CSE344 Final Report

Yeşim Yalçın-200104004094

Contents

1-) Client
A-) Command Line Arguments
B-) Request Retrieval
C-) Thread Creation
D-) Threads

2-) Server
A-) Command Line Arguments
B-) Thread Creation
C-) Server Socket Creation and Request Handling
D-) Thread Request Handling
E-) SIGINT

3-) Servant
A-) Command Line Arguments
B-) Dataset Retrieval
C-) Port Number Determination
E-) Connection with Server Socket & Request Handling
F-) Thread Request Handling
H-) SIGINT

4-) Parts I Have Completed

5-) Documentation

1-) Client

A-) Command Line Arguments
I start with checking if the given command line argument count is correct. If needed amount is provided then I check if they are valid.
I collect the valid arguments and continue. If there is any invalid argument, an exception happens.

B-) Request Retrieval
I open the given request file and collect requests one by one. Put them in a requests array to later distribute to threads.
If the request file contains any extra spaces or newlines, they will be ignored. However, aside from those, it is expected that there is a valid request file.

C-) Thread Creation
After the requests are retrieved, the count of how many requests will be sent will be known as well. That many threads are created an each are given one request through their thread function parameters.
After client creates the threads, it proceeds to wait till all threads are finished thenit exits as well.

D-) Threads
When a thread is created, it firstly checks if all the other threads are created as well. This is done by a mutex and a global “arrived” counter variable.
After all threads are created, they try to establish connection with server socket.
If they successfuly connect to it, they write their requests and wait for a response.
When they get a response, they print them out and terminate.
If they get a response of -1 it means the searched city was not in the dataset of servants.

2-) Server

A-) Command Line Arguments
I start with checking if the given command line argument count is correct. If needed amount is provided then I check if they are valid.
I collect the valid arguments and continue. If there is any invalid argument, an exception happens.

B-) Thread Creation
Server continues with creating given amount of threads. When the server needs to terminate, it makes sure to join all of it’s threads.

C-) Server Socket Creation and Request Handling
After creating all of it’s threads, the server opens it’s port with address “127.0.0.1”. I have given this address statically. When the initialization of the port is done with socket, bind, listen, the server starts waiting for requests.
I have given 1024 as a request queue count in listen.

When a request arrives, the server puts the retrieved file descripter from accept to the request queue.

This queue is a global queue to store requests. It is like a circular array. The currentIndex points to the followin request that needs to be handled. TotalRequests points to the latest inserted request.

The server firstly checks if the queue is full. If so, it waits till there is a space. When there is space, it inserts the file descriptor and returns to waiting for requests.
These operations are done with 1 critical region mutex and 2 condition variables. 1 condition is to check if the queue is full, the other is to check if the queue is empty.
When the server inserts a file descriptor into the queue, it wakes up all threads waiting on the empty condition.

D-) Thread Request Handling
When the threads are created, they firstly wait on the empty condition. Whenever there is a request in the queue, one thread gets it and starts handling it.
To handle a request, the thread firstly reads the request from the file descriptor it is provided and then takes action accordingly.

If the request starts with the letter c, it means it is coming from a client. If the request starts with the letter s, it means it is coming from a servant.

If it is a servant request, the thread saves that servant's information and returns back to waiting for requests.
If it is a client request, it checks if there is any servant able to respond to it. If so, sends the request to the servant through it's port. If not, returns -1 to the client.
If there is no city specified in the request, the request is sent to all registered servants.

After sending the request, the thread waits for a response and when it gets it, it delivers it to the client through the file descriptor it had retrieved.

F-) SIGINT
For SIGINT, I have a signal handler which sets the global sigintFlag as 1. This sigintFlag is checked in some specific places of the server & server thread code. Especially at the blocking places. Whenever a thread or the server sees this flag as 1, it makes sure all the other threads see it as well by unlocking mutexes, broadcasting necessary conditions.
After all the threads are terminated, the server sends SIGINT to all servants as well.

3-) Servant

A-) Command Line Arguments
I start with checking if the given command line argument count is correct. If needed amount is provided then I check if they are valid.
I collect the valid arguments and continue. If there is any invalid argument, an exception happens.

B-) Dataset Retrieval
To retrieve the dataset, I firstly determine which cities the servant is responsible for. I do that by looking at all the city names in the dataset directory. I order the names and find the ones which the servant needs.
Later on I iterate through the servant's city directories and retrieve all transactions.
I store the transactions in an AVL Tree ordered by their city names.

I chose AVL Tree because using a regular binary tree would not be balanced in a servant with alphabetically ordered city names
.

C-) Port Number Determination

To assing each servant a unique port number, I use a shared memory. When the first servant starts, it takes the portnumber given in arguments + 10 as it's port number. Initializes a shared memory and writes it's port number here. When another servant stars, it checks this memory segment and assigns it's port number as content of shared memory segment +10. So, no two servants share the same port.

E-) Connection with Server Socket & Request Handling

The servant establishes a connection via the server and waits for a request to arrive. If a request arrives, creates a thread and gives the retrieved socket file descriptor to it.
I chose to make these threads detahced because the threads run for only 1 request and there might be hundreds of threads running at the same time. Having them detached made it easier to handle them and their resources.

F-) Thread Request Handling

After a thread is created, it reads the request from the given socket file descriptor then according to the request, looks at the AVL Tree and calculates the result. In the end sends the result to the server and terminates.

H-) SIGINT

For SIGINT, I have a signal handler which sets the global sigintFlag as 1. This sigintFlag is checked in some specific places of the servant code. If the servant sees it as 1, it terminates. However, it terminates with pthread_exit to be able to clean up after it's detached threads. Detached threads clean up after themselves anyway but exiting with pthread_exit on the main process, helps them not to do it forcefully.

4-) Parts I Have Completed

I believe I have completed all the parts.

5-) Documentation

I have included details related to functions in header files. I will be adding them here as well.

```c
//Retrieves the required information from the arguments.
//Returns 0 on success, -1 on failure
//The retrieved info are assigned to path, start, end, ip, portNo
int retrieveArguments(int count, char* arguements[], char** path, int* start, int* end, char** ip, int* portNo);

//Retrieves the servant's assigned citys' numebrs from the given numbersString
//Assigns them to start and end
void retrieveCityNumbers(char* numbersString, int* start, int* end);

//Compares two city names.
//Returns a value >0 if a>b
//Returns 0 if a=b
//Returns a value <0 is a<b
int compareCityNames(const void * a, const void * b);

//Arranges the given currentservant in a one line string format.
//Returns the pointer on success, NULL on failure
//Also assigns the new input to the first parameter
char* arrangeSocketInput(char** socketInput, struct servantInfo currentservant);

//Formats the data loaded output of servant
//The output is put inside the first parameter
//pid is the pid of the servant
//The servant will be responsible of cities between firstCityName and secondCityName
void arrangeDataLoadedOutput(char* output, int servantPid, char* firstCityName, char* secondCityName);

//Formats the servant start output of servant
//The output is put inside the first parameter
//port is the port number of the server
//pid is the pid of the servant
void arrangeServantStartOutput(char* output, int pid, int port);

//Thread function of servant. Deals with the given connection.
//args contain a file descriptor for server socket
void* threadFunction(void* args);
```

```c
//Adds a new transaction to a dynamic transaction array.
//Returns the pointer on success, NULL on failure
//Increments the size of the string given in 3rd parameter
//Assigns the new transaction to the second parameter as well
struct transactions* addTransaction(struct transactions currentTransaction, struct transactions** allTransactions, int *transactionCount);

//Checks if the given dateE lies in between startT and endD
//Returns 1 if true, 0 if false
int compareDates(char* endD, char* startT, char* dateE);

//Calculates if the date with given d1, m1, y1 is bigger than given d2, m2, y2
//If true, returns 1. If false, return -1
int calculateDateDifference(int d1, int m1, int y1, int d2, int m2, int y2);

//When a SIGINT is received, this handler is run.
//It assigns the sigintFlag as 1
void sigintHandler(int signum);

//Formats the termination output of servant
//The output is put inside the first parameter
//pid is the servant's pid
void arrangeTerminationOutput(char* output, int pid);

//Used during atexit
//Frees some globally used pointers
void exitFunction(void);
```

```c
//requiredData structure is used to pass information to a created thread.
//threadRequest is the request that the threads is assigned to
struct requiredData{
    struct request threadRequest;
    int threadNo;
};

//Retrieves the required information from the arguments.
//Returns 0 on success, -1 on failure
//The retrieved info are assigned to path, no and ip
//Path is the request file datapath, no is port no, ip is server ip
int retrieveArguments(int count, char* arguements[], char** path, int* no, char** ip);

//Retrieves the request data from request file
//Breaks apart the given requestString and inserts the information into newRequest structure
void retrieveRequestDataFromFile(char* requestStringG, struct request* newRequest);

//Adds a new request to a dynamic request array.
//Returns the pointer on success, NULL on failure
//Increments the size of the string given in 3rd parameter
//Assigns the new request to the second parameter as well
struct request* addRequest(struct request currentRequest, struct request** allRequests, int *requestCount);
```

```c
//Formats the start output of the client
//The output is put inside the first parameter
//requestC is client's total request count
void arrangeStartOutput(char* output, int requestC);

//Thread function of client. Deals with it's assigned request
//args include a requiredData structure
void* threadFunction(void* args);

//Formats the thread start output of client
//The output is put inside the first parameter
//threadNo is the number of the printing thread
void arrangeThreadStartOutput(char* output, int threadNo);

//Formats the request output of client
//The output is put inside the first parameter
//threadNo is the number of the printing thread
//threadRequest is a structure containing the request information of the client
void arrangeRequestOutput(char* output, int threadNo, struct request threadRequest);

//Formats the response output of client
//The output is put inside the first parameter
//threadRequest is a structure containing the request information of the client
//result is the result for the request
void arrangeResponseOutput(char* output, int threadNo, struct request threadRequest, int result);

//Formats the thread end output of client
//The output is put inside the first parameter
//threadNo is the number of the printing thread
void arrangeThreadEndOutput(char* output, int threadNo);

//Formats the end output of client
//The output is put inside the first parameter
void arrangeEndOutput(char* output);

//Checks if a string given as the first parameter is only space
int isAllSpace(char* line);
```

```c
//Retrieves the required information from the arguments.
//Returns 0 on success, -1 on failure
//The retrieved info are assigned to path, start, end, ip, portNo
int retrieveArguments(int count, char* arguements[], char** path, int* start, int* end, char** ip, int* portNo);

//Retrieves the servant's assigned citys' numebrs from the given numbersString
//Assigns them to start and end
void retrieveCityNumbers(char* numbersString, int* start, int* end);

//Compares two city names.
//Returns a value >0 if a>b
//Returns 0 if a=b
//Returns a value <0 is a<b
int compareCityNames(const void * a, const void * b);

//Arranges the given currentservant in a one line string format.
//Returns the pointer on success, NULL on failure
//Also assigns the new input to the first parameter
char* arrangeSocketInput(char** socketInput, struct servantInfo currentservant);

//Formats the data loaded output of servant
//The output is put inside the first parameter
//pid is the pid of the servant
//The servant will be responsible of cities between firstCityName and secondCityName
void arrangeDataLoadedOutput(char* output, int servantPid, char* firstCityName, char* secondCityName);

//Formats the servant start output of servant
//The output is put inside the first parameter
//port is the port number of the server
//pid is the pid of the servant
void arrangeServantStartOutput(char* output, int pid, int port);

//Thread function of servant. Deals with the given connection.
//args contain a file descriptor for server socket
void* threadFunction(void* args);
```

```c
//Adds a new transaction to a dynamic transaction array.
//Returns the pointer on success, NULL on failure
//Increments the size of the string given in 3rd parameter
//Assigns the new transaction to the second parameter as well
struct transactions* addTransaction(struct transactions currentTransaction, struct transactions** allTransactions, int *transactionCount);

//Checks if the given dateE lies in between startT and endD
//Returns 1 if true, 0 if false
int compareDates(char* endD, char* startT, char* dateE);

//Calculates if the date with given d1, m1, y1 is bigger than given d2, m2, y2
//If true, returns 1. If false, return -1
int calculateDateDifference(int d1, int m1, int y1, int d2, int m2, int y2);

//When a SIGINT is received, this handler is run.
//It assigns the sigintFlag as 1
void sigintHandler(int signum);

//Formats the termination output of servant
//The output is put inside the first parameter
//pid is the servant's pid
void arrangeTerminationOutput(char* output, int pid);

//Used during atexit
//Frees some globally used pointers
void exitFunction(void);
```

```c
//requestQueue structure is used to represent the request queue in the server
//In this queue retrieved file descriptors from connections with clients are held
//totalRequests is used to check if the queue is full, if the queue needs to go back to the start (circular)
//currentIndex is used by threads to find out the next waiting file descriptor
//active requests is used to check if there is any waiting requests
struct requestQueue{
    int clientSocketFdArray[1024];
    int totalRequests;
    int currentIndex;
    int activeRequests;
};

//Retrieves the required information from the arguments.
//Returns 0 on success, -1 on failure
//The retrieved info are assigned to portNo, tCount
int retrieveArguments(int count, char* arguements[], int* portNo, int* tCount);

//Retrieves servant's data from the given dataString
//Puts the retrieved information into given newServant structure
void retrieveServantData(char* dataString, struct servantInfo* newServant);

//Adds a new servantInfo to a dynamic servantInfo array.
//Returns the pointer on success, NULL on failure
//Increments the size of the string given in 3rd parameter
//Assigns the new servantInfo to the second parameter as well
struct servantInfo* addServant(struct servantInfo currentServant, struct servantInfo** allServants, int *servantCount);

//Thread function of server. Waits for a request, retrieves when available and deals with it.
//The args include the thread number of the thread
void* threadFunction(void* args);

//Formats the request arrived output of server
//The output is put inside the first parameter
//threadNo is the number of the printing thread
//no is the thread no
void arrangeRequestArrivedOutput(char* output, struct request arrivedRequest, int no);
```

```c
//Formats the servant connection output of server
//The output is put inside the first parameter
//threadNo is the number of the printing thread
//no is the thread no
//servantP is the connecting servant PID
void arrangeServantConnectionOutput(char* output, int servantP, int no);

//Formats the result retrieved output of server
//The output is put inside the first parameter
//threadNo is the number of the printing thread
//no is the thread no
//result is the retrieved result for a request
void arrangeResultRetrieved(char* output, int result, int no);

//Formats the servant loaded output of server
//The output is put inside the first parameter
//threadNo is the number of the printing thread
//loadedServant is a structure containing information related to the recently loaded servant
void arrangeServantLoadedOutput(char* output, struct servantInfo loadedServant);
```