uc3m | Universidad Carlos III de Madrid

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

PRACTICAL EXERCISE 2

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

SYLLABUS

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 "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. 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 sockets.

This file also contains a method called "`deal_request()`" that processes the request sent by the client through the socket, implemented with mutual exclusion in mind so that the server is concurrent and there is no race condition when accessing a file or variable.

For the last part of this file, we have the main loop of the server, implemented using multithreading and sockets to receive the requests from the client.

KEYS

Here we implemented a "`keys.h`" header which will store all the function headers requested in the practice assignment and will be accessed through the "`keys.c`" file.

For this we deal with two types of struct: request and response. Then, we make use of a "`send_request()`" method that will take as an input a "`request`" type of data (struct holding the data for the request) and return a "`response`" type of data (struct holding all the fields relative to the response). This method of "`send_request()`" will do all the socket handling on the client side and will send the messages that will be later received by the server with a specific operation code for each method and thus will process it accordingly. The individual methods are setting / receiving the data that we want to interact with the server side which will perform all operations atomically.

CLIENT

The file "`client.c`" includes the "`keys.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 50070
#Terminal 2:
> IP_TUPLES=127.0.0.1 PORT_TUPLES=50070 ./client
```

As it can be seen, a port number needs to be specified when calling both the server and client. This port number can be chosen as long as that value is not reserved for a specific service. For that reason, we have chosen one of the private ports, 50070.

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).

Also, it contains some auxiliar methods that will be used in the main test function

- `reset()`: Resets our DB (deletes the files stored in ./messages) and sends the first tuple.
- `resetMultiple()`: Does the same as reset, but inserts 5 files into the DB.

In addition, we have manually checked that the changes we wanted to do (deleting keys, modifying values, etc) actually worked.

PROBLEMS FOUND & CONCLUSIONS

With this second exercise we have found that some of the problems we had in the previous exercise were related to the message queues, and by the time we implemented the code with sockets, those problems were solved.

We have been able to test almost all of our code this time and thus we have narrowed the problems to one. It seems to be mainly with threads, as sometimes when we execute the code it runs flawlessly and in some cases it gets stuck on the client side after the last method of the program, whichever that might be (we tried swapping them).

When the program runs without that last issue, the tests of `num_items()` fails. However, when it clogs, one of the tests for `num_items()` works but the other never returns whether it failed or was correct. If we don't test that one function, the program runs perfectly.

We tried debugging it but we were unsuccessful, and we hope we can clarify it in a tutorship.