uc3m | Universidad Carlos III de Madrid

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

PRACTICAL EXERCISE 3 - RPC

Students:
Moisés Hidalgo Gonzalez - 100405918
Álvaro Morata Hontanaya - 100405846

SYLLABUS

SYLLABUS **2**

DESIGN **3**

DATA STRUCTURE 3

SERVER 3

KEYS 3

CLIENT 4

COMPILATION **4**

COMPILATION USING CMAKE 4

TEST PLAN **5**

PROBLEMS FOUND & CONCLUSIONS **5**

DESIGN

In this part of the document we will give a brief description of the different design choices we have made to develop this exercise.

DATA STRUCTURE

The data structure we have chosen is based on an external storage implementation. There will be a folder named "`/messages`" in the parent directory that will be created with the function `init()`. Inside this folder there will be files with the naming "`{key}.txt`". These files will be the messages of the database. Their contents will be a string with the format "`key-value1-value2-value3`", where:

- `key` is an integer that will identify the message.
- `value1` is a string of a maximum of 255 characters.
- `value2` is an integer.
- `value3` is a floating point number.

There will be only one message per file and vice versa.

SERVER

In the file "keys_server.c" we have programmed everything related to our data structure, as well as the main loop of the server. It also contains the server-side implementation of the functions stated in the statement of the exercise, but in this one, they have been modified to be able to communicate with RPCs. This server-side implementation means that these functions are called by the client to retrieve data or store it in the database. The function they execute is communicating directly with the files that store the messages and returning the data through RPCs.

The mutual exclusion has been implemented in each one of the methods that access the critical section (the database of messages), so there's no race condition when the database is accessed by a client.

The file was originally generated with the *rpcgen* command and then modified by us with our implementation of the data structure.

KEYS

In the file libkeys.c there is implemented the connection between the files generated by using rpcgen and the client that calls the remote functions.The connection is established in each call to a remote function. Therefore, in our design, all functions create a new CLIENT.

The checkenv() function will capture the environment variable IP_TUPLES which will contain the server's IP.

All the functions follow the same structure:

1. Initialization of a new CLIENT
2. initialization of retval
3. Checking that parameters are valid
4. CLIENT declaration
5. Calling the keys_clnt.c function that will call the function on the server side

6. Checking that the value of retval is correct
7. Destruction of the CLIENT
8. Loading data into the parameters if necessary
9. Return

CLIENT

It's the same client as the previous exercise, since the communication has changed but it's supposed to work with the same code as before (which it does).

The file "`client.c`" includes the "`libkeys.h`" header, accessing all of the methods programmed and required by the exercise.

This file contains different methods:
First, we have the "`sendData()`" method that will call the "`set_value()`" function with some predefined values and will send them to the server so it will store it in the backend as a file.
Next, we have the "`reset()`" method that will call the "`init()`" method, destroy all existent files in the "database" with it, and insert a new one with the "sendData()" method.

Then, we made unit tests for each of the methods in "`keys.c`" to later call them in a "`test_function()`" that will be run in the threads. This "`test_function()`" calls the different unit tests and "`reset()`" the data before each.

COMPILATION

The compilation of the project has been realized using CMake. In this section we will explain how to compile the project using this tool.

COMPILATION USING CMAKE

The file "`CMakeLists.txt`" has been written using the different instructions that allowed us to compile the source code and the shared library in the same way we were able to do with GCC.
To compile the different files we need to enter the following commands into a linux terminal being in the same folder as the file "`CMakeLists.txt`" is and the C files:

```
> cmake .  #It will create a makefile to compile the project
> make     #Will compile the files using gcc
```

With these two commands the project will be compiled, delivering us the executables "`server`", "`client`" and the dynamic library "`libkeys.so`".
To run the programs, we need to run the following commands:

```
#Terminal 1:
> ./server
#Terminal 2:
> IP_TUPLES=127.0.0.1 ./client
```

We need to specify the host address (server) when we are calling the client, so we need to set the environment variable `IP_TUPLES` with `export` or call the programs as it is shown here.

TEST PLAN

We have test methods that do the same functionality for our example thread 1 and thread 2. The only difference is the data they handle (different keys, values, etc).

The test plan aims to test in different scenarios the functions required for this exercise:

- `int init()`
- `int set_value(int, char *value1, int value2, float value3)`
- `int get_value(int, char *value1, int *value2, float *value3)`
- `int modify_value(int, char *value1, int value2, float value3)`
- `int delete_key(int)`
- `int exist(int)`
- `int num_items()`

Following the aforementioned objective, our main test function is conformed of the following main methods:

- `init_Test()`: Tests the init method and checks that it returns proper value.
- `set_valueTest()`: Sets a value and checks that it returns the proper value.
- `get_valueTest()`: Prepares some test data in the server side and in the method and uses get_value method to check it matches.
- `existTest()`: Checks that given a local key it returns a value of that key (previously inserted on the DB).
- `modify_valueTest()`: Checks that it returns a successful value after calling the function modify_value with some test data.
- `delete_keyTest()`: Checks that it returns a successful value after calling the function modify_value with some test data (key has been deleted).

Before, it contained reset() and resetMultiple() methods which were giving a lot of troubles because of the inclusion of the init test in the middle of the threads execution, which caused problems since it deleted values and concurrency problems. Therefore, we moved the init_Test() to the main and did some fixes on the test data sent (now sent on the main too) and on the order of executions in the threads' test functions. Everything works correctly now.

In addition, we have manually checked that the changes we wanted to do (deleting keys, modifying values, etc) actually worked, represented on the files in the directory "messages".

PROBLEMS FOUND & CONCLUSIONS

We have been able to complete the exercise. During the making of the exercise we did find several errors we had in the code of the data structure, but we were able to fix them. One of them was modify_value(), since it wrote into the file the modified message but also garbage that wasn't readable by the text editor (binary code). We solved this by implementing it with a delete_key and set_value and it works correctly now. We also found errors in other parts of the code but fixed them, and right now, the program runs concurrently without issue. All the tests work as expected after the redesign of the test plan as well.