Message Passing Exercises

Exercise 1. Develop a server that allows you to get the day of the week in which a particular day falls. Design and develop the client and server in the following two assumptions

a) We have a system with the following services:
   - int **Connect**(int pid): this service establishes a connection with a process with pid identifier. Returns a connection identifier.
   - int **Accept** (): this service accepts a connection from a process running the Connect service. Returns a connection identifier.
   - **Send**(int ic, char *message, int long): this service sends a message of a certain length through the connection identifier ic.
   - **Receive(**int ic, char *message, int long): this service receives a message of a certain length of the connection with identifier ic.

   Assume that the process identifiers are integers and that the server is identified by the number 1000.

b) Consider a system that uses UNIX message queues.

Exercise 2. We want to design a distributed vector system. A distributed vector defines the following services:

- int **init** (char *name, int N). This service allows to initialize a distributed array of N integers. The function returns 1 when the array was first created. In case the array is already created, the function returns 0. The function returns -1 in case of error.
- int **set**(char *name, int i, int value). This service inserts the value at position `i` of the array `name`.
- int **get**(char *name, int i, int *value). This service allows you to retrieve the value of the element `i` in the array `name`.

Consider a system that uses UNIX message queues. Design a system that implements this distributed vector service.

Exercise 3. We want to build a distributed program consisting of N processes. Each of these processes will have associated a process identifier, an integer between 0 and N-1. Each process is identified by the PID variable (input parameter). Likewise, it can know the total number of processes (N) by consulting the NP variable. It is ensured that there are no two processes with the same identifier. We want these processes to have access to the following services:

- int **init**(void); This service must be executed by all processes at the beginning of execution. This operation is essential to use the rest of services.
- int **send**(int n, char *buf, int len); This service sends a message (`buf`) of length `len` to the process with identifier n (between 0 and N-1). This service is non-blocking.
- int **recv**(int n, char *buf, int len); This service receives from process `n` a `buf` message of length `len`. Reception is blocking.
- int **broadcast**(int root, char *buf, int len); With this service, the process with `root` identifier sends the message length `len` to all processes of the distributed program except the one that executes the call. All processes must execute the call.
- int **barrier**(void); This call blocks a process until all the processes of the distributed program have executed it, all processes must be executed so that they can continue their execution.

We want to implement these functions on a system that offers port-based message passing services. In this system, each process is associated with an address that is unique throughout the system and is specified by an integer. This address is associated with the process at the time of its execution. The services available in this system are as follows:

- int **get_address** (void); This service allows a process to obtain its address which ensures that it is unique throughout the system and that it receives at the time of commencement of its execution.
- int **publish_address** (int address, int n); This service allows the publication in a name service of a value associated with an address.
- int **seek_address** (int n); This service returns the address associated with the naming service with the value `n`.

- int **connect** (int address); This service establishes a connection with the process that executes in the passed address as argument. The call returns a communication descriptor that can be used in sending and receiving operations.
- int **accept** (int address); This service blocks the process that executes it until the process that executes in the passed address as argument makes a connect. The call returns a communication descriptor that can be used in sending and receiving operations.
- int **send_mess** (int fd, char * p, int len); This operation sends a message through the channel with `fd` descriptor. Shipping is non-blocking.
- int **recv_mess** (int fd, char * p, int len); This operation receives a message through the channel with `fd` descriptor. The reception is blocking.