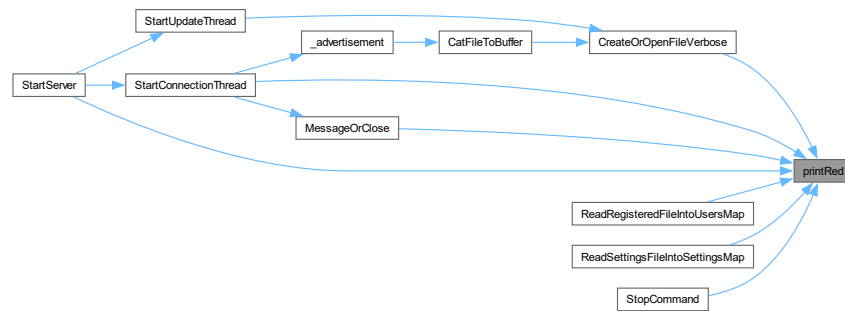
Prints to the console in red.

Parameters

<i>format</i>	A format, as printf.
...	args, as printf.

Definition at line 11 of file [Util.c](#).

Here is the caller graph for this function:



6.8.3.2 printGreen()

```
void printGreen (
    const char * format,
    ... )
```

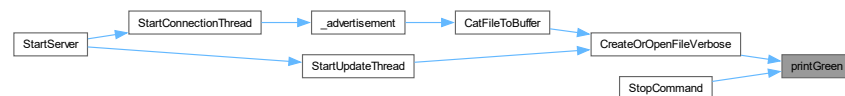
Prints to the console in green.

Parameters

<i>format</i>	A format, as printf.
...	args, as printf.

Definition at line 20 of file [Util.c](#).

Here is the caller graph for this function:



6.8.3.3 printYellow()

```
void printYellow (
    const char * format,
    ... )
```

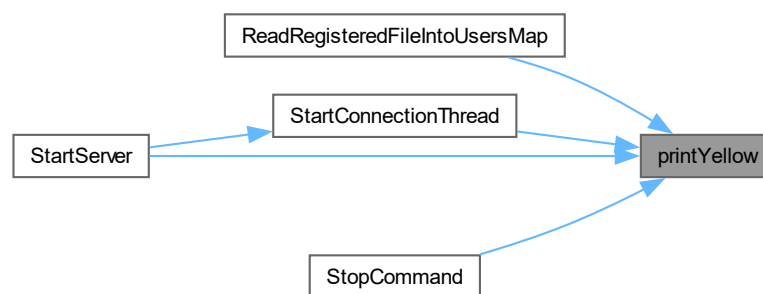
Prints to the console in yellow.

Parameters

<i>format</i>	A format, as printf.
...	args, as printf.

Definition at line 29 of file [Util.c](#).

Here is the caller graph for this function:

6.8.3.4 `printBlue()`

```

void printBlue (
    const char * format,
    ... )
  
```

Prints to the console in blue.

Parameters

<i>format</i>	A format, as printf.
...	args, as printf.

Definition at line 38 of file [Util.c](#).

Here is the caller graph for this function:



6.8.3.5 RandomInteger()

```
int RandomInteger (
    int min,
    int max )
```

Returns an integer between min and max.

Parameters

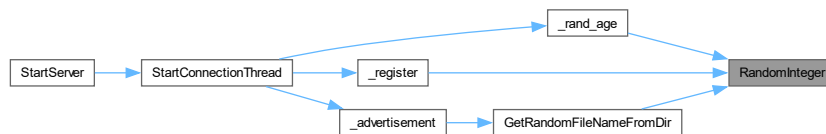
<i>min</i>	The minimum, inclusive.
<i>max</i>	The maximum, inclusive.

Returns

A random integer between min and max.

Definition at line 47 of file [Util.c](#).

Here is the caller graph for this function:



6.8.3.6 RandomFloat()

```
float RandomFloat (
    float min,
    float max )
```

Returns a float between min and max.

Parameters

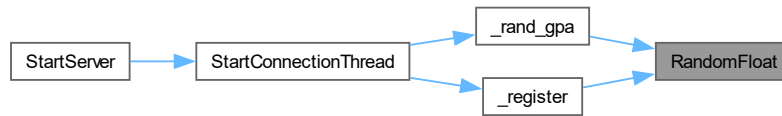
<i>min</i>	The minimum, inclusive.
<i>max</i>	The maximum, inclusive.

Returns

A random integer between min and max.

Definition at line 53 of file [Util.c](#).

Here is the caller graph for this function:



6.8.3.7 RandomFlag()

```
short RandomFlag (
    float percentage_chance )
```

Returns 1, percentage_chance of the time.

Parameters

<code>percentage_chance</code>	The chance to return 1.
--------------------------------	-------------------------

Note

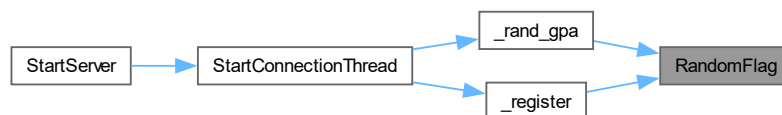
If `percentage_chance > 1`, this will always return true.

Returns

1 or 0

Definition at line 60 of file [Util.c](#).

Here is the caller graph for this function:



6.9 Process

Functions

- void [RunHeadless](#) (char *processName)
- void **StopCommand** ()

Stops the server that is running headlessly and prints the results of running the command.

Variables

- [User](#) * [users_array](#)
- [map](#) * [users_map](#)
- [map](#) * [settings_map](#)
- char * [default_settings](#)
- int [active_clients](#)

6.9.1 Detailed Description

6.9.2 Function Documentation

6.9.2.1 RunHeadless()

```
void RunHeadless (
    char * processName )
```

Uses `nohup ./{processName} run` to run the process headlessly.

Parameters

<i>processName</i>	The name of the currently running process, by default, 'server'.
--------------------	--

Definition at line 190 of file [Process.c](#).

Here is the call graph for this function:



6.9.3 Variable Documentation

6.9.3.1 users_array

```
User* users_array
```

The array of users. This will be populated on initialize by functions in Build.

Definition at line 20 of file [Process.c](#).

6.9.3.2 users_map

```
map* users_map
```

The map of userIDs to users. Populated on Initialize by functions in Build.

Definition at line 22 of file [Process.c](#).

6.9.3.3 settings_map

```
map* settings_map
```

The map of settings stored in the server settings file.

Definition at line 24 of file [Process.c](#).

6.9.3.4 default_settings

```
char* default_settings
```

Initial value:

```
= "port                = 3000\n"
   "send_buffer_size    = 1024\n"
   "receive_buffer_size = 1024\n"
   "backlog              = 10\n"
   "max_connections     = 20\n"
   "log_file             = log.txt\n"
   "log_level            = 1\n"
   "log_to_console       = true"
```

The default contents of the settings file, if it doesn't exist.

Definition at line 26 of file [Process.c](#).

6.9.3.5 active_clients

```
int active_clients
```

The number of active clients.

Definition at line 36 of file [Process.c](#).

Chapter 7

Data Structure Documentation

7.1 `_map_bucket` Struct Reference

`map_bucket` is an endpoint in the map. It is also a node in a linked list; if there were collisions, then the buckets are appended to the linked list at that location, then traversed until the matching key is found.

7.1.1 Detailed Description

Definition at line 81 of file [map.h](#).

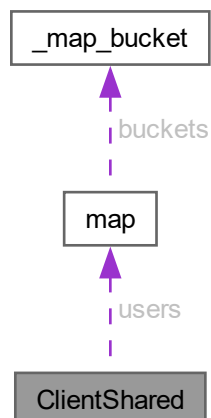
The documentation for this struct was generated from the following file:

- `src/server/map.h`

7.2 `ClientShared` Struct Reference

```
#include <Connection.h>
```

Collaboration diagram for `ClientShared`:



Data Fields

- `map * users`
- `pthread_mutex_t mutex`
- `short dirty`
- `short shutting_down`
- `size_t send_buffer_size`
- `size_t receive_buffer_size`

7.2.1 Detailed Description

Shared between the Connections and the Server.

Definition at line 17 of file [Connection.h](#).

7.2.2 Field Documentation

7.2.2.1 users

```
map* users
```

The user's map.

Definition at line 19 of file [Connection.h](#).

7.2.2.2 mutex

```
pthread_mutex_t mutex
```

A mutex to provide mutual-exclusion to connection threads operating on the user's map.

Definition at line 21 of file [Connection.h](#).

7.2.2.3 dirty

```
short dirty
```

Whether there were changes to the user's map that need to be saved in a file.

Definition at line 23 of file [Connection.h](#).

7.2.2.4 shutting_down

```
short shutting_down
```

Whether the server is shutting down.

Definition at line 25 of file [Connection.h](#).

7.2.2.5 send_buffer_size

```
size_t send_buffer_size
```

Passed along from server settings at the time shared is initialized

Definition at line 28 of file [Connection.h](#).

7.2.2.6 receive_buffer_size

```
size_t receive_buffer_size
```

Passed along from server settings at the time shared is initialized

Definition at line 30 of file [Connection.h](#).

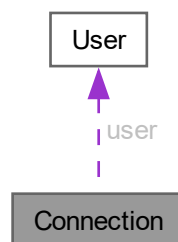
The documentation for this struct was generated from the following file:

- [src/server/Connection.h](#)

7.3 Connection Struct Reference

```
#include <Connection.h>
```

Collaboration diagram for Connection:



Data Fields

- ConnectionState [status](#)
- int [socket](#)
- struct sockaddr_in [address](#)
- socklen_t [address_length](#)
- pthread_t [thread_id](#)
- time_t [time_connected](#)
- ClientState [state](#)
- [User](#) * [user](#)

7.3.1 Detailed Description

Data for a single client socket connection to the server. Passed into the thread runner as the parameter.

Definition at line 46 of file [Connection.h](#).

7.3.2 Field Documentation

7.3.2.1 status

```
ConnectionState status
```

Whether this connection is closed (0) or active (1) or closing (2). This is set by the SERVER just prior to starting the thread. The thread sets it back to 0 when it is completely done.

Definition at line 48 of file [Connection.h](#).

7.3.2.2 socket

```
int socket
```

The underlying socket file descriptor.

Definition at line 50 of file [Connection.h](#).

7.3.2.3 address

```
struct sockaddr_in address
```

The socket address of the connection.

Definition at line 52 of file [Connection.h](#).

7.3.2.4 address_length

```
socklen_t address_length
```

The actual size of the client address; send by accept.

Definition at line 54 of file [Connection.h](#).

7.3.2.5 thread_id

```
pthread_t thread_id
```

The pthread ID of this client thread.

Definition at line 56 of file [Connection.h](#).

7.3.2.6 time_connected

```
time_t time_connected
```

When the client connected.

Definition at line 58 of file [Connection.h](#).

7.3.2.7 state

```
ClientState state
```

The client state.

Definition at line 60 of file [Connection.h](#).

7.3.2.8 user

```
User* user
```

The user associated with this client.

Definition at line 62 of file [Connection.h](#).

The documentation for this struct was generated from the following file:

- [src/server/Connection.h](#)

7.4 LogSettings Struct Reference

Data Fields

- [FILE](#) * **ostream**
output the logger should go to
- char **printLevel**
minimum level that the log should be in order to print stdout
- char **logLevel**
minimum level that the log should be for it to be printed to log file
- char **printAllToStdOut**
if not 0 will print to stdout as well as ostream

7.4.1 Detailed Description

Definition at line 27 of file [Log.h](#).

The documentation for this struct was generated from the following file:

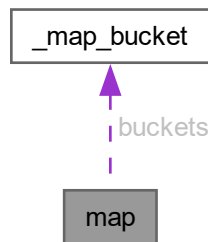
- [src/server/Log.h](#)

7.5 map Struct Reference

A map. Stores key-value pairs for near constant lookup and insertion time.

```
#include <map.h>
```

Collaboration diagram for map:



7.5.1 Detailed Description

Note

- Use [NewMap\(\)](#) to create a new map.
- Use [Map_Set\(\)](#) to set a key in the map.
- Use [Map_Get\(\)](#) to get a value from the map.

The values stored are of type void pointer.

Definition at line 101 of file [map.h](#).

The documentation for this struct was generated from the following file:

- [src/server/map.h](#)

7.6 map_result Struct Reference

The result of a map retrieval.

```
#include <map.h>
```

7.6.1 Detailed Description

Definition at line 111 of file [map.h](#).

The documentation for this struct was generated from the following file:

- [src/server/map.h](#)

7.7 ServerProperties Struct Reference

```
#include <Server.h>
```

Data Fields

- `uint16_t` [port](#)
- `size_t` [send_buffer_size](#)
- `size_t` [receive_buffer_size](#)
- `int` [backlog](#)
- `int` [active_connections](#)
- `int` [max_connections](#)
- `time_t` [time_started](#)

7.7.1 Detailed Description

Defines the properties for the server.

Defined in `server-settings.txt`, a configuration file.

Definition at line 20 of file [Server.h](#).

7.7.2 Field Documentation

7.7.2.1 port

```
uint16_t port
```

The port the server will connect on.

Definition at line 22 of file [Server.h](#).

7.7.2.2 send_buffer_size

```
size_t send_buffer_size
```

The size of each send buffer.

Definition at line 24 of file [Server.h](#).

7.7.2.3 receive_buffer_size

```
size_t receive_buffer_size
```

The size of each receive buffer.

Definition at line 26 of file [Server.h](#).

7.7.2.4 backlog

```
int backlog
```

The size of the backlog of unprocessed connections.

Definition at line 28 of file [Server.h](#).

7.7.2.5 active_connections

```
int active_connections
```

The number of active connections.

Definition at line 30 of file [Server.h](#).

7.7.2.6 max_connections

```
int max_connections
```

The maximum number of active connections the server supports.

Definition at line 32 of file [Server.h](#).

7.7.2.7 time_started

```
time_t time_started
```

The time the server was started.

Definition at line 34 of file [Server.h](#).

The documentation for this struct was generated from the following file:

- [src/server/Server.h](#)

7.8 User Struct Reference

```
#include <Data.h>
```

Data Fields

- char **id** [[ID_MAX_LENGTH](#)]
The user ID; equal to an element in `accepted_userIDs`.
- char **name** [[NAME_MAX_LENGTH](#)]
The user's real name; equal to an element in `userFullNames`.
- int **age**
The user's age, randomized between 18 and 22.
- float **gpa**
The user's gpa, randomized between 2.5 and 4.0.
- short **connected**
Whether the user is connected.
- char **ip** [[IP_LENGTH](#)]
The last IP used by the user; set on connection.
- long **lastConnection**
A unix timestamp representing the last time a user connected.
- short **registered**
Whether user has executed the 'register' command.

7.8.1 Detailed Description

A [User](#) of this server. The ID and Name fields are populated initially. GPA and age are populated at the time a user is registered, and saved and loaded from a file. Active is set and unset when a user connects. IP is set each time a user connects, and saved in the file.

Definition at line [44](#) of file [Data.h](#).

The documentation for this struct was generated from the following file:

- `src/server/Data.h`

Chapter 8

File Documentation

8.1 Build.c

```
00001
00005 #include <stdlib.h>
00006 #include <string.h>
00007 #include "Build.h"
00008
00009 User * CreateUsersArray(char ** userIDs, char ** userNames, int recordsCount)
00010 {
00011     size_t uarr_size = sizeof(User) * recordsCount;
00012     User * uarr = malloc(uarr_size);
00013     memset(uarr, 0, uarr_size);
00014     int i;
00015     for(i = 0; i < recordsCount; i++)
00016     {
00017         strcpy(uarr[i].id, userIDs[i]);
00018         strcpy(uarr[i].name, userNames[i]);
00019     }
00020     return uarr;
00021 }
00022
00023 map * CreateUsersMap(User * usersArray, int recordsCount)
00024 {
00025     map * umap = NewMap(recordsCount * 3);
00026     int i;
00027     for(i = 0; i < recordsCount; i++) {
00028         Map_Set(umap, usersArray[i].id, &usersArray[i]);
00029     }
00030     return umap;
00031 }
00032
00033
```

8.2 Build.h

```
00001 #ifndef Build_h
00002 #define Build_h
00008 #include "Data.h"
00009 #include "map.h"
00010
00018 User * CreateUsersArray(char ** userIDs, char ** userNames, int recordsCount);
00019
00027 map * CreateUsersMap(User * usersArray, int recordsCount);
00028
00029
00030
00034 #endif
```

8.3 Connection.c

```
00001
```

```

00005 #include "Connection.h"
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008 #include <strings.h>
00009 #include <string.h>
00010 #include <arpa/inet.h>
00011 #include <unistd.h>
00012 #include "Util.h"
00013 #include "Log.h"
00014 #include "Data.h"
00015 #include "File.h"
00016
00017 ClientShared shared;
00018
00019 ClientShared * InitializeShared(map * users_map, size_t send_buffer_size, size_t receive_buffer_size)
00020 {
00021     shared.users = users_map;
00022     shared.dirty = 0;
00023     shared.shutting_down = 0;
00024     shared.send_buffer_size = send_buffer_size;
00025     shared.receive_buffer_size = receive_buffer_size;
00026     pthread_mutex_init(&(shared.mutex), NULL);
00027     return &shared;
00028 }
00029
00030 void * StartUpdateThread(void * parameter)
00031 {
00032     while(shared.shutting_down == 0) {
00033         if(shared.dirty) {
00034             pthread_mutex_lock(&(shared.mutex));
00035             shared.dirty = 0;
00036             FILE * reg_file = CreateOrOpenFileVerbose(REGISTERED_FILE, NULL);
00037             if(reg_file != NULL) {
00038                 UpdateRegisteredFileFromUsersMap(reg_file, shared.users);
00039                 fclose(reg_file);
00040             } else {
00041                 LogfError("FAILED TO OPEN REGISTERED FILE - NO DATA WILL BE UPDATED");
00042                 shared.dirty = 1;
00043             }
00044             pthread_mutex_unlock(&(shared.mutex));
00045         }
00046         sleep(1);
00047     }
00048     return NULL;
00049 }
00050
00051
00052 void * StartConnectionThread(void * p_connection)
00053 {
00054     Connection * connection = (Connection *) p_connection;
00055     connection->state = ClientState_ENTRY;
00056     connection->user = NULL;
00057     time(&(connection->time_connected));
00058     // allocate send and receive buffers.
00059     char * send_buffer = malloc(shared.send_buffer_size);
00060     char * receive_buffer = malloc(shared.receive_buffer_size);
00061     //int bytes_received;
00062     map_result result;
00063
00064     // ask for their user ID initially, or disconnect them.
00065     strcpy(send_buffer, "<Message>Welcome. Please send your user ID.");
00066     MessageOrClose(send_buffer, receive_buffer, connection);
00067     if(connection->status == ConnectionStatus_ACTIVE) {
00068         result = Map_Get(shared.users, receive_buffer);
00069         if(!result.found)
00070         {
00071             printYellow("Unauthorized access attempt by %s with name '%s'.\n",
inet_ntoa(connection->address.sin_addr), receive_buffer);
00072             strcpy(send_buffer, "<Error>No such user");
00073             MessageAndClose(send_buffer, connection);
00074             // send a one-way message to the client
00075         } else {
00076             User * user = (User *) result.data;
00077             if(user->connected) {
00078                 printYellow("User %s attempted to double connect from IP %s.\n", user->id,
inet_ntoa(connection->address.sin_addr));
00079                 strcpy(send_buffer, "<Error>You are already connected.");
00080                 MessageAndClose(send_buffer, connection);
00081                 // send the other connected user an informative message?
00082             } else {
00083                 connection->user = user;
00084                 connection->user->connected = 1;
00085                 strcpy(connection->user->ip, inet_ntoa(connection->address.sin_addr));
00086                 if(connection->user->registered) {
00087                     connection->state = ClientState_REGISTERED;
00088                 } else {
00089                     connection->state = ClientState_ACCESSING;

```

```

00090     }
00091     }
00092     }
00093 }
00094
00095 if(connection->state == ClientState_ACCESSING && connection->status == ConnectionStatus_ACTIVE) {
00096     strcpy(send_buffer, "<Message>Say something, unregistered user!");
00097 } else if (connection->state == ClientState_REGISTERED && connection->status ==
ConnectionStatus_ACTIVE) {
00098     strcpy(send_buffer, "<Message>Say something, registered user!");
00099 }
00100
00101 while(connection->status == ConnectionStatus_ACTIVE)
00102 {
00103     if(connection->state == ClientState_ACCESSING) {
00104         MessageOrClose(send_buffer, receive_buffer, connection);
00105         if (strcmp(receive_buffer, "help") == 0) {
00106             _help(connection, send_buffer);
00107         } else if (strcmp(receive_buffer, "exit") == 0) {
00108             strcpy(send_buffer, "<Message>Goodbye.");
00109             MessageAndClose(send_buffer, connection);
00110         } else if (strcmp(receive_buffer, "register") == 0) {
00111             _register(connection, send_buffer);
00112         } else {
00113             strcpy (send_buffer, "<Error>Invalid command, use 'help' for list of commands");
00114         }
00115     } else if(connection->state == ClientState_REGISTERED) {
00116         MessageOrClose(send_buffer, receive_buffer, connection);
00117         if (strcmp(receive_buffer, "help") == 0) {
00118             _help(connection, send_buffer);
00119         } else if (strcmp(receive_buffer, "exit") == 0) {
00120             strcpy(send_buffer, "<Message>Goodbye.");
00121             MessageAndClose(send_buffer, connection);
00122         } else if (strcmp(receive_buffer, "myinfo") == 0) {
00123             _myinfo(connection, send_buffer);
00124         } else if (strcmp(receive_buffer, "who") == 0) {
00125             _who(send_buffer);
00126         } else if (strcmp(receive_buffer, "random-gpa") == 0) {
00127             _rand_gpa(connection, send_buffer);
00128         } else if (strcmp(receive_buffer, "random-age") == 0) {
00129             _rand_age(connection, send_buffer);
00130         } else if (strcmp(receive_buffer, "advertisement") == 0) {
00131             _advertisement(connection, send_buffer);
00132         } else {
00133             strcpy(send_buffer, "<Error>Invalid command, use 'help' for list of commands");
00134         }
00135         // call a function for processing this state.
00136     } else {
00137         printRed("Client entered invalid state. Disconnecting. \n");
00138         strcpy(send_buffer, "<Error>You entered an invalid state!");
00139         MessageAndClose(send_buffer, connection);
00140         connection->status = ConnectionStatus_CLOSING;
00141     }
00142 }
00143
00144 if(connection->user != NULL) {
00145     connection->user->connected = 0;
00146     LogfInfo("User %s from ip %s disconnected.\n", connection->user->id, connection->user->ip);
00147 } else {
00148     LogfInfo("Ip %s disconnected.\n", inet_ntoa(connection->address.sin_addr));
00149 }
00150
00151 free(send_buffer);
00152 free(receive_buffer);
00153 close(connection->socket);
00154 if(connection->user != NULL) {
00155     connection->user->connected = 0;
00156 }
00157 connection->status = ConnectionStatus_CLOSED;
00158 return NULL;
00159 }
00160
00161
00162
00163 int MessageOrClose(char * send_buffer, char * receive_buffer, Connection * connection) {
00164     receive_buffer[0] = '\0';
00165     //memset(receive_buffer, 0, shared.receive_buffer_size);
00166     if(send(connection->socket, send_buffer, shared.send_buffer_size, 0) < 0) {
00167         printRed("Failed to send message to %s. Disconnecting.\n",
inet_ntoa(connection->address.sin_addr));
00168         perror("Error:");
00169         connection->status = ConnectionStatus_CLOSING;
00170         return 0;
00171     }
00172     int received_size = recv(connection->socket, receive_buffer, shared.receive_buffer_size, 0);
00173     if(received_size < 0) {
00174         printRed("Failed to receive message from %s. Disconnecting.\n",

```

```

    inet_ntoa(connection->address.sin_addr));
00175     perror("Error: ");
00176     connection->status = ConnectionStatus_CLOSING;
00177     return 0;
00178 }
00179 if(received_size == 0 ) {
00180     printBlue("%s disconnected.\n", inet_ntoa(connection->address.sin_addr));
00181     connection->status = ConnectionStatus_CLOSING;
00182     return 0;
00183 }
00184 send_buffer[0] = '\0';
00185 // memset(send_buffer, 0, shared.send_buffer_size);
00186 return received_size;
00187 }
00188
00189
00190
00191 void MessageAndClose(char * send_buffer, Connection * connection) {
00192     strcat(send_buffer, "<Disconnect>");
00193     send(connection->socket, send_buffer, shared.send_buffer_size, 0);
00194     connection->status = ConnectionStatus_CLOSING;
00195     if(connection -> user != NULL) {
00196         connection->user->connected = 0;
00197     }
00198 }
00199
00200 void _help(Connection* connection, char* response) {
00201     if(connection->state != ClientState_REGISTERED) {
00202         strcpy(response, "<Message>help - get a list of available commands\n");
00203         strcat(response, "register - register your user\n");
00204         strcat(response, "exit - disconnect from the server\n");
00205     } else if(connection->state == ClientState_REGISTERED) {
00206         strcpy(response, "<Message>help- get a list of available commands\n");
00207         strcat(response, "exit - disconnect from the server\n");
00208         strcat(response, "who - get a list of online users\n");
00209         strcat(response, "random-gpa - set your gpa to a new random value\n");
00210         strcat(response, "random-age - set your age to a new random value\n");
00211         strcat(response, "advertisement - get a colorful advertisement\n");
00212         strcat(response, "myinfo - get info about yourself");
00213     }
00214 }
00215
00216 int _register(Connection * connection, char* response) {
00217     if(connection->user->registered) {
00218         strcpy(response, "<Error>");
00219         strcat(response, connection->user->id);
00220         strcat(response, " is already registered.");
00221     }
00222     LogfError("%s from ip %s has attempted to register a second time.\n", connection->user->id,
00223     inet_ntoa(connection->address.sin_addr));
00224     return 0;
00225 }
00226
00227 pthread_mutex_lock(&(shared.mutex));
00228
00229 connection->user->registered = 1;
00230
00231 connection->user->age = RandomInteger(18, 22);
00232
00233 if(RandomFlag(.4)) {
00234     connection->user->gpa = 4.0;
00235 } else {
00236     connection->user->gpa = RandomFloat(2.5, 4);
00237 }
00238
00239 connection->state = ClientState_REGISTERED;
00240
00241 LogfDebug("%s has been registered.\n", connection->user->id);
00242
00243 shared.dirty = 1;
00244 pthread_mutex_unlock(&(shared.mutex));
00245
00246 strcpy(response, "<Message>You Have been registered ");
00247 strcat(response, connection->user->name);
00248
00249 return 1;
00250 }
00251
00252 int _myinfo(Connection* connection, char* response) {
00253     InitializeLogger(stdout, 0, 0, 0);
00254
00255     if (!(connection->user->registered)) {
00256         strcpy(response, "<Error>");
00257         strcat(response, connection->user->id);
00258         strcat(response, " is not registered.");
00259     }

```



```

00260         LogfError("%s from ip %s has attempted to view their information as an unregistered user.\n",
connection->user->id, inet_ntoa(connection->address.sin_addr));
00261
00262         return 1;
00263     }
00264
00265     //Referenced snprintf from https://cplusplus.com/reference/cstdio/snprintf/
00266     snprintf(response, shared.send_buffer_size, "<User.Name>%s<User.Age>%d<User.GPA>%.2f<User.IP>%s",
connection->user->name, connection->user->age, connection->user->gpa,
inet_ntoa(connection->address.sin_addr));
00267
00268     LogfInfo("%s viewed their information.\n", connection->user->id);
00269
00270     return 0;
00271 }
00272
00273 void _who(char * response) {
00274     int i;
00275     for(i = 0; i < RECORD_COUNT; i++) {
00276         map_result result = Map_Get(shared.users, accepted_userIDs[i]);
00277         if(result.found) {
00278             User* user = (User *) result.data;
00279
00280             if(user->connected) {
00281                 strcat(response, "<OnlineUser>");
00282                 strcat(response, user->id);
00283             }
00284         }
00285     }
00286 }
00287
00288 void _rand_gpa(Connection* connection, char* response) {
00289     char gpa_str[5];
00290     pthread_mutex_lock(&(shared.mutex));
00291     if(RandomFlag(.4)) {
00292         connection->user->gpa = 4.0;
00293     } else {
00294         connection->user->gpa = RandomFloat(2.2, 4.0);
00295     }
00296     shared.dirty = 1;
00297     pthread_mutex_unlock(&(shared.mutex));
00298     sprintf(gpa_str, "%.2f", connection->user->gpa);
00299     strcat(response, "<User.GPA>");
00300     strcat(response, gpa_str);
00301 }
00302
00303 void _rand_age(Connection* connection, char * response) {
00304     char age_str[5];
00305     pthread_mutex_lock(&(shared.mutex));
00306     connection->user->age = RandomInteger(18, 22);
00307     shared.dirty = 1;
00308     pthread_mutex_unlock(&(shared.mutex));
00309
00310     sprintf(age_str, "%d", connection->user->age);
00311     strcat(response, "<User.Age>");
00312     strcat(response, age_str);
00313 }
00314
00315 void _advertisement(Connection * connection, char * response) {
00316     char filename[FILENAME_MAX];
00317
00318     GetRandomFileNameFromDir(ADS_DIR, filename);
00319
00320     char* filepath = malloc(FILENAME_MAX + sizeof(ADS_DIR));
00321     strcpy(filepath, ADS_DIR);
00322     strcat(filepath, "/");
00323     strcat(filepath, filename);
00324
00325     strcat(response, "<Message>");
00326     CatFileToBuffer(filepath, response, shared.send_buffer_size);
00327 }

```

8.4 Connection.h

```

00001 #ifndef Connection_h
00002 #define Connection_h
00008 #include <netinet/in.h>
00009 #include <pthread.h>
00010 #include <time.h>
00011 #include <Data.h>
00012 #include "map.h"
00013
00017 typedef struct {

```

```

00019     map * users;
00021     pthread_mutex_t mutex;
00023     short dirty;
00025     short shutting_down;
00026
00028     size_t send_buffer_size;
00030     size_t receive_buffer_size;
00031 } ClientShared;
00032
00033 #define ClientState_ENTRY 1
00034 #define ClientState_ACCESSING 2
00035 #define ClientState_REGISTERED 3
00036 typedef short ClientState;
00037
00038 #define ConnectionStatus_CLOSED 0
00039 #define ConnectionStatus_ACTIVE 1
00040 #define ConnectionStatus_CLOSING 2
00041 typedef short ConnectionState;
00042
00046 typedef struct {
00048     ConnectionState status;
00050     int socket;
00052     struct sockaddr_in address;
00054     socklen_t address_length;
00056     pthread_t thread_id;
00058     time_t time_connected;
00060     ClientState state;
00062     User * user;
00063
00064 } Connection;
00065
00071 ClientShared * InitializeShared(map * users_map, size_t send_buffer_size, size_t receive_buffer_size);
00072
00077 void * StartConnectionThread(void * connection);
00078
00088 int MessageOrClose(char * send_buffer, char * receive_buffer, Connection * connection);
00089
00095 void MessageAndClose(char * send_buffer, Connection * connection);
00096
00103 void * StartUpdateThread(void * parameter);
00104
00112 int _register(Connection * connection, char* response);
00113
00120 void _help(Connection* connection, char* response);
00121
00129 int _myinfo(Connection* connection, char* response);
00130
00136 void _who(char* response);
00137
00144 void _rand_age(Connection* connection, char* response);
00145
00152 void _rand_gpa(Connection* connection, char* response);
00153
00160 void _advertisement(Connection * connection, char * response);
00164 #endif

```

8.5 Data.c

```

00001
00005 #include "Data.h"
00006
00007 char * accepted_userIDs[] = {
00008     "chen",
00009     "beal389",
00010     "bol4559",
00011     "cal6258",
00012     "kre5277",
00013     "lon1150",
00014     "mas9309",
00015     "mes08346",
00016     "mil7233",
00017     "nef9476",
00018     "nov7488",
00019     "pan9725",
00020     "rac3146",
00021     "rub4133",
00022     "shr5683",
00023     "vay3083",
00024     "yos2327"};
00025
00026 char * userFullNames[] = {
00027     "Weifeng Chen",
00028     "Christian Beatty",

```

```

00029     "Emily Bolles",
00030     "Cameron Calhoun",
00031     "Ty Kress",
00032     "Cody Long",
00033     "Caleb Massey",
00034     "Christian Messmer",
00035     "Karl Miller",
00036     "Jeremiah Neff",
00037     "Kaitlyn Novacek",
00038     "Joshua Panaro",
00039     "Caleb Rachocki",
00040     "Caleb Ruby",
00041     "Paul Shriner",
00042     "Alan Vayansky",
00043     "Assefa Ayalew Yoseph";
00044

```

8.6 Data.h

```

00001 #ifndef Data_h
00002 #define Data_h
00012 #define RECORD_COUNT 17
00017 #define ID_MAX_LENGTH 9
00022 #define NAME_MAX_LENGTH 21
00023
00028 #define IP_LENGTH 16
00029
00030
00034 extern char * accepted_userIDs[];
00035
00039 extern char * userFullNames[];
00040
00044 typedef struct
00045 {
00047     char id[ID_MAX_LENGTH];
00049     char name[NAME_MAX_LENGTH];
00051     int age;
00053     float gpa;
00055     short connected;
00057     char ip[IP_LENGTH];
00059     long lastConnection;
00061     short registered;
00062 } User;
00063
00067 #endif

```

8.7 File.c

```

00001
00005 #include <unistd.h>
00006 #include <fcntl.h>
00007 #include <stdio.h>
00008 #include <string.h>
00009 #include <stdlib.h>
00010 #include <unistd.h>
00011 #include <dirent.h>
00012 #include "File.h"
00013 #include "Data.h"
00014 #include "Util.h"
00015 #include "Log.h"
00016
00017 short FileStatus(char * filename) {
00018     int err = access(filename, F_OK);
00019     if(!err) {
00020         err = access(filename, F_OK | R_OK | W_OK);
00021         if(!err) {
00022             return 1;
00023         }
00024         return 2;
00025     }
00026     return 0;
00027 }
00028
00029 FILE * CreateOrOpenFileVerbose(char * filename, char * defaultContents) {
00030     FILE * file = NULL;
00031     int status = FileStatus(filename);
00032
00033     if(status == 2) {
00034         printRed("Error: %s exists but you do not have permission to access it.\n", filename);

```

```

00035         return NULL;
00036     }
00037
00038     if(status == 0) {
00039         printf("Creating %s.\n", filename);
00040         file = fopen(filename, "w+");
00041     } else if(status == 1) {
00042         printf("Opening %s.\n", filename);
00043         file = fopen(filename, "r+");
00044     }
00045
00046
00047     if(file == NULL) {
00048         printf(COLOR_RED);
00049         if(status == 0) {
00050             printf("Failed to create %s.\n", filename);
00051             perror("Error: ");
00052         } else if(status == 1) {
00053             printf("Failed to open %s.\n", filename);
00054             perror("Error: ");
00055         } else {
00056             printf("Unknown error opening %s.", filename);
00057         }
00058         printf(COLOR_RESET);
00059         return NULL;
00060     }
00061
00062     if(status == 0) {
00063         printGreen("Created %s.\n", filename);
00064         if(defaultContents != NULL) {
00065             fpos_t start_pos;
00066             fgetpos(file, &start_pos);
00067             fprintf(file, defaultContents, 0);
00068             fsetpos(file, &start_pos);
00069         }
00070     } else if(status == 1) {
00071         printGreen("Opened %s.\n", filename);
00072     }
00073
00074     return file;
00075 }
00076
00077
00078 int ReadRegisteredFileIntoUsersMap(FILE * reg_file, map * users_map) {
00079
00080     char userID[ID_MAX_LENGTH];
00081     int user_age;
00082     float user_gpa;
00083     char userLastIP[IP_LENGTH];
00084     long lastConnection;
00085
00086     int scan_items;
00087     int line = 1;
00088
00089     while( (scan_items = fscanf(reg_file, "%s\t%d\t%f\t%s\t%d", userID, &user_age, &user_gpa,
userLastIP, &lastConnection)) == 5) {
00090         map_result result = Map_Get(users_map, userID);
00091         if(result.found == 0) {
00092             printYellow("Couldn't find user %s. Continuing read.\n", userID);
00093             continue;
00094         }
00095         User * user = (User*)result.data;
00096         user->age = user_age;
00097         user->gpa = user_gpa;
00098         strcpy(user->ip, userLastIP);
00099         user->lastConnection = lastConnection;
00100         user->registered = 1;
00101         line++;
00102     }
00103
00104     if(scan_items != EOF) {
00105         printRed("Error scanning registered file on line %d. Expected 5 items but had %d.\n", line,
scan_items);
00106         return 1;
00107     }
00108     return 0;
00109 }
00110
00111 void UpdateRegisteredFileFromUsersMap(FILE * reg_file, map * users_map) {
00112     int i;
00113     for(i = 0; i < RECORD_COUNT; i++) {
00114         map_result result = Map_Get(users_map, accepted_userIDs[i]);
00115         if(!result.found) {
00116             LogfError("User %s was not found in users map.", accepted_userIDs[i]);
00117             continue;
00118         }
00119     }

```

```

00120         User * user = (User *) result.data;
00121         if(user->registered) {
00122             fprintf(reg_file, "%s\t%d\t%f\t%s\t%d", user->id, user->age, user->gpa, user->ip,
user->lastConnection);
00123         }
00124     }
00125 }
00126
00127 int NumberOfFilesInDirectory(char* dir_name) {
00128     int count = 0;
00129
00130     DIR * dirp;
00131     struct dirent * entry;
00132
00133     dirp = opendir(dir_name);
00134
00135     while((entry = readdir(dirp)) != NULL) {
00136         if(entry->d_type == DT_REG) {
00137             count++;
00138         }
00139     }
00140
00141     closedir(dirp);
00142
00143     return count;
00144 }
00145
00146 void GetRandomFileNameFromDir(char * dir_name, char* file_name) {
00147     int file = RandomInteger(0, NumberOfFilesInDirectory(dir_name) - 1);
00148
00149     DIR* dirp = opendir(dir_name);
00150     struct dirent * entry;
00151
00152     while(file >= 0 && ((entry = readdir(dirp)) != NULL)) {
00153         if(entry->d_type == DT_REG) {
00154             file--;
00155         }
00156     }
00157
00158     if(entry != NULL) {
00159         strcpy(file_name, entry->d_name);
00160     }
00161
00162     closedir(dirp);
00163 }
00164
00165 int ReadSettingsFileIntoSettingsMap(FILE * settings_file, map * settings_map) {
00166     char key_read[100];
00167     char value_read[100];
00168
00169     int scan_items;
00170     int line = 1;
00171
00172     while( (scan_items = fscanf(settings_file, " %s = %s ", key_read, value_read)) == 2) {
00173         char * key_alloc = malloc( (strlen(key_read)+1) * sizeof(char));
00174         memset(key_alloc, 0, strlen(key_read)+1);
00175         strcpy(key_alloc, key_read);
00176         char * val_alloc = malloc( (strlen(value_read)+1) * sizeof(char));
00177         memset(val_alloc, 0, strlen(value_read)+1);
00178         strcpy(val_alloc, value_read);
00179         Map_Set(settings_map, key_alloc, val_alloc);
00180         line++;
00181     }
00182
00183     if(scan_items != EOF) {
00184         printRed("Error scanning settings file on line %d. Expected 2 items but had %d.\n", line,
scan_items);
00185         return 1;
00186     }
00187     return 0;
00188 }
00189
00190 void CatFileToBuffer(char* file_name, char* buffer, size_t buffer_size) {
00191     if(FileStatus(file_name)) {
00192         FILE* file = CreateOrOpenFileVerbose(file_name, NULL);
00193         char* temp = malloc(buffer_size);
00194
00195         while(fgets(temp, buffer_size - strlen(buffer), file) && buffer_size - strlen(buffer) > 1) {
00196             strcat(buffer, temp);
00197         }
00198
00199         free(temp);
00200     }
00201 }
00202
00203 int CreateLockfile()
00204 {

```

```

00205     FILE * file = fopen(LOCKFILE, "w");
00206     if(file == NULL) {
00207         return 0;
00208     }
00209     fprintf(file, "0 %d", getpid());
00210     fclose(file);
00211     return 1;
00212 }
00213
00214 int DeleteLockfile()
00215 {
00216     return remove(LOCKFILE);
00217 }
00218

```

8.8 File.h

```

00001 #ifndef Files_h
00002 #define Files_h
00008 #include <stdio.h>
00009 #include "map.h"
00010
00011 // ~~~~ Macros ~~~~ //
00012
00016 #define LOCKFILE "/tmp/lab5.lock"
00017
00026 #define REGISTERED_FILE "registered.txt"
00027
00038 #define SERVER_SETTINGS_FILE "server-settings.txt"
00039
00045 #define ADS_DIR "ads"
00046
00047 // ~~~~~ General File Functions ~~~~~ //
00048
00053 short FileStatus(char * filename);
00054
00063 FILE * CreateOrOpenFileVerbose(char * filename, char * defaultContents);
00064
00072 int ReadRegisteredFileIntoUsersMap(FILE * reg_file, map * users_map);
00073
00074
00082 int ReadSettingsFileIntoSettingsMap(FILE * settings_file, map * settings_map);
00083
00090 void UpdateRegisteredFileFromUsersMap(FILE * reg_file, map * users_map);
00091
00098 void GetRandomFileNameFromDir(char* dir_name, char* file_name);
00099
00106 int NumberOfFilesInDirectory(char* dir_name);
00107
00115 void CatFileToBuffer(char* file_name, char* buffer, size_t buffer_size);
00116
00117 /**
00118     Creates a lockfile.
00119     @warning This should only be called by a running server process when a lockfile does not already
00120     exist.
00120     @returns 1 on success, otherwise 0.
00121 */
00122 int CreateLockfile();
00123
00124 /**
00125     Deletes a lockfile.
00126     @returns 1 on success, otherwise 0.
00127 */
00128 int DeleteLockfile();
00129
00133 #endif

```

8.9 Log.c

```

00001
00004 #include "Log.h"
00005 #include <stdarg.h>
00006
00007 LogSettings L;
00008
00009 void InitializeLogger(FILE* _printStream, char printLevel, char logLevel, char printAllToStdOut) {
00010     L.ostream = _printStream;
00011     L.printLevel = printLevel;
00012     L.logLevel = logLevel;

```

```

00013     L.printAllToStdOut = printAllToStdOut;
00014 }
00015
00016 void _logf(int level, const char * format, va_list argptr) {
00017
00018     // TODO add logic to include timestamp to the
00019     // Stamp level to log
00020     // if printing to standard out is used print with color based on level
00021     if (level >= L.printLevel || L.printAllToStdOut != 0) {
00022         vprintf(format, argptr);
00023     } else if (level >= L.logLevel) {
00024         vfprintf(L.ostream, format, argptr);
00025     }
00026 }
00027
00028 void LogfFatal(const char * format, ...) {
00029     va_list argptr;
00030     va_start(argptr, format);
00031     _logf(FATAL, format, argptr);
00032     va_end(argptr);
00033 }
00034
00035 void LogfError(const char * format, ...) {
00036     va_list argptr;
00037     va_start(argptr, format);
00038     _logf(ERROR, format, argptr);
00039     va_end(argptr);
00040 }
00041
00042 void LogfWarning(const char * format, ...) {
00043     va_list argptr;
00044     va_start(argptr, format);
00045     _logf(WARNING, format, argptr);
00046     va_end(argptr);
00047 }
00048
00049 void LogfInfo(const char * format, ...) {
00050     va_list argptr;
00051     va_start(argptr, format);
00052     _logf(INFO, format, argptr);
00053     va_end(argptr);
00054 }
00055
00056 void LogfDebug(const char * format, ...) {
00057     va_list argptr;
00058     va_start(argptr, format);
00059     _logf(DEBUG, format, argptr);
00060     va_end(argptr);
00061 }
00062
00063 void LogfTrace(const char * format, ...) {
00064     va_list argptr;
00065     va_start(argptr, format);
00066     _logf	TRACE, format, argptr);
00067     va_end(argptr);
00068 }

```

8.10 Log.h

```

00001 #ifndef Log_h
00002 #define Log_h
00003
00004 #include <stdio.h>
00005
00020 #define TRACE 0
00021 #define DEBUG 1
00022 #define INFO 2
00023 #define WARNING 3
00024 #define ERROR 4
00025 #define FATAL 5
00026
00027 typedef struct
00028 {
00030     FILE * ostream;
00032     char printLevel;
00034     char logLevel;
00036     char printAllToStdOut;
00037
00038 } LogSettings;
00039
00040 //set up functions that print to stdout no matter if printtostdout is set to true
00041
00048 void LogfFatal(const char * format, ...);

```

```
00049
00057 void LogfError(const char * format, ...);
00058
00065 void LogfWarning(const char * format, ...);
00066
00073 void LogfInfo(const char * format, ...);
00074
00081 void LogfDebug(const char* format, ...);
00082
00089 void LogfTrace(const char * format, ...);
00090
00099 void InitializeLogger(FILE* _printStream, char printLevel, char logLevel, char printAllToStdOut);
00100
00101
00105 #endif
```

8.11 main.c

```
00001 #include <stdio.h>
00002 #include "Util.h"
00003 #include "Process.h"
00004 #include <string.h>
00005
00006
00096 int main(int argc, char **argv) {
00097
00098     if(argc <= 1 )
00099     {
00100         RunCommand();
00101     }
00102     else if (strcmp(argv[1], "headless") == 0)
00103     {
00104         RunHeadless(argv[0]);
00105     }
00106     else if (strcmp(argv[1], "stop") == 0)
00107     {
00108         StopCommand();
00109     }
00110     else
00111     {
00112         RunCommand();
00113     }
00114     return 0;
00115 }
```

8.12 map.c

```
00001
00005 #include "stdlib.h"
00006 #include "string.h"
00007 #include "map.h"
00008 #include "math.h"
00009
00011 int hash_log2(int num_to_log)
00012 {
00013     int t = 1;
00014     int i = 0;
00015     do
00016     {
00017         num_to_log = num_to_log & ~t;
00018         t = t << 1;
00019         i++;
00020     } while (num_to_log > 0);
00021     return i;
00022 }
00023
00025 int hash_upperLimit(int bitsize)
00026 {
00027     return 1 << bitsize;
00028 }
00029
00031 int char_ratio = (int)(sizeof(int) / sizeof(char));
00032
00034 int hash_string(int hash_table_size, char *string, int strlen)
00035 {
00036     int i, hash = 2166136261;
00037     for (i = 0; i < strlen; i += 1)
00038     {
00039         hash *= 16777619;
```



```

00040     hash ^= string[i];
00041 }
00042 if (hash < 0)
00043 {
00044     hash *= -1;
00045 }
00046 return hash % hash_table_size;
00047 }
00048
00049 map *NewMap(int capacity)
00050 {
00051     int log2 = hash_log2(capacity);
00052     int capac = hash_upperLimit(log2);
00053     int sz = sizeof(struct _map_bucket) * capac;
00054     struct _map_bucket *buckets = malloc(sz);
00055     memset(buckets, 0, sz);
00056     int i;
00057     for (i = 0; i < capac; i++)
00058     {
00059         buckets[i] = (struct _map_bucket){NULL, NULL, NULL};
00060     }
00061     map newm = (map){capac, buckets};
00062     map *map_p = malloc(sizeof(map));
00063     *map_p = newm;
00064     return map_p;
00065 }
00066
00068 void _bucket_insert(struct _map_bucket *bucket, char *key, void *value)
00069 {
00070     struct _map_bucket *check = bucket;
00071     while (check->key != NULL)
00072     {
00073         if (strcmp(check->key, key) == 0)
00074         {
00075             check->data = value;
00076             return;
00077         }
00078         if (check->next == NULL)
00079         {
00080             check->next = malloc(sizeof(struct _map_bucket));
00081             *(check->next) = (struct _map_bucket){NULL, NULL, NULL};
00082         }
00083         check = check->next;
00084     }
00085     check->key = key;
00086     check->data = value;
00087 }
00088
00089 void Map_Set(map *a_map, char *key, void *value)
00090 {
00091     int keyl = (int)strlen(key);
00092     int hash = hash_string(a_map->size, key, keyl);
00093     _bucket_insert(&(a_map->buckets[hash]), key, value);
00094 }
00096 void _bucket_get(struct _map_bucket *bucket, char *key, map_result *result)
00097 {
00098     struct _map_bucket *check = bucket;
00099     while (check->key != NULL)
00100     {
00101         if (strcmp(check->key, key) == 0)
00102         {
00103             result->found = 1;
00104             result->data = check->data;
00105             return;
00106         }
00107         else if (check->next != NULL)
00108         {
00109             check = check->next;
00110         }
00111         else
00112         {
00113             result->found = 0;
00114             break;
00115         }
00116     }
00117 }
00118
00119 map_result Map_Get(map *a_map, char *key)
00120 {
00121     map_result res = (map_result){0, NULL};
00122     int keyl = (int)strlen(key);
00123     int hash = hash_string(a_map->size, key, keyl);
00124     _bucket_get(&(a_map->buckets[hash]), key, &res);
00125     return res;
00126 }
00127
00128 void _bucket_delete(struct _map_bucket *bucket, char *key, short free_it, map_result *result)

```

```

00129 {
00130     struct _map_bucket *last = bucket;
00131     struct _map_bucket *next = bucket->next;
00132     while (next != NULL)
00133     {
00134         if (strcmp(next->key, key) == 0)
00135         {
00136             result->found = 1;
00137             result->data = next->data;
00138             if (free_it)
00139             {
00140                 free(next->data);
00141                 result->data = NULL;
00142             }
00143             last->next = next->next;
00144             free(next);
00145         }
00146         else
00147         {
00148             last = next;
00149             next = next->next;
00150         }
00151     }
00152 }
00153
00154 map_result Map_Delete(map *a_map, char *key, short free_it)
00155 {
00156     map_result res = (map_result){0, NULL};
00157     int keyl = (int)strlen(key);
00158     int hash = hash_string(a_map->size, key, keyl);
00159
00160     struct _map_bucket top = a_map->buckets[hash];
00161     if (top.key == NULL)
00162     {
00163         return res;
00164     }
00165     if (strcmp(top.key, key) == 0)
00166     {
00167         res.found = 1;
00168         res.data = top.data;
00169         if (free_it)
00170         {
00171             free(top.data);
00172             res.data = NULL;
00173         }
00174         if (top.next != NULL)
00175         {
00176             a_map->buckets[hash] = *(top.next);
00177             free(top.next);
00178         }
00179         else
00180         {
00181             a_map->buckets[hash] = (struct _map_bucket){NULL, NULL, NULL};
00182         }
00183         return res;
00184     }
00185     if (top.next == NULL)
00186     {
00187         return res;
00188     }
00189     _bucket_delete(&(a_map->buckets[hash]), key, free_it, &res);
00190
00191     return res;
00192 }

```

8.13 map.h

```

00001 #ifndef map_h
00002 #define map_h
00003
00041 // -----
00042 //           Hashing Math
00043 // -----
00044
00051 int hash_log2(int number_to_log);
00052
00062 int hash_string(int hash_table_capacity, char *string, int strlen);
00063
00070 int hash_upperLimit(int bitsize);
00071
00072 // -----
00073 //           General Map Operations
00074 // -----

```

```

00075
00081 struct _map_bucket
00082 {
00084     char *key;
00086     void *data;
00088     struct _map_bucket *next;
00089 };
00090
00101 typedef struct
00102 {
00103     int size;
00104     struct _map_bucket *buckets;
00105 } map;
00106
00111 typedef struct
00112 {
00113
00114     short found;
00115     void *data;
00116 } map_result;
00117
00124 map *NewMap(int capacity);
00125
00133 void Map_Set(map *a_map, char *key, void *value);
00134
00141 map_result Map_Get(map *a_map, char *key);
00142
00150 map_result Map_Delete(map *a_map, char *key, short free_it);
00151
00152 #endif

```

8.14 Process.c

```

00001
00005 #include <stdio.h>
00006 #include <string.h>
00007 #include <signal.h>
00008 #include <stdlib.h>
00009 #include <unistd.h>
00010 #include "Data.h"
00011 #include "Build.h"
00012 #include "map.h"
00013 #include "File.h"
00014 #include "Util.h"
00015 #include "Server.h"
00016 #include "Log.h"
00017 #include "Connection.h"
00018
00020 User * users_array;
00022 map * users_map;
00024 map * settings_map;
00026 char * default_settings = "port           = 3000\n"
00027                          "send_buffer_size = 1024\n"
00028                          "receive_buffer_size = 1024\n"
00029                          "backlog           = 10\n"
00030                          "max_connections    = 20\n"
00031                          "log_file           = log.txt\n"
00032                          "log_level          = 1\n"
00033                          "log_to_console     = true";
00034
00036 int active_clients;
00037
00038 int _initializeLogger() {
00039     //char* fileName = "log.txt";
00040     int printLevel, LogLevel; // printAlltoStdOut;
00041     map_result result = Map_Get(settings_map, "log_file");
00042     if(!result.found) {
00043         printYellow("No output file found. Defaulting to 'log.txt'\n");
00044     } else {
00045         //fileName = result.data;
00046     }
00047
00048     result = Map_Get(settings_map, "print_level");
00049     if(!result.found) {
00050         printYellow("No print_level found, defaulting to 3\n");
00051         printLevel = 3;
00052     } else {
00053         printLevel = atoi(result.data);
00054         if(printLevel < 0 || printLevel > 5) {
00055             printYellow("Invalid print_level of %d, defaulting to 3\n", printLevel);
00056             printLevel = 3;
00057         }
00058     }

```

```

00059
00060     result = Map_Get(settings_map, "log_level");
00061     if(!result.found) {
00062         printYellow("No log_level found, defaulting to 3\n");
00063         LogLevel = 3;
00064     } else {
00065         LogLevel = atoi(result.data);
00066         if(LogLevel < 0 || LogLevel > 5) {
00067             printYellow("Invalid log_level of %d, defaulting to 3\n", LogLevel);
00068             LogLevel = 3;
00069         }
00070     }
00071
00072     result = Map_Get(settings_map, "log_to_console");
00073     if(!result.found) {
00074         printYellow("No log_to_console found, defaulting to true\n");
00075         //printAlltoStdOut = 1;
00076     } else {
00077         if(strcmp(result.data, "true") == 0) {
00078             //printAlltoStdOut = 1;
00079         } else if(strcmp(result.data, "false") == 0) {
00080             //printAlltoStdOut = 0;
00081         } else {
00082             printYellow("invalid data in log_to_console, defaulting to true\n");
00083             //printAlltoStdOut = 1;
00084         }
00085     }
00086     return 1;
00087 }
00088
00089 int Initialize() {
00090
00091     // Create the data structures on the heap.
00092     printf("Initializing User data structures.\n");
00093     users_array = CreateUsersArray(accepted_userIDs, userFullNames, RECORD_COUNT);
00094     users_map = CreateUsersMap(users_array, RECORD_COUNT);
00095     active_clients = 0;
00096     printGreen("User data structures initialized.\n");
00097
00098     // Create the registered file that tracks registered users.
00099     printf("Checking for registered file.\n");
00100     FILE * reg_file = CreateOrOpenFileVerbose(REGISTERED_FILE, NULL);
00101     if(reg_file == NULL) {
00102         printRed("Initialization failed during accessing of file: %s.\n", REGISTERED_FILE);
00103         return 0;
00104     }
00105
00106     // Update the User's map with with the data from the registered file.
00107     printf("Reading registered file.\n");
00108     int read_error = ReadRegisteredFileIntoUsersMap(reg_file, users_map);
00109     fclose(reg_file);
00110     if(read_error) {
00111         printRed("Initialization failed during reading of file: %s.\n", REGISTERED_FILE);
00112         return 0;
00113     }
00114     printGreen("Loaded %s into users map.\n", REGISTERED_FILE);
00115
00116     printf("Reading settings file.\n");
00117     settings_map = NewMap(50);
00118     FILE * settings_file = CreateOrOpenFileVerbose(SERVER_SETTINGS_FILE, default_settings);
00119     if(settings_file == NULL) {
00120         printRed("Initialization failed during accessing of file: %s.\n", SERVER_SETTINGS_FILE);
00121         return 0;
00122     }
00123     int settings_read_err = ReadSettingsFileIntoSettingsMap(settings_file, settings_map);
00124     if(settings_read_err) {
00125         printRed("Initialization failed while reading settings file %s. Correct this file or delete it
so a default can be generated.\n", SERVER_SETTINGS_FILE);
00126         return 0;
00127     }
00128     fclose(settings_file);
00129     printGreen("Read %s.\n", SERVER_SETTINGS_FILE);
00130
00131     printf("Initializing logger.\n");
00132     int logger_initialized = _initializeLogger();
00133     if(!logger_initialized) {
00134         printRed("Failed to initalize logger.\n");
00135     }
00136
00137
00138     printf("Initializing server.\n");
00139     int server_initialized = InitializeServer(settings_map);
00140     if(!server_initialized) {
00141         printRed("Failed to initialize server.\n");
00142         return 0;
00143     }
00144

```

```

00145
00146     return 1;
00147 }
00148
00149 void SignalHandle(int signo) {
00150     if(signo == SIGINT || signo == SIGTERM) {
00151         printYellow("Received signal. Shutting down server.\n");
00152         DeleteLockfile();
00153         exit(0);
00154     }
00155 }
00156 }
00157
00158 int RunCommand() {
00159     if (FileStatus(LOCKFILE) > 0)
00160     {
00161         printf("Server process already running.\n");
00162         return 0;
00163     }
00164     signal(SIGTERM, SignalHandle);
00165     signal(SIGINT, SignalHandle);
00166     int lockfile_success = CreateLockfile();
00167     if(!lockfile_success) {
00168         printRed("Failed to create Lockfile! Server cannot start.");
00169         return 0;
00170     }
00171     int init_success = Initialize();
00172     if(!init_success) {
00173         printRed("Could not start the server due to failed initialization.\n");
00174         return 0;
00175     }
00176     printf("Running server.\n");
00177     int server_success = StartServer(users_map);
00178     if(!server_success) {
00179         printRed("There was a problem running the server.\n");
00180         return 0;
00181     }
00182     int delete_lockfile_success = DeleteLockfile();
00183     if(!delete_lockfile_success) {
00184         printRed("There was a problem deleting the Lockfile.\n");
00185         return 0;
00186     }
00187     return 1;
00188 }
00189
00190 void RunHeadless(char *processName) {
00191     if (FileStatus(LOCKFILE) > 0)
00192     {
00193         printf("Server process already running.\n");
00194         return;
00195     }
00196     char commandFront[] = " nohup ";
00197     char commandEnd[] = " & exit";
00198     size_t comm_length = strlen(commandFront) + strlen(commandEnd) + strlen(processName) + 1;
00199     char *commandFull = malloc(comm_length * sizeof(char));
00200     memset(commandFull, 0, comm_length * sizeof(char));
00201     strcpy(commandFull, commandFront);
00202     strcat(commandFull, processName);
00203     strcat(commandFull, commandEnd);
00204
00205     printf("Executing: %s\n", commandFull);
00206     popen(commandFull, "we");
00207     printf("Server running headlessly.\n");
00208 }
00209
00210 int TerminateExistingServer()
00211 {
00212     FILE *file = fopen(LOCKFILE, "r");
00213     if (file == NULL)
00214     {
00215         perror("Error opening lockfile");
00216         return -1;
00217     }
00218     int need_rewrite;
00219     int pid = 0;
00220     fscanf(file, "%d %d", &need_rewrite, &pid);
00221     fclose(file);
00222     if (pid > 0)
00223     {
00224         return kill(pid, SIGTERM);
00225     }
00226     return -2;
00227 }
00228
00229 void StopCommand() {
00230     printYellow("\nStopping server...\n");
00231     int err = TerminateExistingServer();

```

```

00232     if (err)
00233     {
00234         if (err == -1)
00235         {
00236             printRed("Server isn't running.\n");
00237         }
00238         else if (err == -2)
00239         {
00240             printRed("Lockfile did not contain a valid process id!\n");
00241         }
00242         else
00243         {
00244             printRed("Sending terminate signal failed!\n");
00245         }
00246     }
00247     else
00248     {
00249         printGreen("Server terminated.\n");
00250     }
00251 }
00252
00253 void ResetCommand() {
00254     if(FileStatus(REGISTERED_FILE)) {
00255         fclose(fopen(REGISTERED_FILE, "w")); //empties the registered file
00256     }
00257 }

```

8.15 Process.h

```

00001 #ifndef Process_h
00002 #define Process_h
00003 /**
00004  * \defgroup Process
00005  * \brief This module holds functions that realize the primary business logic of the program.
00006  * \details When command line arguments are parsed by main, functions in this module are called.
00007  * @{
00008  */
00009
00010 /**
00011  * Handles an interrupt or quit signal.
00012  *
00013  * Send server to shutdown mode, resulting in graceful deletion of lockfile.
00014  */
00015 void SignalHandle(int signo);
00016
00017 /**
00018  * Performs initializing activities which must occur prior to a server loop starting.
00019  *
00020  * Will print errors if there are problems initializing.
00021  *
00022  * @returns 1 on success, otherwise 0.
00023  */
00024 int Initialize();
00025
00026 /**
00027  * Runs the server.
00028  *
00029  * @returns 1 on success, otherwise 0.
00030  */
00031 int RunCommand();
00032
00033 /**
00034  * Finds the process ID of a running server using the lockfile, and calls kill on it, sending a
    SIGTERM.
00035  *
00036  * @returns -1 if the file doesn't exist, -2 if no valid process ID existed in the file, 1 if sending
    the kill signal failed, or 0 on success.
00037  */
00038 int TerminateExistingServer();
00039
00040 void StopCommand();
00041
00042 void RunHeadless(char *processName);
00043
00044 #endif

```

8.16 Server.c

```

00001
00002 #include <stdlib.h>

```

```

00006 #include <netinet/in.h>
00007 #include <sys/types.h>
00008 #include <sys/socket.h>
00009 #include <arpa/inet.h>
00010 #include <linux/net.h>
00011 #include <stdio.h>
00012 #include <pthread.h>
00013 #include <unistd.h>
00014
00015 #include "Server.h"
00016 #include "Connection.h"
00017 #include "Util.h"
00018 #include "map.h"
00019
00020 ServerProperties server;
00021 Connection * connections;
00022
00023 // A private function just for reading the settings map into the server struct and printing warnings
00024 // as necessary.
00025 void _readSettingsMapIntoServerStruct(map * server_settings) {
00026     map_result result = Map_Get(server_settings, "port");
00027     if(!result.found) {
00028         printYellow("No port setting found. Defaulting to 3000.\n");
00029         server.port = 3000;
00030     } else {
00031         int found_port = atoi(result.data);
00032         if(found_port <= 0) {
00033             printYellow("Invalid port setting: %s. Defaulting to 3000.\n", result.data);
00034             server.port = htons(3000);
00035         } else {
00036             server.port = htons(found_port);
00037         }
00038     }
00039     result = Map_Get(server_settings, "send_buffer_size");
00040     if(!result.found) {
00041         printYellow("No send_buffer_size setting found. Defaulting to 1024.\n");
00042         server.send_buffer_size = 1024;
00043     } else {
00044         int found_sb_size = atoi(result.data);
00045         if(found_sb_size <= 0) {
00046             printYellow("Invalid send_buffer_size setting: %s. Defaulting to 1024.\n", result.data);
00047             server.send_buffer_size = 1024 * sizeof(char);
00048         } else {
00049             server.send_buffer_size = found_sb_size * sizeof(char);
00050         }
00051     }
00052     result = Map_Get(server_settings, "receive_buffer_size");
00053     if(!result.found) {
00054         printYellow("No receive_buffer_size setting found. Defaulting to 1024.\n");
00055         server.receive_buffer_size = 1024;
00056     } else {
00057         int found_rb_size = atoi(result.data);
00058         if(found_rb_size <= 0) {
00059             printYellow("Invalid receive_buffer_size setting: %s. Defaulting to 1024.\n",
00060                 result.data);
00061             server.receive_buffer_size = 1024 * sizeof(char);
00062         } else {
00063             server.receive_buffer_size = found_rb_size * sizeof(char);
00064         }
00065     }
00066     result = Map_Get(server_settings, "backlog");
00067     if(!result.found) {
00068         printYellow("No backlog setting found. Defaulting to 10.\n");
00069         server.backlog = 10;
00070     } else {
00071         int found_backlog = atoi(result.data);
00072         if(found_backlog <= 0) {
00073             printYellow("Invalid backlog setting: %s. Defaulting to 10.\n", result.data);
00074             server.backlog = 10;
00075         } else {
00076             server.backlog = found_backlog;
00077         }
00078     }
00079     result = Map_Get(server_settings, "max_connections");
00080     if(!result.found) {
00081         printYellow("No max_connections setting found. Defaulting to 20.\n");
00082         server.max_connections = 20;
00083     } else {
00084         int found_max_connections = atoi(result.data);
00085         if(found_max_connections <= 0) {
00086             printYellow("Invalid max_connections setting: %s. Defaulting to 20.\n", result.data);
00087             server.max_connections = 20;
00088         } else {
00089             server.max_connections = found_max_connections;
00090         }
00091     }
00092 }

```

```

00091
00092 int InitializeServer(map * server_settings) {
00093     _readSettingsMapIntoServerStruct(server_settings);
00094     connections = malloc(server.max_connections * sizeof(Connection));
00095     int i;
00096     for(i=0; i < server.max_connections; i++) {
00097         connections[i].status = ConnectionStatus_CLOSED;
00098     }
00099     printGreen("Server initialized with %d max connections.\n", server.max_connections);
00100     return 1;
00101 }
00102
00103 int StartServer(map * users_map) {
00104     int serverSocket = 0;
00105     struct sockaddr_in server_address;
00106     // Record the time the server started.
00107     time(&server.time_started);
00108     // Get a socket file pointer associated with ipv4 internet protocols that represents a two-way
    connection based byte stream.
00109     serverSocket = socket(AF_INET, SOCK_STREAM, 0);
00110     server_address.sin_family = AF_INET;
00111     // Set the address to bind to all available interfaces.
00112     server_address.sin_addr.s_addr = htonl(INADDR_ANY);
00113     // Set the port.
00114     server_address.sin_port = server.port;
00115     // Assign a name to the socket.
00116     int bind_error = bind(serverSocket, (struct sockaddr*)&server_address, sizeof(server_address));
00117     if(bind_error) {
00118         printRed("Error binding the server to port %d.\n", ntohs(server.port));
00119         perror("Bind Error:");
00120         return 0;
00121     }
00122     // Initialized a shared space that will be used across threads.
00123     ClientShared * shared = InitializeShared(users_map, server.send_buffer_size,
    server.receive_buffer_size);
00124     // The update thread is responsible for checking if there is 'dirty' data that should be saved to
    the registered user's file.
00125     pthread_t registered_update_thread;
00126     pthread_create(&registered_update_thread, NULL, StartUpdateThread, NULL);
00127     printBlue("Server listening on port: %d\n", ntohs(server.port));
00128     // begin listening according to the socket settings
00129     listen(serverSocket, server.backlog);
00130     while(!shared->shutting_down) {
00131         // Get an available connection.
00132         Connection * next_client = NextAvailableConnection();
00133         if(next_client == NULL) {
00134             printYellow("Server connections are maxxed.\n");
00135             sleep(1);
00136             continue;
00137         }
00138         // Accept a connection.
00139         next_client->address_length = sizeof(next_client->address);
00140         next_client->socket = accept(serverSocket, (struct sockaddr *)&(next_client->address),
    &(next_client->address_length));
00141         if(next_client->socket < 0)
00142         {
00143             printRed("Failed to accept() client!\n");
00144             sleep(1);
00145             continue;
00146         }
00147         printBlue("New client connection from IP: %s\n", inet_ntoa(next_client->address.sin_addr));
00148         next_client->status = ConnectionStatus_ACTIVE;
00149         // Start a thread to handle communication from that connection.
00150         pthread_create(&(next_client->thread_id), NULL, StartConnectionThread, next_client);
00151     }
00152     return 1;
00153 }
00154
00155
00156 Connection * NextAvailableConnection()
00157 {
00158     int i;
00159     for(i = 0; i < server.max_connections; i++) {
00160         if(connections[i].status == ConnectionStatus_CLOSED)
00161         {
00162             return &(connections[i]);
00163         }
00164     }
00165     return NULL;
00166 }
00167
00168

```


8.17 Server.h

```

00001 #ifndef Server_h
00002 #define Server_h
00008 #include <stdint.h>
00009 #include <time.h>
00010 #include "map.h"
00011 #include "Connection.h"
00012
00020 typedef struct {
00022     uint16_t port;
00024     size_t send_buffer_size;
00026     size_t receive_buffer_size;
00028     int backlog;
00030     int active_connections;
00032     int max_connections;
00034     time_t time_started;
00035 } ServerProperties;
00036
00042 int InitializeServer();
00043
00050 int StartServer(map * users_map);
00051
00057 Connection * NextAvailableConnection();
00058
00062 #endif

```

8.18 Util.c

```

00001
00005 #include <stdio.h>
00006 #include <stdarg.h>
00007 #include <string.h>
00008 #include <stdlib.h>
00009 #include "Util.h"
00010
00011 void printRed(const char * format, ...) {
00012     printf(COLOR_RED);
00013     va_list args;
00014     va_start(args, format);
00015     vprintf(format, args);
00016     va_end(args);
00017     printf(COLOR_RESET);
00018 }
00019
00020 void printGreen(const char * format, ...) {
00021     printf(COLOR_GREEN);
00022     va_list args;
00023     va_start(args, format);
00024     vprintf(format, args);
00025     va_end(args);
00026     printf(COLOR_RESET);
00027 }
00028
00029 void printYellow(const char * format, ...) {
00030     printf(COLOR_YELLOW);
00031     va_list args;
00032     va_start(args, format);
00033     vprintf(format, args);
00034     va_end(args);
00035     printf(COLOR_RESET);
00036 }
00037
00038 void printBlue(const char * format, ...) {
00039     printf(COLOR_BLUE);
00040     va_list args;
00041     va_start(args, format);
00042     vprintf(format, args);
00043     va_end(args);
00044     printf(COLOR_RESET);
00045 }
00046
00047 int RandomInteger(int min, int max)
00048 {
00049     int r_add = rand() % (max - min + 1);
00050     return r_add + min;
00051 }
00052
00053 float RandomFloat(float min, float max)
00054 {
00055     float dif = max - min;
00056     int rand_int = rand() % (int)(dif * 10000);
00057     return min + (float)rand_int / 10000.0;

```

```
00058 }
00059
00060 short RandomFlag(float percentage_chance)
00061 {
00062     float random_value = (float)rand() / RAND_MAX;
00063     if (random_value < percentage_chance)
00064     {
00065         return 1;
00066     }
00067     return 0;
00068 }
00069
```

8.19 Util.h

```
00001 #ifndef Util_h
00002 #define Util_h
00010 #define COLOR_RED "\e[38;2;255;75;75m"
00012 #define COLOR_GREEN "\e[38;2;0;240;0m"
00014 #define COLOR_YELLOW "\e[38;2;255;255;0m"
00016 #define COLOR_BLUE "\e[38;2;0;240;240m"
00018 #define COLOR_RESET "\e[0m"
00019
00025 void printRed(const char * format, ...);
00026
00032 void printGreen(const char * format, ...);
00033
00039 void printYellow(const char * format, ...);
00040
00046 void printBlue(const char * format, ...);
00047
00048
00055 int RandomInteger(int min, int max);
00056
00063 float RandomFloat(float min, float max);
00064
00071 short RandomFlag(float percentage_chance);
00072
00076 #endif
```

Index

- [_advertisement](#)
[Connection](#), [22](#)
 - [_help](#)
[Connection](#), [17](#)
 - [_map_bucket](#), [51](#)
 - [_myinfo](#)
[Connection](#), [18](#)
 - [_rand_age](#)
[Connection](#), [21](#)
 - [_rand_gpa](#)
[Connection](#), [20](#)
 - [_register](#)
[Connection](#), [17](#)
 - [_who](#)
[Connection](#), [19](#)
- [accepted_userIDs](#)
[Data](#), [24](#)
- [active_clients](#)
[Process](#), [49](#)
- [active_connections](#)
[ServerProperties](#), [58](#)
- [address](#)
[Connection](#), [54](#)
- [address_length](#)
[Connection](#), [54](#)
- [ADS_DIR](#)
[Files](#), [26](#)
- [backlog](#)
[ServerProperties](#), [57](#)
- [Build](#), [11](#)
[CreateUsersArray](#), [11](#)
[CreateUsersMap](#), [11](#)
- [CatFileToBuffer](#)
[Files](#), [32](#)
- [ClientShared](#), [51](#)
[dirty](#), [52](#)
[mutex](#), [52](#)
[receive_buffer_size](#), [53](#)
[send_buffer_size](#), [52](#)
[shutting_down](#), [52](#)
[users](#), [52](#)
- [COLOR_BLUE](#)
[Util](#), [43](#)
- [COLOR_GREEN](#)
[Util](#), [42](#)
- [COLOR_RED](#)
[Util](#), [42](#)
- [COLOR_RESET](#)
[Util](#), [43](#)
- [COLOR_YELLOW](#)
[Util](#), [43](#)
- [Compilation](#), [3](#)
- [Connection](#), [12](#), [53](#)
[_advertisement](#), [22](#)
[_help](#), [17](#)
[_myinfo](#), [18](#)
[_rand_age](#), [21](#)
[_rand_gpa](#), [20](#)
[_register](#), [17](#)
[_who](#), [19](#)
[address](#), [54](#)
[address_length](#), [54](#)
[InitializeShared](#), [13](#)
[MessageAndClose](#), [16](#)
[MessageOrClose](#), [15](#)
[socket](#), [54](#)
[StartConnectionThread](#), [14](#)
[StartUpdateThread](#), [13](#)
[state](#), [55](#)
[status](#), [54](#)
[thread_id](#), [54](#)
[time_connected](#), [54](#)
[user](#), [55](#)
- [CreateOrOpenFileVerbose](#)
[Files](#), [27](#)
- [CreateUsersArray](#)
[Build](#), [11](#)
- [CreateUsersMap](#)
[Build](#), [11](#)
- [Data](#), [22](#)
[accepted_userIDs](#), [24](#)
[ID_MAX_LENGTH](#), [23](#)
[IP_LENGTH](#), [23](#)
[NAME_MAX_LENGTH](#), [23](#)
[RECORD_COUNT](#), [23](#)
[userFullNames](#), [24](#), [25](#)
- [default_settings](#)
[Process](#), [49](#)
- [dirty](#)
[ClientShared](#), [52](#)
- [ecet4640-lab5](#), [1](#)
- [Files](#), [25](#)
[ADS_DIR](#), [26](#)
[CatFileToBuffer](#), [32](#)

- CreateOrOpenFileVerbose, [27](#)
- FileStatus, [27](#)
- GetRandomFileNameFromDir, [30](#)
- LOCKFILE, [26](#)
- NumberOfFilesInDirectory, [30](#)
- ReadRegisteredFileIntoUsersMap, [28](#)
- ReadSettingsFileIntoSettingsMap, [31](#)
- REGISTERED_FILE, [26](#)
- SERVER_SETTINGS_FILE, [26](#)
- UpdateRegisteredFileFromUsersMap, [29](#)
- FileStatus
 - Files, [27](#)
- GetRandomFileNameFromDir
 - Files, [30](#)
- ID_MAX_LENGTH
 - Data, [23](#)
- InitializeLogger
 - Log, [36](#)
- InitializeServer
 - Server, [41](#)
- InitializeShared
 - Connection, [13](#)
- IP_LENGTH
 - Data, [23](#)
- LOCKFILE
 - Files, [26](#)
- Log, [33](#)
 - InitializeLogger, [36](#)
 - LogfDebug, [35](#)
 - LogfError, [34](#)
 - LogfFatal, [34](#)
 - LogfInfo, [35](#)
 - LogfTrace, [36](#)
 - LogfWarning, [34](#)
- LogfDebug
 - Log, [35](#)
- LogfError
 - Log, [34](#)
- LogfFatal
 - Log, [34](#)
- LogfInfo
 - Log, [35](#)
- LogfTrace
 - Log, [36](#)
- LogfWarning
 - Log, [34](#)
- LogSettings, [55](#)
- Map, [37](#)
 - Map_Delete, [39](#)
 - Map_Get, [39](#)
 - Map_Set, [38](#)
 - NewMap, [38](#)
- map, [56](#)
- Map_Delete
 - Map, [39](#)
- Map_Get
 - Map, [39](#)
- map_result, [56](#)
- Map_Set
 - Map, [38](#)
- max_connections
 - ServerProperties, [58](#)
- MessageAndClose
 - Connection, [16](#)
- MessageOrClose
 - Connection, [15](#)
- mutex
 - ClientShared, [52](#)
- NAME_MAX_LENGTH
 - Data, [23](#)
- NewMap
 - Map, [38](#)
- NextAvailableConnection
 - Server, [41](#)
- NumberOfFilesInDirectory
 - Files, [30](#)
- port
 - ServerProperties, [57](#)
- printBlue
 - Util, [45](#)
- printGreen
 - Util, [44](#)
- printRed
 - Util, [43](#)
- printYellow
 - Util, [44](#)
- Process, [47](#)
 - active_clients, [49](#)
 - default_settings, [49](#)
 - RunHeadless, [48](#)
 - settings_map, [49](#)
 - users_array, [48](#)
 - users_map, [48](#)
- RandomFlag
 - Util, [47](#)
- RandomFloat
 - Util, [46](#)
- RandomInteger
 - Util, [45](#)
- ReadRegisteredFileIntoUsersMap
 - Files, [28](#)
- ReadSettingsFileIntoSettingsMap
 - Files, [31](#)
- receive_buffer_size
 - ClientShared, [53](#)
 - ServerProperties, [57](#)
- RECORD_COUNT
 - Data, [23](#)
- REGISTERED_FILE
 - Files, [26](#)
- RunHeadless

- Process, [48](#)
- send_buffer_size
 - ClientShared, [52](#)
 - ServerProperties, [57](#)
- Server, [40](#)
 - InitializeServer, [41](#)
 - NextAvailableConnection, [41](#)
 - StartServer, [40](#)
- SERVER_SETTINGS_FILE
 - Files, [26](#)
- ServerProperties, [57](#)
 - active_connections, [58](#)
 - backlog, [57](#)
 - max_connections, [58](#)
 - port, [57](#)
 - receive_buffer_size, [57](#)
 - send_buffer_size, [57](#)
 - time_started, [58](#)
- settings_map
 - Process, [49](#)
- shutting_down
 - ClientShared, [52](#)
- socket
 - Connection, [54](#)
- src/server/Build.c, [61](#)
- src/server/Build.h, [61](#)
- src/server/Connection.c, [61](#)
- src/server/Connection.h, [65](#)
- src/server/Data.c, [66](#)
- src/server/Data.h, [67](#)
- src/server/File.c, [67](#)
- src/server/File.h, [70](#)
- src/server/Log.c, [70](#)
- src/server/Log.h, [71](#)
- src/server/main.c, [72](#)
- src/server/map.c, [72](#)
- src/server/map.h, [74](#)
- src/server/Process.c, [75](#)
- src/server/Process.h, [78](#)
- src/server/Server.c, [78](#)
- src/server/Server.h, [81](#)
- src/server/Util.c, [81](#)
- src/server/Util.h, [82](#)
- StartConnectionThread
 - Connection, [14](#)
- StartServer
 - Server, [40](#)
- StartUpdateThread
 - Connection, [13](#)
- state
 - Connection, [55](#)
- status
 - Connection, [54](#)
- thread_id
 - Connection, [54](#)
- time_connected
 - Connection, [54](#)
- time_started
 - ServerProperties, [58](#)
- UpdateRegisteredFileFromUsersMap
 - Files, [29](#)
- User, [58](#)
- user
 - Connection, [55](#)
- userFullNames
 - Data, [24](#), [25](#)
- users
 - ClientShared, [52](#)
- users_array
 - Process, [48](#)
- users_map
 - Process, [48](#)
- Util, [42](#)
 - COLOR_BLUE, [43](#)
 - COLOR_GREEN, [42](#)
 - COLOR_RED, [42](#)
 - COLOR_RESET, [43](#)
 - COLOR_YELLOW, [43](#)
 - printBlue, [45](#)
 - printGreen, [44](#)
 - printRed, [43](#)
 - printYellow, [44](#)
 - RandomFlag, [47](#)
 - RandomFloat, [46](#)
 - RandomInteger, [45](#)