Distributed Systems - Exercise 1 (group 89)

Ramiro Pinto Prieto - 100346042 Maryna Kotok - 100379674

1. Description of the code

1.1. server.c

File *server.c* contains the server code responsible for managing the data structures that store the information. In our case, the data structure chosen was a linked list, which is an easy way to implement a dynamic storage system that does not set a limit on the number of elements that can be stored. For its implementation we have based on the code provided by the website geeksforgeeks.com. From which, we made the necessary changes to adapt it to the functions and data that we had to manage. The elements of the list are structs called *Node*, which contains one struct request *(See section keys.h)* and the pointer of the next element of the chain.

```
struct Node {
    struct request myreq;
    struct Node *next;
};
```

This server satisfy the clients request by means of a busy waiting. Each request is managed in a concurrent fashion. In order to do this, first, the server's posix message queue is created with the proper attributes and permissions. Then, after defining the threads, mutex and conditional variables attrbutes, the infinite loop in charge of carrying out the busy waiting is started. When the server receives one client's request, it creates a thread to manage it atomicly by invoking function *process_message* within each thread.

Function *process_message* enters the critical section for preparing the reply that it is going to send back to the client. First, it initializes the reply struct and makes a local copy of the received message, both reply and msg_local are *struct request (See section keys.h)*. Then, it executes the function required by analyzing the *fcode* of the client request within a switch statement. All switch cases follow this execution path:

1.Function execution. 2.Store result within the reply.[1] 3.Open client posix message queue. 4.Send reply to client. 5.Close client queue.

Each function has a numeric code which allows to identify them:

- **Case -1** and **default**: Error code. It sends the default reply with the *fcode* -1.
- **Case 0** - ***init***: It executes "`int s_init(struct Node** head_ref)`" function. This function initializes the linked list by deleting all nodes storing the triplets values. It returns 0 on success and -1 on error.
- **Case 1** - ***set_value***: It executes "`int s_set_value(struct Node **head_ref, struct request *req)`" function. This function inserts the *struct request* sent by the client within the linked list. It returns 0 if it was inserted successfully and -1 in case of standard error. Trying to insert an element with an already existing key is considered an error (-1 is returned).
- **Case 2** - ***get_value***: It executes "`struct request* s_get_value(struct Node* node, int key)`" function. This function obtains the item of the list associated with the provided key. It returns a pointer to the *struct request* of that list element. The status of the execution (0 on success, -1 on standard error or 1 on *not existing key* error) is stored on the *fcode* attribute of the returned *struct request*.
- **Case 3** - ***modify_value***: It executes "`int s_modify_value(struct Node* node, struct request *myrequest)`" function. This function modify the values of the list element associated with the key of the given *struct request*. It changes the previous values by the ones provided on the clients request, which is passed as an argument of the function. It returns 0 on success, -1 on standard error and 1 on *not existing key error*.
- **Case 4** - ***delete_key***: It executes "`s_delete_key(struct Node **head_ref, int key)`" function. This function deletes the list element associated with the given key. It returns 0 on success, -1 on standard error and 1 on *not existing key error*.
- **Case 5** - ***num_items***: It executes "`s_num_items(struct Node *node)`" function. This function returns the number of items within the linked list. It returns -1 in case of error.

[1] The *fcode* of the reply instead of determining the function to execute, as in the client's message, it indicates whether there has been any error in the function execution (-1 [Standard error] or 1 [Repeated or not existing key]) or not (0)

In addition, the ***print_list*** function was implemented for debugging purposes. This function prints on the standard output the key and values of all the elements stored within the linked list.

The first element of our linked list is declared as *head*. This element is passed as argument to all the functions that deal with the linked list. It may be passed by reference (*Node** head_ref*) or by value (*Node* node*).

1.2. keys.c

File *keys.c* contains the implementation of the API calls used by the clients. All functions follow the same structure:

1. Open the client queue.
2. Connect to server.
3. Prepare reply by encapsulating the arguments and adding the corresponding *fcode*.
4. Send request to to server.
5. Receive reply from server.
6. Return de service execution result stored in *reply.fcode*.
7. Close connection with server.

Main functions[2] are:

- **int init()**
- **int set_value(int key, char *value1, float value2)**
- **int get_value(int key, char **value1, float *value2)**
- **int modify_value(int key, char *value1, float *value2)**
- **int delete_key(int key)**
- **int num_items()**

This functions invoke the servers services described in point *1.1.*. Each one corresponds to the homonym server function but starting with a *s_*.

For efficiency reasons other support functions were implemented. The purpose of this functions is to gain code readability.

- **open_client()**: It creates (or opens, if it is already created) the clientes message queue by means of *mq_open* call.
- **connect_server()**: It opens with write only permissions the servers message queue by means of *mq_open* call.
- **close_conection()**: It closes the server and client queue by using *mq_close* and destroy the clients queue by means of *mq_unlink*.
- **send(struct request * req)**: It sends *struct request* argument to server by using *mq_send* call.
- **int receive(struct request * reply)**: It receive the servers reply by using *mq_receive*.

1.3. keys.h

File *keys.h* is the header file which is included in the *server.c* and *client.c* files. This header provides the declaration of the functions implemented in *keys.c* and the *struct request* used as message in the posix queues communication process. *Struct request* contains 5 attributes:

- ***key***: it is an integer value that acts as message identifier.
- ***value1***: It is a string value with a maximum length of 256 bytes.
- ***value2***: It is a float number.

[2] Functions description included in the exercise statement

- ***q_name***: It is a string with a maximum length of 256 bytes that stores the name of the queue that is sending the message, in order to allow the receiver to reply to the sender.
- ***fcode***: It is an integer value. If the *struct request* is sent from client to server, it indicates the service to execute (-1-*error*, 0-*init*, 1-*set_value*, 2-*get_value*, 3-*modify_value*, 4-*delete_key*, 5-*num_items*). Otherwise, if the *struct request* is sent from server to client, it indicates the execution status (0-*success*, -1-*standard error*, 1-*not existing or repeated key error*).

```
struct request{
  int key;
  char value1[256];
  float value2;
  char q_name[256];
  int fcode;
};
```

1.4. client.c

File *client.c* invokes the functions implemented in *keys.c*. It has been used for tenting purposes. Further explanation on section *2* (*Testing plan*).

1.5 Makefile

File *Makefile* is an script which allows to compile the client, server and keys files. In addition, it automatically creates and links *libkeys.a* static library.

Finally, it includes an option for deleting all the executable files previously created (*make clean*).

2. Test plan

The test plan has been designed with the purpose of checking all the requirements described in the exercise statement. In order to achieve this, we have implemented a test plan in the *client.c* file that invoke the *libkeys.a* library functions in different contexts. Each function is invoking at least one time for testing: its normal execution, the empty list case and the no existing/repeated key.

Order	*Function*	*Expected output*	*Context*
1	init	Init success	Empty list
2	set_value	Set value success	Empty list
3	set_value	Set value error	Repeated key
4	set_value	Set value success	Not empty list
5	init	Init success	Not empty list
6	get_value	Get value error	Empty list
7	modify_value	Modify value error	Empty list
8	delete_key	Delete key error	Empty list
9	num_items	Count items success	Empty list
10	set_value	Set value success	Empty list (Fill list 1)
11	set_value	Set value success	Not empty list (Fill list 2)
12	get_value	Get value success	Not empty list
13	get_value	Get value error	Not existing key
14	modify_value	Modify value success	Not empty list
15	modify_value	Modify value error	Not existing key
16	get_value	Get value success	Not empty list and value modified
17	num_items	Count items success	Not empty list
18	delete_key	Delete key success	Not empty list
19	delete_key	Delete key error	Not existing key
20	num_items	Count items success	Not empty list and deleted value
21	client.c	Client success	Single client
22	client.c	Client success	Single client after previous client
23	client.c	Client success	Two concurrent clients

All test cases were successfully passed. (See *test_server.txt* and *test_client.txt*).