Carlos Gutiérrez Paradela
100291121

Message Passing: POSIX Message Queues

Introduction

The purpose of this exercise is to get familiar with message passing using queues in POSIX, in order to communicate one or several running clients that send requests to a server, which deals with each request stored in the server queue by creating threads on demand that will respond to the corresponding client queue with the provided result of the consumed services. In this report there will be described the main functionalities of the different parts of the system and how their design look like.

Design

This section will describe the different acting parts of the system and how they are organized.

Protocol.h

As several programs need to be handling the same request and response structures, a header file was decided to be created to define the common structure of these messages structure. The *request* structure consists of 5 fields.

- *name:* A string with the *name* of the vector that will act as input to the system.
- *Ni:* An integer *Ni* where the value of N (in case of invoking the *init()* method) or *i* (in case of invoking the *set()* or *get()* methods) is passed as parameter.
- *value:* An integer with the *value* needed by the *set()* function.
- *op:* A character storing the type of service that will be invoked. 0 for *init()*, 1 for *set()*, 2 for *get()* and 3 for *destroy().*
- *queue:* A string that stores the name of the queue to which the server must respond to.

The *response* structure from the server has the following structure:

- *value:* A char type variable that stores the output of the four different services.
- *get_value:* An integer containing the obtained element of the *get()* operation.
- *error:* A char that stores the error state of the server.

This header file also contains the function headers of the methods provided by the *array.c* code.

Client

The client side consists of two main files, described in the following subsections.

Array.c

It is the actual magic happening behind the scenes for the creation of the queues on the client side. It provides the interface to which the client connects and invokes functions in order to create a queue for each type of request, and then wait to receive the answer on the same created queue. It also opens the file descriptor of the server queue to which the requests are sent to. However, this code only has creation and deletion privileges for the client queue. The destruction of the server queue is only handled at server side. Therefore, it is worth mentioning that the client queue is created, if already did not exist, with read only permissions (as will be reading the responses of the server from it), while the server queue descriptor is opened with write only permissions.

This code contains the functionalities of the four services to be consumed by the server. All the four methods in this file follow the same structure: build the *request* structure values using the input parameters of the function; open the client queue; open the server queue; send the *request* to the server queue; wait for the *response* at the client queue; treat the *response* and send back the result as output. The only different between the functions resides in the number of operation (0, 1, 2 or 3) inserted to the *op* field of the *request*. Before exiting the functions with an error or with a result, the client queue is closed and unlinked.

Client.c

This is the 'dumb' client that only invokes the abovementioned functions, who will be in charge of contacting with the server. It only prints on the console the result obtained from each function called.

Server.c

The server is the entity that deals with all the complexity. To begin with, it defines a *node* structure in charge of acting as linked list to store the different initialized vectors in memory. Once the server process ends, the list is erased. These nodes have the following fields:

- *name:* String that stores the name of the initialized vector.
- *v:* Array of integers containing the distributed vector values.
- *N:* Integer containing the size of the array, which will simplify calculations.
- *next:* A pointer to another *node* structure.

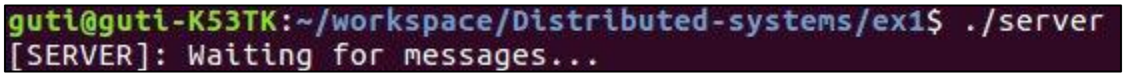
Firstly, the server defines a pointer to a *node* that will be the head of the list, the common starting point for every operation done with it. The *main()* function is in charge of initializing the queue, the mutex and attributes of the thread, and looping until a kill signal is detected by the *interruptHandler()* method. It was created in order to close and unlink the server queue with a Ctrl+C termination signal. The loop waits on the previously opened server queue until a *request* is received. When this happens, it creates a thread with a *detached* state, and the thread starts its execution in the *prepareThread()* function, bringing as parameter the received *request* structure.

The *prepareThread()* method first stores a private local copy of the *request* structure to avoid any race conditions between threads and mutex locks and unlocks. After that, it checks the value coming in the *op* field to redirect the flow into one service or another. Before invoking any function, it opens the client queue provided in the *queue* field of the *request* with write only conditions. Then, once the result is obtained from one of the four functions, it sends it back to the opened queue for the client to read it. At the end of this *prepareThread()* method, the thread exits and finishes its execution.

In order to avoid any conflict among threads when dealing with the list of *nodes*, where the distributed vectors are stored, the mutex is locked before iterating and operating over the list, and finally released when the operation is finished.

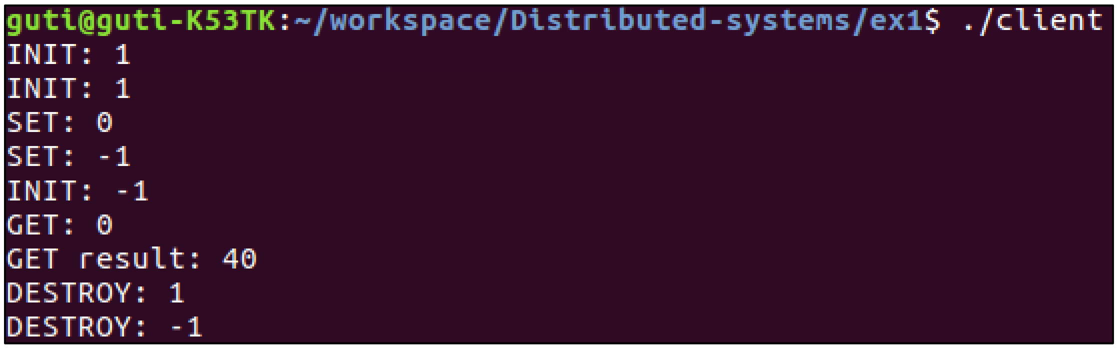
Compilation and execution

In order to compile and create the executables of the code, it will suffice by running the command `Make` inside the directory where the **exercise1.tgz** is decompressed. Once the executables are created, it is needed to put the server to run by typing '`./server`' into the command line. We will know if it is running because it will print in the console "`[SERVER]: Waiting for messages…`".



```
guti@guti-K53TK:~/workspace/Distributed-systems/ex1$ ./server
[SERVER]: Waiting for messages...
```

At this point, the serves is waiting for a client to invoke its services, so we will run the '`./client`' command in another console window. When done, the following result should be displayed at the client side:



```
guti@guti-K53TK:~/workspace/Distributed-systems/ex1$ ./client
INIT: 1
INIT: 1
SET: 0
SET: -1
INIT: -1
GET: 0
GET result: 40
DESTROY: 1
DESTROY: -1
```

It is worth mentioning that an additional example of the behavior of the get function is provided in this screenshot.