System Programming

Final Project Report | 2021 - 2022

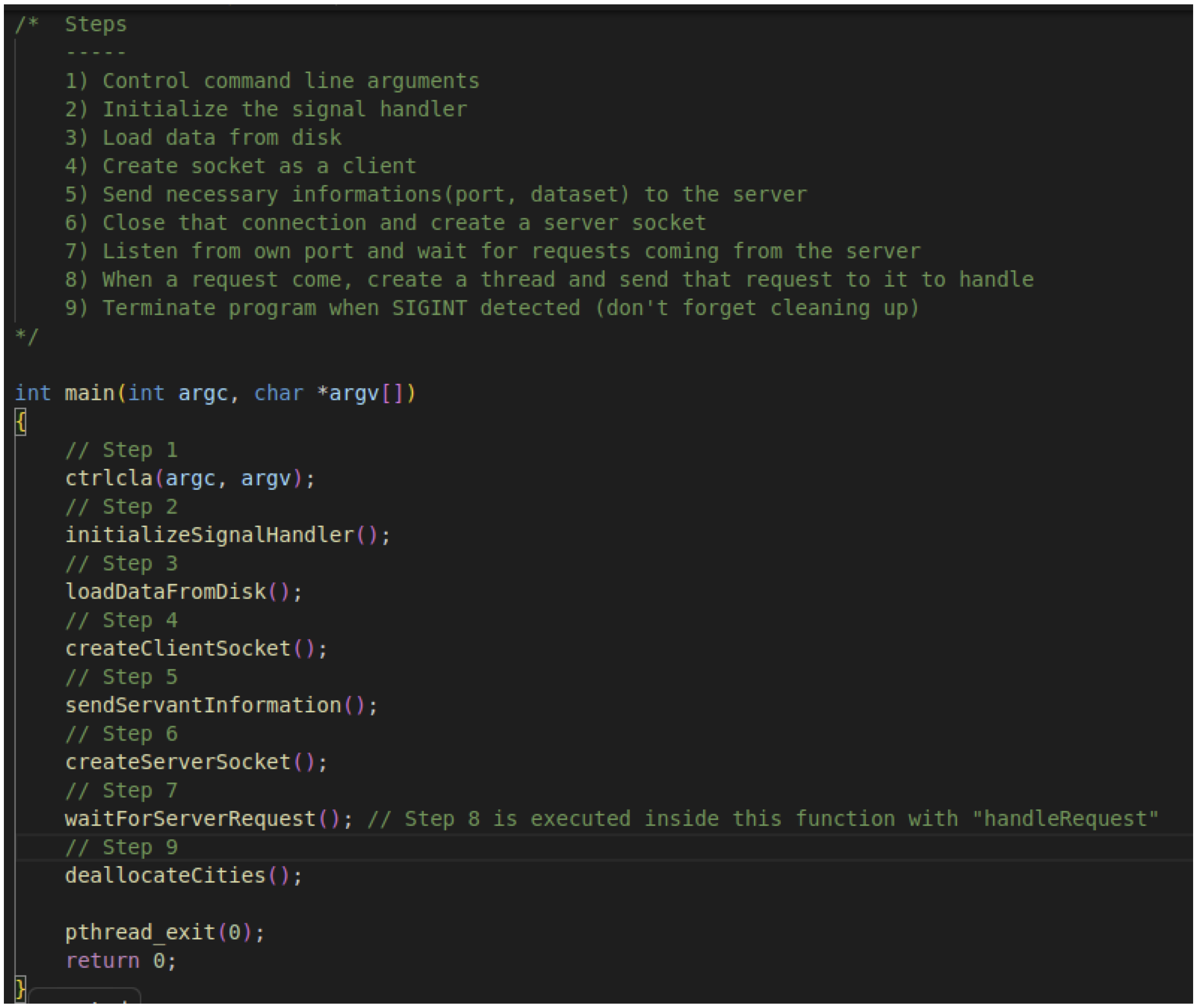
Ahmet Tuğkan Ayhan

1901042692

Abstract

In this final project, we are asked to construct a **server**, which communicates with both **servant**(transaction database) and the **client** processes. Our servant process reads the database and stores all information that belongs to it in it's **heap**. Server process reads the request(s) from client(s) and according to the request, it talks with the servant and after receiving the response from servant process, it response back to the client. Now, I will explain each of the processes(servant, server and client) step by step and how I implemented them. Let's start with the servant process.

Servant

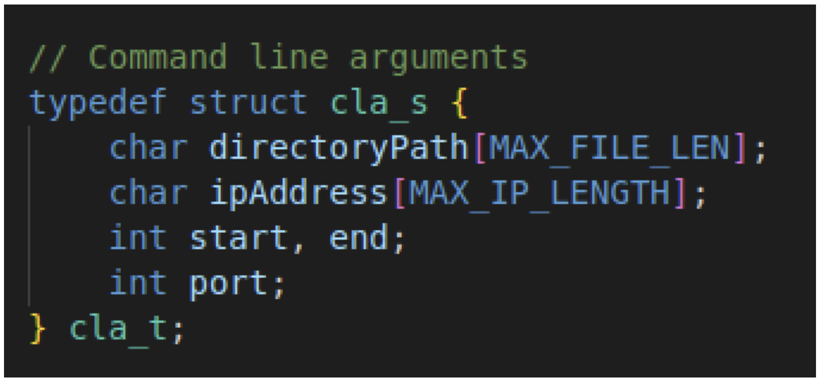


```
/*  Steps
    -----
    1) Control command line arguments
    2) Initialize the signal handler
    3) Load data from disk
    4) Create socket as a client
    5) Send necessary informations(port, dataset) to the server
    6) Close that connection and create a server socket
    7) Listen from own port and wait for requests coming from the server
    8) When a request come, create a thread and send that request to it to handle
    9) Terminate program when SIGINT detected (don't forget cleaning up)
*/

int main(int argc, char *argv[])
{
    // Step 1
    ctrlcla(argc, argv);
    // Step 2
    initializeSignalHandler();
    // Step 3
    loadDataFromDisk();
    // Step 4
    createClientSocket();
    // Step 5
    sendServantInformation();
    // Step 6
    createServerSocket();
    // Step 7
    waitForServerRequest(); // Step 8 is executed inside this function with "handleRequest"
    // Step 9
    deallocateCities();

    pthread_exit(0);
    return 0;
}
```

Step 1 - Controlling Command Line Arguments

Valid Example : “ ./servant -d ../dataset -c 1-9 -r 127.0.0.1 -p $(port) ”

Every process starts with command line arguments control. Here I first control the argc value. For servant process it must be 9. Then, I split these arguments to individuals and assign them to a global variable for later use. To make it clean, I created **args_t** struct for every process. For servant, it looks like this :

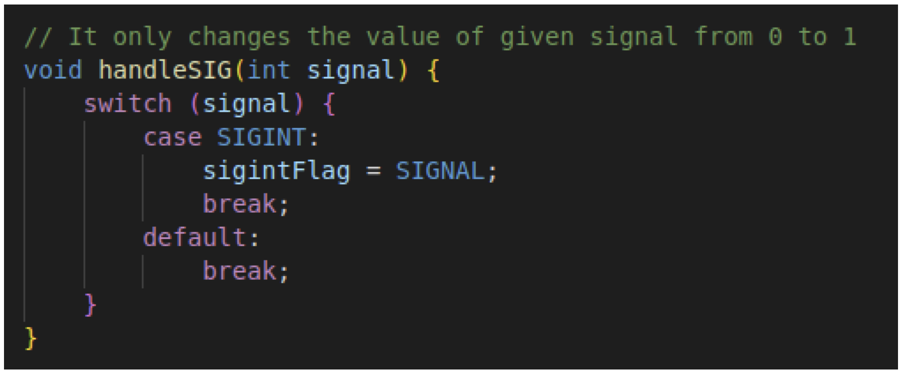


```c
// Command line arguments
typedef struct cla_s {
    char directoryPath[MAX_FILE_LEN];
    char ipAddress[MAX_IP_LENGTH];
    int start, end;
    int port;
} cla_t;
```

After filling this struct object, I finish this step.

Step 2 - Initializing the Signal Handler

For signal handling, I have only 2 function. These functions are **initializeSignalHandler** and **handleSignal.** First function, **initializeSignalHandler,** initializes a sigaction struct object and as a handler function it gives handleSignal to it. Inside of handleSignal is like this :

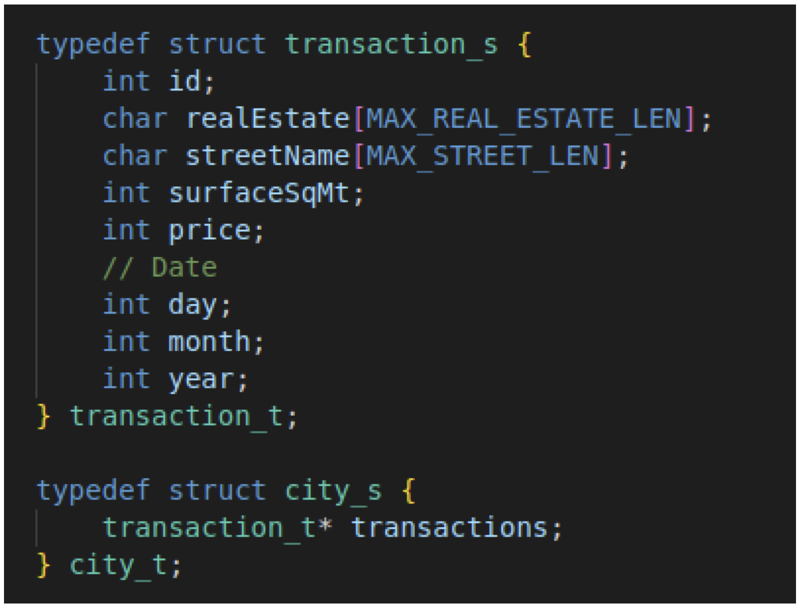


```c
// It only changes the value of given signal from 0 to 1
void handleSIG(int signal) {
    switch (signal) {
        case SIGINT:
            sigintFlag = SIGNAL;
            break;
        default:
            break;
    }
}
```

Here **SIGNAL** is a define value which represents 1. And there is also **NOSIGNAL** value and it represents 0. Inside handler function I only change global **sigintFlag** value from **NOSIGNAL** to **SIGNAL**. After that inside my functions I control these flag to catch **SIGINT**.

Step 3 - Loading Data from the Disk

Since I know which part of the database is belongs to this servant now, I can search through the directory. For this part, I used "dirent.h" library which allows me to list all directories in a sorted order. Then I select the directories which belongs to this servant and using opendir() and readdir() functions I find date files inside the city directories. After reading those date files, I tokenize the lines and insert to my city_t struct. Here is my transaction_t and city_t structs to store database :

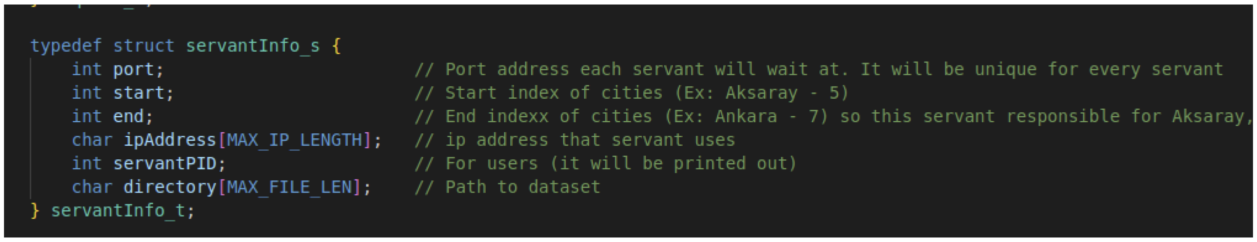


```c
typedef struct transaction_s {
    int id;
    char realEstate[MAX_REAL_ESTATE_LEN];
    char streetName[MAX_STREET_LEN];
    int surfaceSqMt;
    int price;
    // Date
    int day;
    int month;
    int year;
} transaction_t;

typedef struct city_s {
    transaction_t* transactions;
} city_t;
```

All of these struct objects are **dynamically** allocated, so there shouldn't be segmentation fault caused by out of index error.

Step 4 - Creating a Socket as Client

After filling the cities object, which is a **city_t** pointer array declared as a global variable, I open a client socket. This socket sends a request to server to inform the server about port and other informations. Here is the servantInfo_t struct I use :



```c
typedef struct servantInfo_s {
    int port;                      // Port address each servant will wait at. It will be unique for every servant
    int start;                     // Start index of cities (Ex: Aksaray - 5)
    int end;                       // End indexx of cities (Ex: Ankara - 7) so this servant responsible for Aksaray,
    char ipAddress[MAX_IP_LENGTH]; // ip address that servant uses
    int servantPID;                // For users (it will be printed out)
    char directory[MAX_FILE_LEN];  // Path to dataset
} servantInfo_t;
```

After filling this struct, I send it to the server using write function and I close the socket after sending it.

Step 5 - Sending Information to the Server

Explained in previous step

Step 6 - Closing Client Socket and Opening Server

After closing the socket I open another socket as Server and wait at the port I created uniquely for every servant process. To make it different for every servant process, I used command line arguments start and end indexes. These are %100 different for each servant so I sum them up (start + end) and add it to 12000 (since we are wanting a big number for a port value).

Step 7 - Waiting for Request from Server Process

After opening the socket, I used accept function to wait request from the server process. After I receive a connection request from the server, I create a detached thread and give the new socket file descriptor to it.

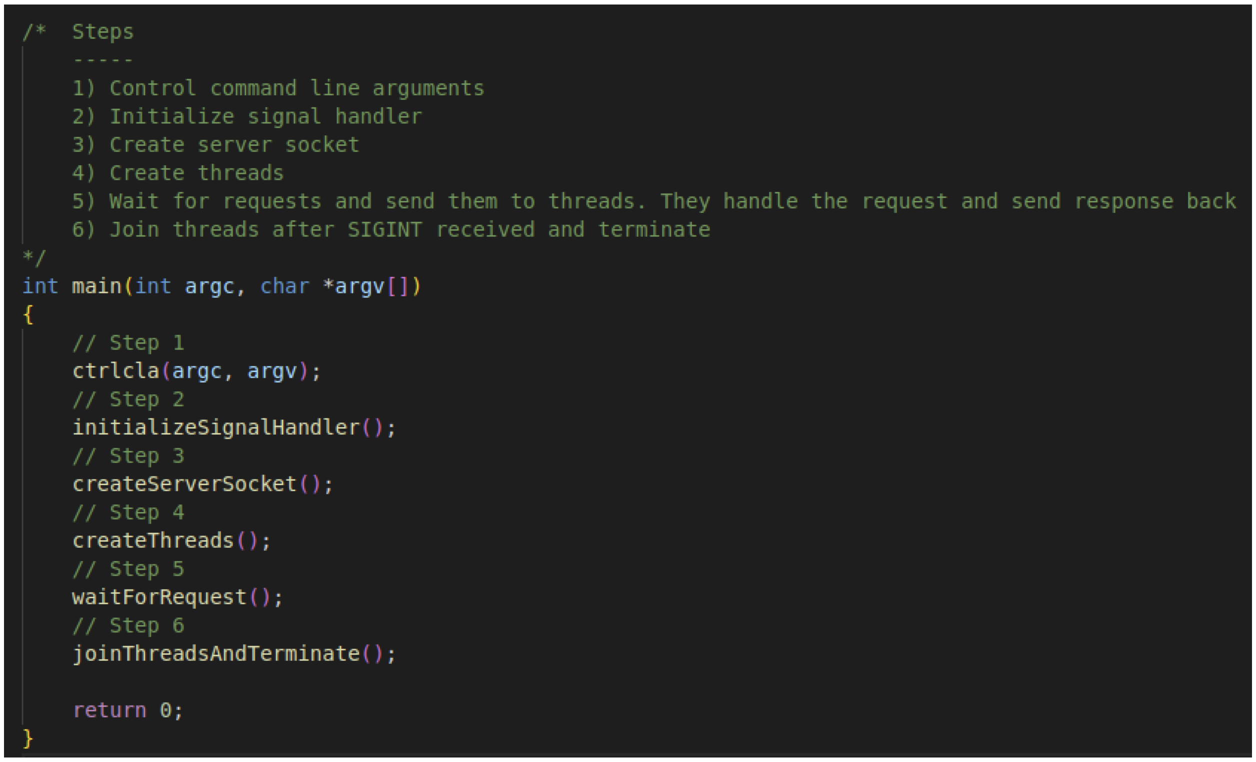
Step 8 - Handling the Server Request

After accepting the connection request and creating the thread I handle the request. To handle it I gave **handleRequest** function to every created thread. Inside this function I call getTransactionCountBetweenDates() function which returns the result that we want. Then I write this value back to server and finish the thread.

Step 9 - Terminating the Program

To terminate program, all we have to do is setting sigintFlag to SIGNAL or 1. When a SIGINT signal received, the process waiting at accept function will be interrupted and directly go sigintFlag if condition. After that it will break the forever for loop and continue in main function. In main function I called 2 last function. First one, deallocateCities. This function deallocates the memory for the variables I have created dynamically. Otherwise, there would be a lot of leak errors. Second and final one is pthread_exit(0). This one to prevent zombie threads. Since I create my threads as detached, I need to be sure that, they all terminated before I terminate main thread. This function waits for all detached threads to terminate.

Server

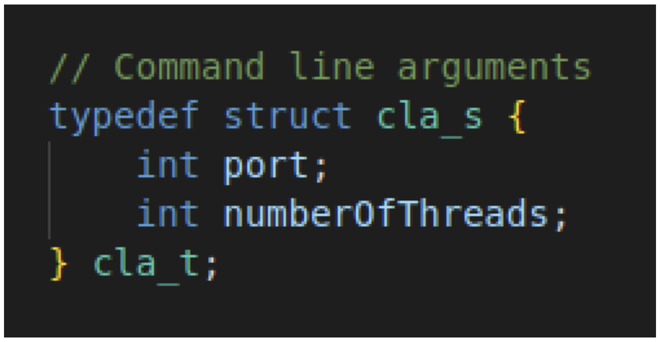


```
/*  Steps
    -----
    1) Control command line arguments
    2) Initialize signal handler
    3) Create server socket
    4) Create threads
    5) Wait for requests and send them to threads. They handle the request and send response back
    6) Join threads after SIGINT received and terminate
*/
int main(int argc, char *argv[])
{
    // Step 1
    ctrlcla(argc, argv);
    // Step 2
    initializeSignalHandler();
    // Step 3
    createServerSocket();
    // Step 4
    createThreads();
    // Step 5
    waitForRequest();
    // Step 6
    joinThreadsAndTerminate();

    return 0;
}
```

Step 1 - Controlling the Command Line Arguments

Valid Example : “ ./server -p $(port) -t 11 ”

Server process also start with similar steps of servant process. Here cla_t struct object is a bit different. Command line argument count(argc), for server process must be 5. If some parts of the command line arguments are invalid, then I inform the user and exit.



```
// Command line arguments
typedef struct cla_s {
    int port;
    int numberOfThreads;
} cla_t;
```

After filling this struct object, I finish this step.

Step 2 - Initializing the Signal Handler

It is exactly same as the servant signal handler.

Step 3 - Creating a Socket as Server

After signal initializing is done, I created a server socket to receive requests from both servant and client.

Step 4 - Creating the Threads

Before accepting requests, I created all of my threads. Number of threads created is given with command line arguments.

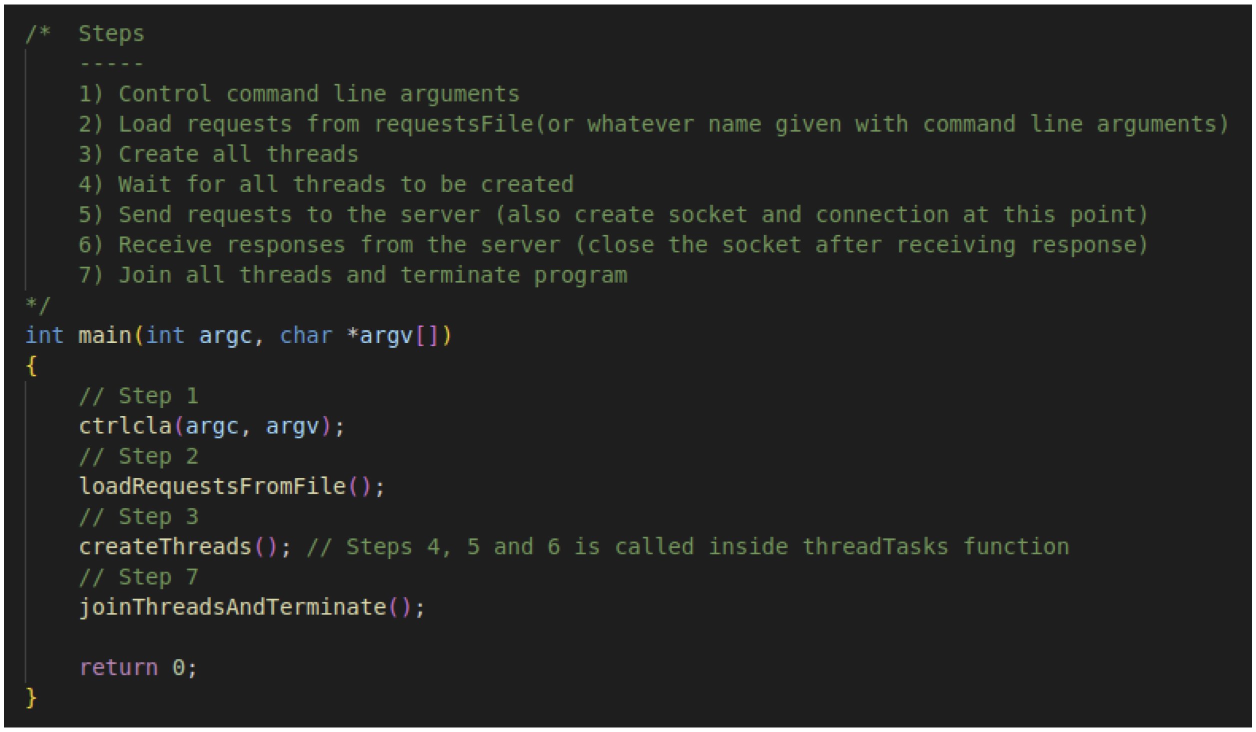
Step 5 - Waiting for Requests from Servant & Client

Here, I started to accept requests from both servant & client. To handle requests, I implemented my own queue library. When a request arrives, I put that request to the queue using enqueue function(while doing this, I used mutexes) and after that I sent a signal using pthread_cond_signal(). One of the idle threads receive that signal and using dequeue function reads the request. It handles it according the size of the request and returns the response to the client or saves the data of servant according to the request.

Step 6 - Joining Threads and Terminating Program

Whenever a SIGINT signal is received, signal handler function changes sigintFlag to SIGNAL and in next sigintFlag condition check, forever loop breaks. After main thread breaks the infinite loop it broadcast to all threads and these threads also break their infinite loop. After that, I used pthread_join until every child thread terminates. Before terminating the program, I deallocated the spaces I used dynamically.

Client

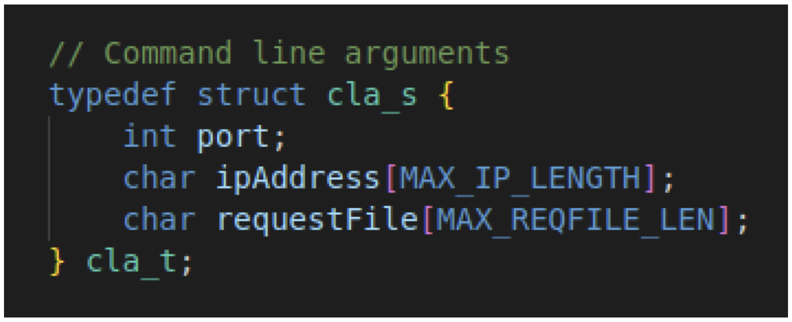


```c
/*  Steps
    -----
    1) Control command line arguments
    2) Load requests from requestsFile(or whatever name given with command line arguments)
    3) Create all threads
    4) Wait for all threads to be created
    5) Send requests to the server (also create socket and connection at this point)
    6) Receive responses from the server (close the socket after receiving response)
    7) Join all threads and terminate program
*/
int main(int argc, char *argv[])
{
    // Step 1
    ctrlcla(argc, argv);
    // Step 2
    loadRequestsFromFile();
    // Step 3
    createThreads(); // Steps 4, 5 and 6 is called inside threadTasks function
    // Step 7
    joinThreadsAndTerminate();

    return 0;
}
```

Step 1 - Controlling the Command Line Arguments

Valid Example : ./client -r ../requestFile -q $(port) -s 127.0.0.1

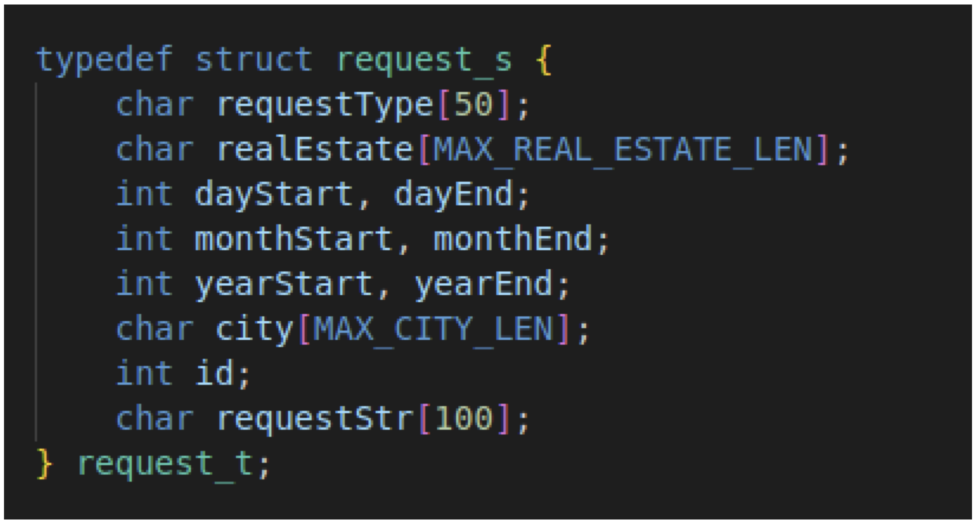
Lastly, I will talk about client process. Like any other process in this project, this one also starts with argument control. For client process, argument count must be 7. cla_t struct looks like this :



```c
// Command line arguments
typedef struct cla_s {
    int port;
    char ipAddress[MAX_IP_LENGTH];
    char requestFile[MAX_REQFILE_LEN];
} cla_t;
```

Step 2 - Loading Requests from the Request File

Reading the request file is simple. First read the command line arguments, assign them to their global **cla_t** struct, then using **requestFile** char array variable open the file. Read the content using **readline() (**this is a function which reads a line per call**)** until it returns -1 (which is EOF). After getting each line, using **strtok**, tokenize them and store them in a **request_t** struct object. These objects then stored in a dynamic array. Here is the request_t struct :



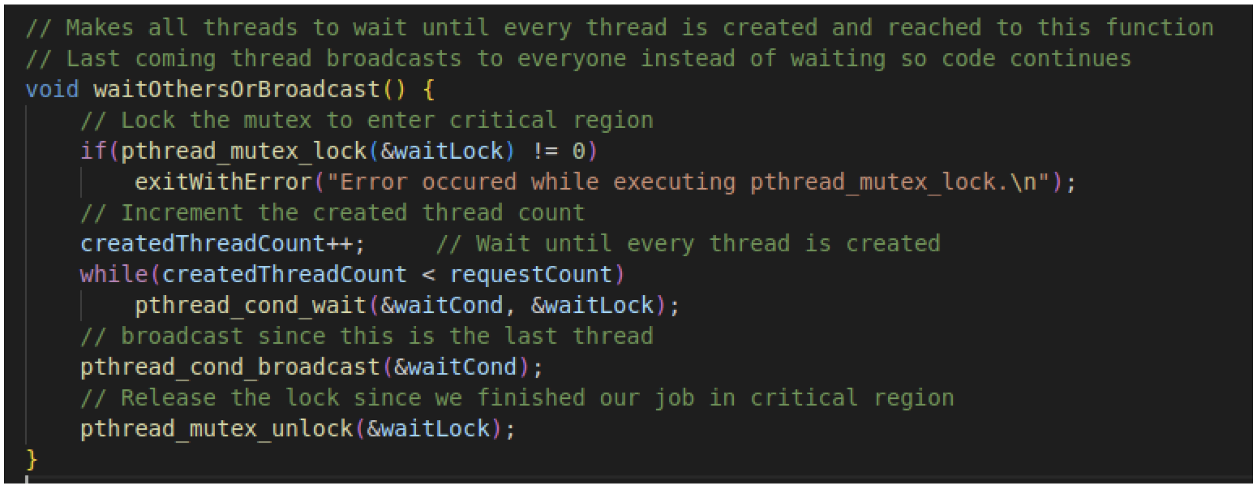
```c
typedef struct request_s {
    char requestType[50];
    char realEstate[MAX_REAL_ESTATE_LEN];
    int dayStart, dayEnd;
    int monthStart, monthEnd;
    int yearStart, yearEnd;
    char city[MAX_CITY_LEN];
    int id;
    char requestStr[100];
} request_t;
```

Step 3 - Creating Threads

After filling the request_t object array, I created all threads using **pthread_create()**.

Step 4 - Waiting for All Threads to be Created

Here, I used a mutex and a condition variable (which together they become a monitor). To make all threads to wait I implemented a simple function like this :



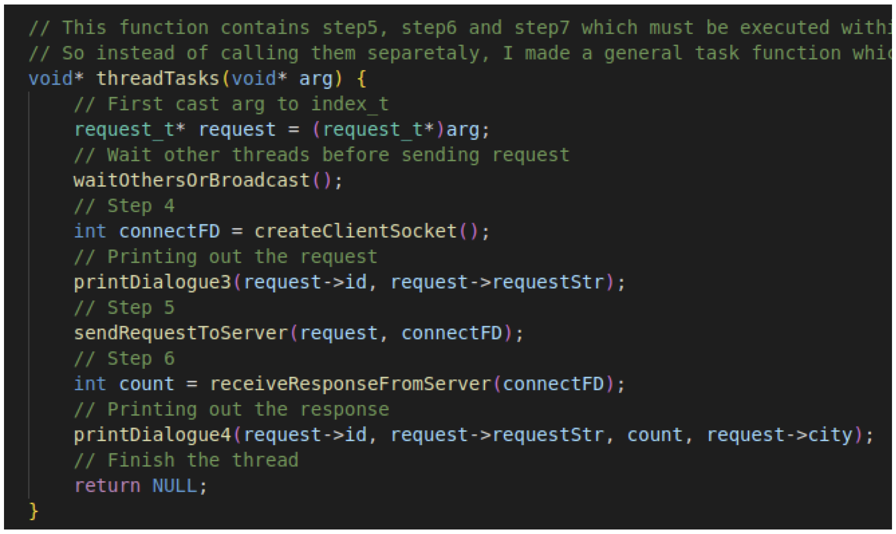
```c
// Makes all threads to wait until every thread is created and reached to this function
// Last coming thread broadcasts to everyone instead of waiting so code continues
void waitOthersOrBroadcast() {
    // Lock the mutex to enter critical region
    if(pthread_mutex_lock(&waitLock) != 0)
        exitWithError("Error occured while executing pthread_mutex_lock.\n");
    // Increment the created thread count
    createdThreadCount++;     // Wait until every thread is created
    while(createdThreadCount < requestCount)
        pthread_cond_wait(&waitCond, &waitLock);
    // broadcast since this is the last thread
    pthread_cond_broadcast(&waitCond);
    // Release the lock since we finished our job in critical region
    pthread_mutex_unlock(&waitLock);
}
```

First I get the lock and incremented the createdThreadCount. Then if created thread is not the last one, I called pthread_cond_wait function. Last thread created calls pthread_cond_broadcast instead of pthread_cond_wait. So all threads are released approximately at the same moment.

Step 5 - Create Socket and Send Requests to the Server

After threads are released, they create their client sockets and sends their requests to the server.

Step 6 - Receive The Response From The Server



```c
// This function contains step5, step6 and step7 which must be executed withi
// So instead of calling them separetaly, I made a general task function whic
void* threadTasks(void* arg) {
    // First cast arg to index_t
    request_t* request = (request_t*)arg;
    // Wait other threads before sending request
    waitOthersOrBroadcast();
    // Step 4
    int connectFD = createClientSocket();
    // Printing out the request
    printDialogue3(request->id, request->requestStr);
    // Step 5
    sendRequestToServer(request, connectFD);
    // Step 6
    int count = receiveResponseFromServer(connectFD);
    // Printing out the response
    printDialogue4(request->id, request->requestStr, count, request->city);
    // Finish the thread
    return NULL;
}
```

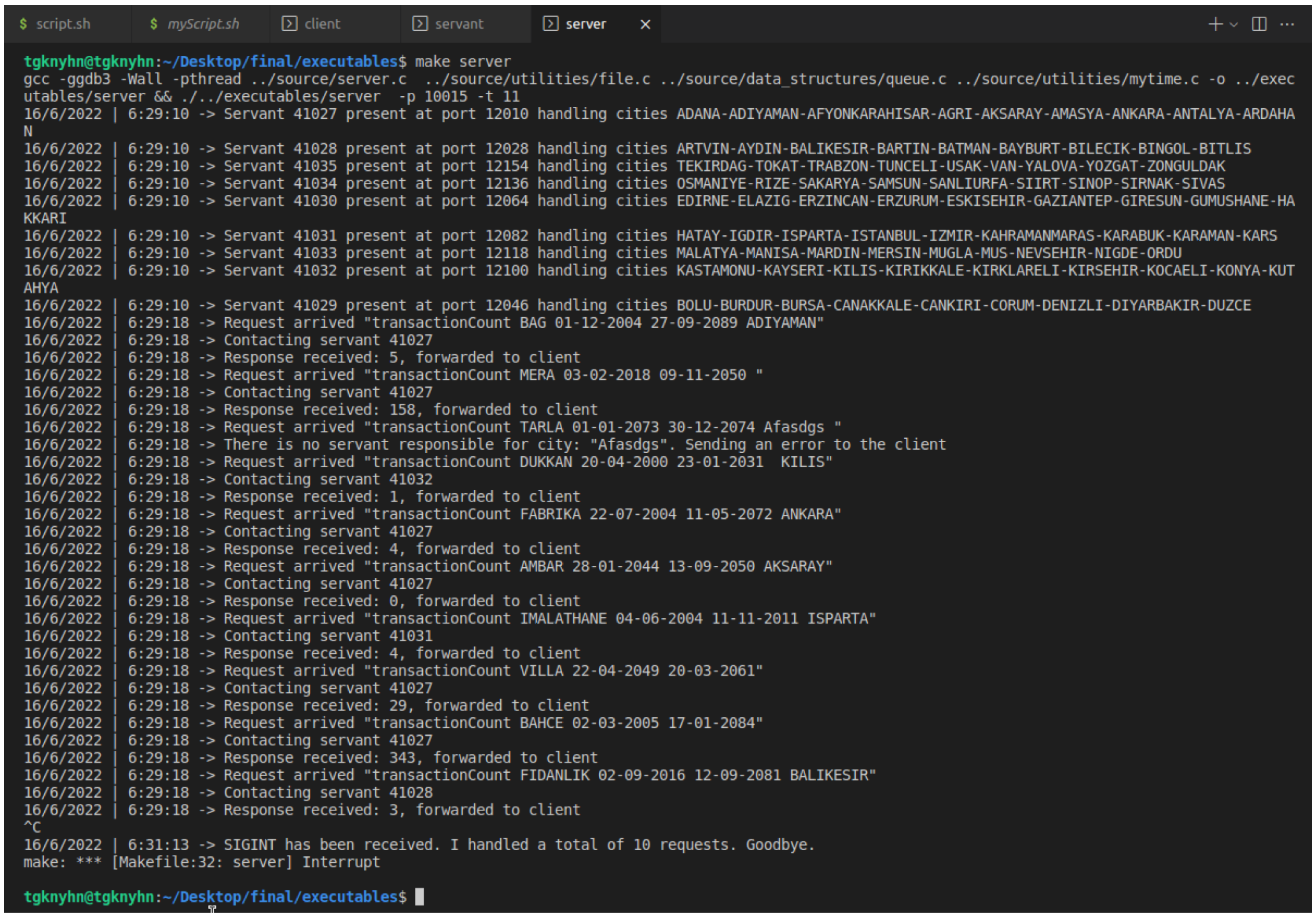
Here we can see what is happening inside a thread function. After they are released they call createClientSocket function. Right after that they send their requests to the server using sendRequestToServer function. When the response is ready, buffer is read by receiveResponseFromServer and after casting it into an int variable, count value is returned. Finally it is printed to the console.

Step 7 - Join All Threads and Terminate

After every client thread receives their response, main thread calls pthread_join function to wait for them. After every child thread terminated, it deallocates all allocated memory and terminates.

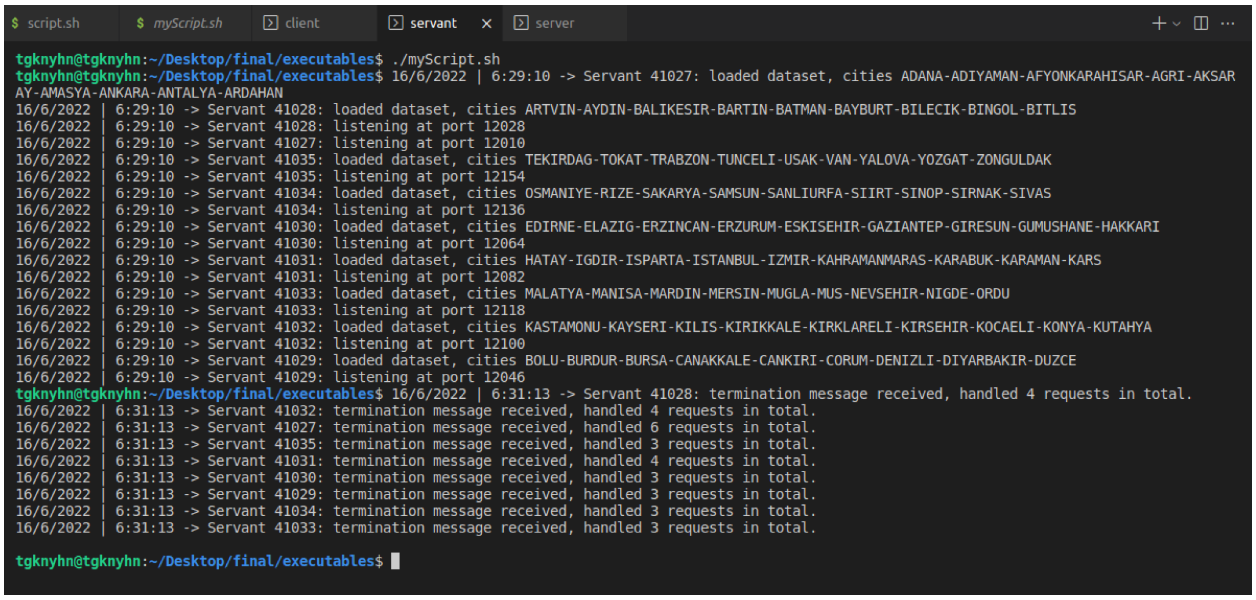
Example Output

Server



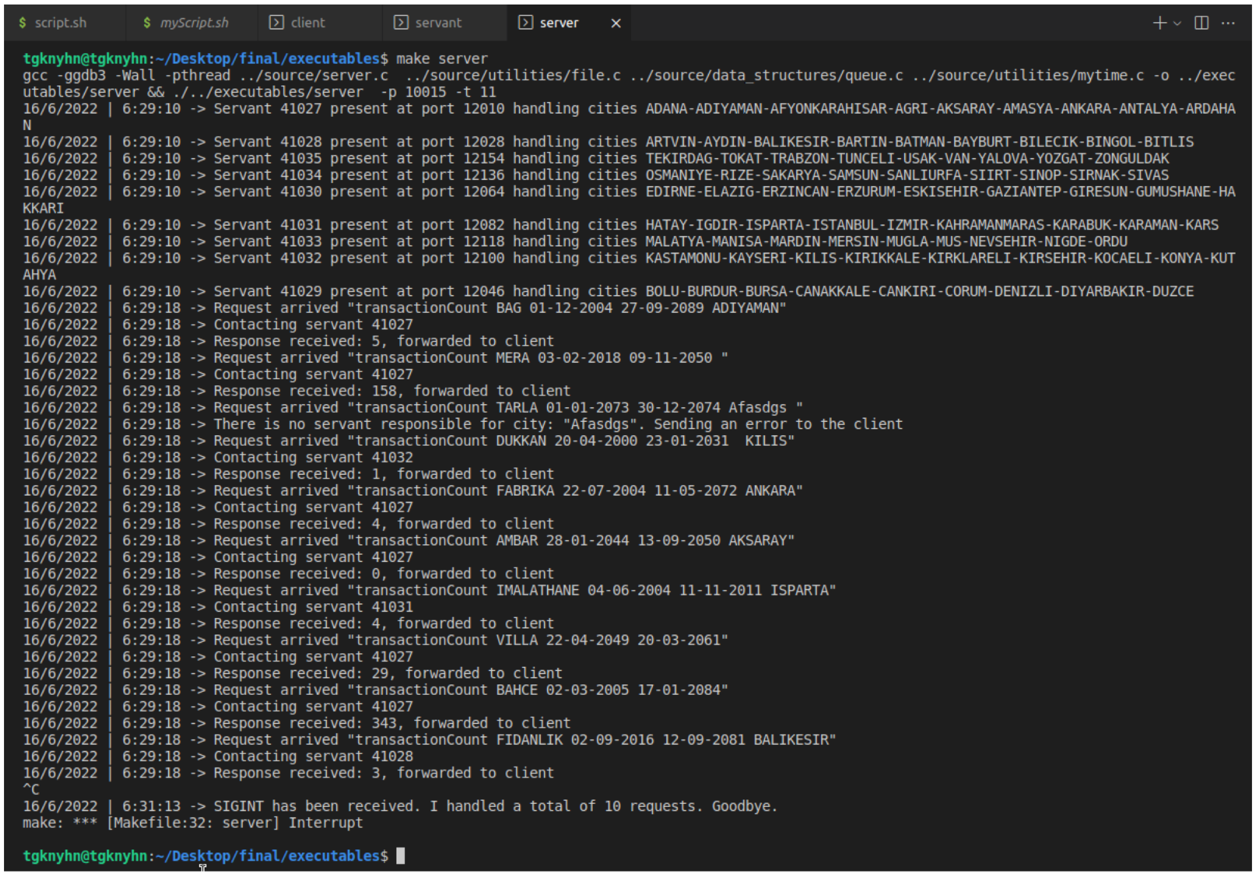
```
tgknyhn@tgknyhn:~/Desktop/final/executables$ make server
gcc -ggdb3 -Wall -pthread ../source/server.c  ../source/utilities/file.c ../source/data_structures/queue.c ../source/utilities/mytime.c -o ../exec
utables/server && ./../executables/server  -p 10015 -t 11
16/6/2022 | 6:29:10 -> Servant 41027 present at port 12010 handling cities ADANA-ADIYAMAN-AFYONKARAHISAR-AGRI-AKSARAY-AMASYA-ANKARA-ANTALYA-ARDAHA
N
16/6/2022 | 6:29:10 -> Servant 41028 present at port 12028 handling cities ARTVIN-AYDIN-BALIKESIR-BARTIN-BATMAN-BAYBURT-BILECIK-BINGOL-BITLIS
16/6/2022 | 6:29:10 -> Servant 41035 present at port 12154 handling cities TEKIRDAG-TOKAT-TRABZON-TUNCELI-USAK-VAN-YALOVA-YOZGAT-ZONGULDAK
16/6/2022 | 6:29:10 -> Servant 41034 present at port 12136 handling cities OSMANIYE-RIZE-SAKARYA-SAMSUN-SANLIURFA-SIIRT-SINOP-SIRNAK-SIVAS
16/6/2022 | 6:29:10 -> Servant 41030 present at port 12064 handling cities EDIRNE-ELAZIG-ERZINCAN-ERZURUM-ESKISEHIR-GAZIANTEP-GIRESUN-GUMUSHANE-HA
KKARI
16/6/2022 | 6:29:10 -> Servant 41031 present at port 12082 handling cities HATAY-IGDIR-ISPARTA-ISTANBUL-IZMIR-KAHRAMANMARAS-KARABUK-KARAMAN-KARS
16/6/2022 | 6:29:10 -> Servant 41033 present at port 12118 handling cities MALATYA-MANISA-MARDIN-MERSIN-MUGLA-MUS-NEVSEHIR-NIGDE-ORDU
16/6/2022 | 6:29:10 -> Servant 41032 present at port 12100 handling cities KASTAMONU-KAYSERI-KILIS-KIRIKKALE-KIRKLARELI-KIRSEHIR-KOCAELI-KONYA-KUT
AHYA
16/6/2022 | 6:29:10 -> Servant 41029 present at port 12046 handling cities BOLU-BURDUR-BURSA-CANAKKALE-CANKIRI-CORUM-DENIZLI-DIYARBAKIR-DUZCE
16/6/2022 | 6:29:18 -> Request arrived "transactionCount BAG 01-12-2004 27-09-2089 ADIYAMAN"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 5, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount MERA 03-02-2018 09-11-2050 "
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 158, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount TARLA 01-01-2073 30-12-2074 Afasdgs "
16/6/2022 | 6:29:18 -> There is no servant responsible for city: "Afasdgs". Sending an error to the client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount DUKKAN 20-04-2000 23-01-2031  KILIS"
16/6/2022 | 6:29:18 -> Contacting servant 41032
16/6/2022 | 6:29:18 -> Response received: 1, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount FABRIKA 22-07-2004 11-05-2072 ANKARA"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 4, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount AMBAR 28-01-2044 13-09-2050 AKSARAY"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 0, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount IMALATHANE 04-06-2004 11-11-2011 ISPARTA"
16/6/2022 | 6:29:18 -> Contacting servant 41031
16/6/2022 | 6:29:18 -> Response received: 4, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount VILLA 22-04-2049 20-03-2061"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 29, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount BAHCE 02-03-2005 17-01-2084"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 343, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount FIDANLIK 02-09-2016 12-09-2081 BALIKESIR"
16/6/2022 | 6:29:18 -> Contacting servant 41028
16/6/2022 | 6:29:18 -> Response received: 3, forwarded to client
^C
16/6/2022 | 6:31:13 -> SIGINT has been received. I handled a total of 10 requests. Goodbye.
make: *** [Makefile:32: server] Interrupt

tgknyhn@tgknyhn:~/Desktop/final/executables$ 
```

Servant



```
tgknyhn@tgknyhn:~/Desktop/final/executables$ ./myScript.sh
tgknyhn@tgknyhn:~/Desktop/final/executables$ 16/6/2022 | 6:29:10 -> Servant 41027: loaded dataset, cities ADANA-ADIYAMAN-AFYONKARAHISAR-AGRI-AKSAR
AY-AMASYA-ANKARA-ANTALYA-ARDAHAN
16/6/2022 | 6:29:10 -> Servant 41028: loaded dataset, cities ARTVIN-AYDIN-BALIKESIR-BARTIN-BATMAN-BAYBURT-BILECIK-BINGOL-BITLIS
16/6/2022 | 6:29:10 -> Servant 41028: listening at port 12028
16/6/2022 | 6:29:10 -> Servant 41027: listening at port 12010
16/6/2022 | 6:29:10 -> Servant 41035: loaded dataset, cities TEKIRDAG-TOKAT-TRABZON-TUNCELI-USAK-VAN-YALOVA-YOZGAT-ZONGULDAK
16/6/2022 | 6:29:10 -> Servant 41035: listening at port 12154
16/6/2022 | 6:29:10 -> Servant 41034: loaded dataset, cities OSMANIYE-RIZE-SAKARYA-SAMSUN-SANLIURFA-SIIRT-SINOP-SIRNAK-SIVAS
16/6/2022 | 6:29:10 -> Servant 41034: listening at port 12136
16/6/2022 | 6:29:10 -> Servant 41030: loaded dataset, cities EDIRNE-ELAZIG-ERZINCAN-ERZURUM-ESKISEHIR-GAZIANTEP-GIRESUN-GUMUSHANE-HAKKARI
16/6/2022 | 6:29:10 -> Servant 41030: listening at port 12064
16/6/2022 | 6:29:10 -> Servant 41031: loaded dataset, cities HATAY-IGDIR-ISPARTA-ISTANBUL-IZMIR-KAHRAMANMARAS-KARABUK-KARAMAN-KARS
16/6/2022 | 6:29:10 -> Servant 41031: listening at port 12082
16/6/2022 | 6:29:10 -> Servant 41033: loaded dataset, cities MALATYA-MANISA-MARDIN-MERSIN-MUGLA-MUS-NEVSEHIR-NIGDE-ORDU
16/6/2022 | 6:29:10 -> Servant 41033: listening at port 12118
16/6/2022 | 6:29:10 -> Servant 41032: loaded dataset, cities KASTAMONU-KAYSERI-KILIS-KIRIKKALE-KIRKLARELI-KIRSEHIR-KOCAELI-KONYA-KUTAHYA
16/6/2022 | 6:29:10 -> Servant 41032: listening at port 12100
16/6/2022 | 6:29:10 -> Servant 41029: loaded dataset, cities BOLU-BURDUR-BURSA-CANAKKALE-CANKIRI-CORUM-DENIZLI-DIYARBAKIR-DUZCE
16/6/2022 | 6:29:10 -> Servant 41029: listening at port 12046
tgknyhn@tgknyhn:~/Desktop/final/executables$ 16/6/2022 | 6:31:13 -> Servant 41028: termination message received, handled 4 requests in total.
16/6/2022 | 6:31:13 -> Servant 41032: termination message received, handled 4 requests in total.
16/6/2022 | 6:31:13 -> Servant 41027: termination message received, handled 6 requests in total.
16/6/2022 | 6:31:13 -> Servant 41035: termination message received, handled 3 requests in total.
16/6/2022 | 6:31:13 -> Servant 41031: termination message received, handled 4 requests in total.
16/6/2022 | 6:31:13 -> Servant 41030: termination message received, handled 3 requests in total.
16/6/2022 | 6:31:13 -> Servant 41029: termination message received, handled 3 requests in total.
16/6/2022 | 6:31:13 -> Servant 41034: termination message received, handled 3 requests in total.
16/6/2022 | 6:31:13 -> Servant 41033: termination message received, handled 3 requests in total.

tgknyhn@tgknyhn:~/Desktop/final/executables$
```

Client



```
tgknyhn@tgknyhn:~/Desktop/final/executables$ make server
gcc -ggdb3 -Wall -pthread ../source/server.c  ../source/utilities/file.c ../source/data_structures/queue.c ../source/utilities/mytime.c -o ../exec
utables/server && ./../executables/server  -p 10015 -t 11
16/6/2022 | 6:29:10 -> Servant 41027 present at port 12010 handling cities ADANA-ADIYAMAN-AFYONKARAHISAR-AGRI-AKSARAY-AMASYA-ANKARA-ANTALYA-ARDAHA
N
16/6/2022 | 6:29:10 -> Servant 41028 present at port 12028 handling cities ARTVIN-AYDIN-BALIKESIR-BARTIN-BATMAN-BAYBURT-BILECIK-BINGOL-BITLIS
16/6/2022 | 6:29:10 -> Servant 41035 present at port 12154 handling cities TEKIRDAG-TOKAT-TRABZON-TUNCELI-USAK-VAN-YALOVA-YOZGAT-ZONGULDAK
16/6/2022 | 6:29:10 -> Servant 41034 present at port 12136 handling cities OSMANIYE-RIZE-SAKARYA-SAMSUN-SANLIURFA-SIIRT-SINOP-SIRNAK-SIVAS
16/6/2022 | 6:29:10 -> Servant 41030 present at port 12064 handling cities EDIRNE-ELAZIG-ERZINCAN-ERZURUM-ESKISEHIR-GAZIANTEP-GIRESUN-GUMUSHANE-HA
KKARI
16/6/2022 | 6:29:10 -> Servant 41031 present at port 12082 handling cities HATAY-IGDIR-ISPARTA-ISTANBUL-IZMIR-KAHRAMANMARAS-KARABUK-KARAMAN-KARS
16/6/2022 | 6:29:10 -> Servant 41033 present at port 12118 handling cities MALATYA-MANISA-MARDIN-MERSIN-MUGLA-MUS-NEVSEHIR-NIGDE-ORDU
16/6/2022 | 6:29:10 -> Servant 41032 present at port 12100 handling cities KASTAMONU-KAYSERI-KILIS-KIRIKKALE-KIRKLARELI-KIRSEHIR-KOCAELI-KONYA-KUT
AHYA
16/6/2022 | 6:29:10 -> Servant 41029 present at port 12046 handling cities BOLU-BURDUR-BURSA-CANAKKALE-CANKIRI-CORUM-DENIZLI-DIYARBAKIR-DUZCE
16/6/2022 | 6:29:18 -> Request arrived "transactionCount BAG 01-12-2004 27-09-2089 ADIYAMAN"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 5, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount MERA 03-02-2018 09-11-2050 "
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 158, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount TARLA 01-01-2073 30-12-2074 Afasdgs "
16/6/2022 | 6:29:18 -> There is no servant responsible for city: "Afasdgs". Sending an error to the client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount DUKKAN 20-04-2000 23-01-2031  KILIS"
16/6/2022 | 6:29:18 -> Contacting servant 41032
16/6/2022 | 6:29:18 -> Response received: 1, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount FABRIKA 22-07-2004 11-05-2072 ANKARA"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 4, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount AMBAR 28-01-2044 13-09-2050 AKSARAY"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 0, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount IMALATHANE 04-06-2004 11-11-2011 ISPARTA"
16/6/2022 | 6:29:18 -> Contacting servant 41031
16/6/2022 | 6:29:18 -> Response received: 4, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount VILLA 22-04-2049 20-03-2061"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 29, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount BAHCE 02-03-2005 17-01-2084"
16/6/2022 | 6:29:18 -> Contacting servant 41027
16/6/2022 | 6:29:18 -> Response received: 343, forwarded to client
16/6/2022 | 6:29:18 -> Request arrived "transactionCount FIDANLIK 02-09-2016 12-09-2081 BALIKESIR"
16/6/2022 | 6:29:18 -> Contacting servant 41028
16/6/2022 | 6:29:18 -> Response received: 3, forwarded to client
^C
16/6/2022 | 6:31:13 -> SIGINT has been received. I handled a total of 10 requests. Goodbye.
make: *** [Makefile:32: server] Interrupt

tgknyhn@tgknyhn:~/Desktop/final/executables$
```