uc3m | Universidad Carlos III de Madrid

BACHELOR’S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

PRACTICAL EXERCISE 4 - DESIGN EXERCISE

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

Table of Contents

1. Introduction

The aim of this exercise is to design a system for the management of the delivery notes for a company. For such a goal, this exercise will be solved with the same steps seen in class for solving this kind of problem. Those steps are the next parts that follow this introduction.

2. Client and Server Roles

Firstly, we are going to show a diagram of how the architecture of the communication is designed.

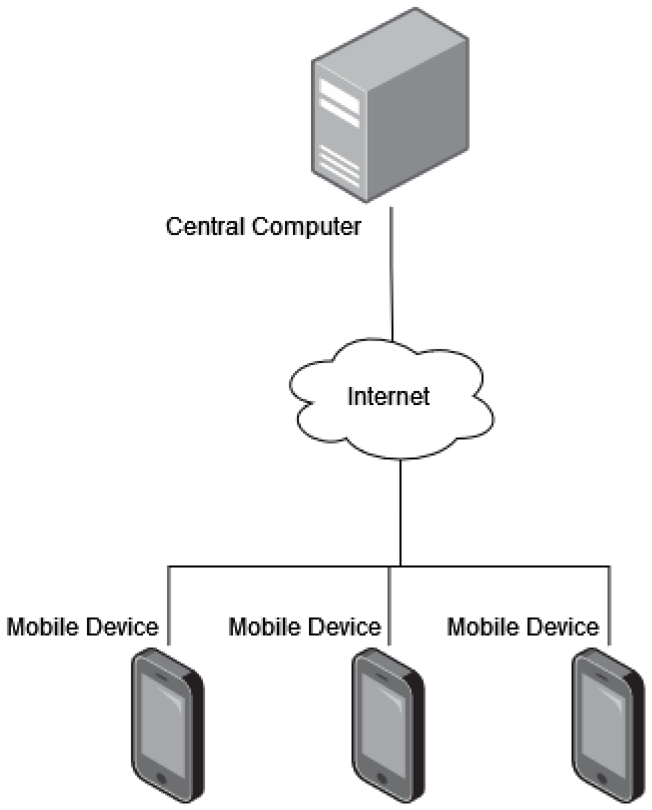

Figure 1. Communications diagram

In this case, the role distribution is the following one:

- **Server:** both the central computer and the mobile devices will act as a server.
    - The central computer will act as the server the majority of the time, as it will be the passive agent waiting for the requests from the mobile devices.
    - The mobile devices will only act as a server when they need to receive data from the request the server did (sending a chat message).
- **Client:** as in the case of the servers, the central computer and the mobile devices will act as clients.
    - The central computer will only act as a client when it sends a chat message to the mobile devices of the carrier.
    - The mobile devices will be the main clients of this communication, as they will be the active agents by requesting data to the central computer.

3. Network Protocol

For this scenario, we want the client-server connection to be connection-oriented, meaning that the protocol used would be TCP. The reason behind this is that TCP is more reliable than UDP, although it is slower. This way we ensure we do not have any data loss or interleave client requests.

On the other hand, we will sacrifice performance, but that won't be an issue in this scenario, as it would be in a streaming service. Reliability here is the priority, so TCP is the network protocol chosen for this scenario.

4. Naming

We will specify in this part the specific IP addresses and port numbers that will be used for the communication. As we do not know how many mobile devices we will have in the communication, we will assign an IP address range so those devices have those IP addresses assigned to them.

- Central computer:
    - IP Address: 10.1.0.1/16
    - Port: 10101
- Mobile devices:
    - IP range: 10.1.0.10/16 - 10.1.255.254/16
    - Port: 10102

  The IP range assigned to the mobile devices gives us a sufficient amount of IP addresses so there's no IP saturation in the network.

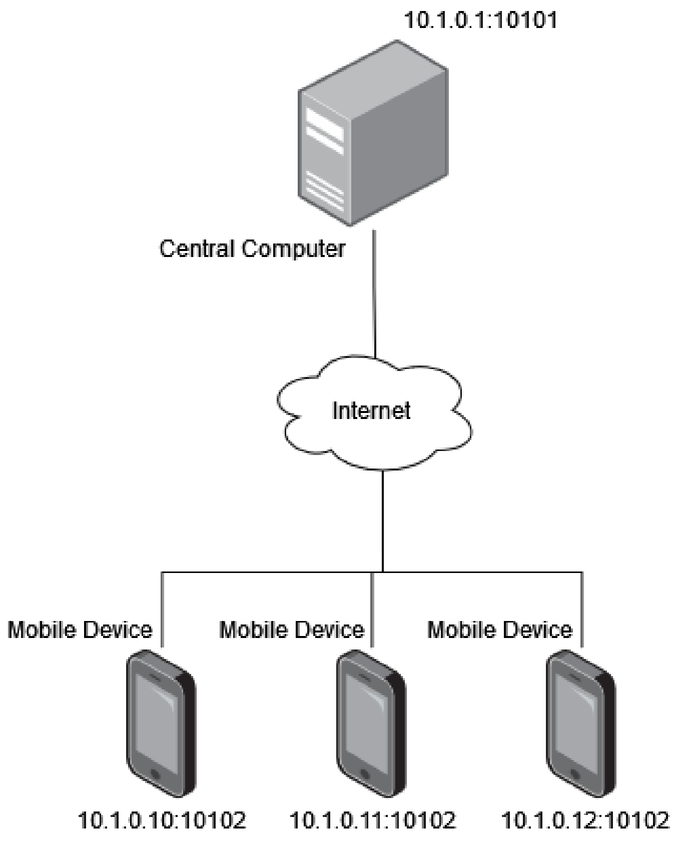

Figure 2. IPs diagram

5. Server Description

5.1. Single threading vs. multithreading

The central computer must be multithreaded, as there will be multiple connections between it and the mobile devices and probably in a large enough scale that will make the communication more efficient after having implemented multithreading.

On the other side, the mobile devices will be single-threaded, since they only need an active open thread to operate within the entire domain of its functionality.

5.2. Stateless / Stateful

A protocol is stateful or stateless depending on the length of interaction a client has with it and how much of the information is stored. In our case, since it's a TCP connection it will be stateful. This means that it will allow for the system to keep track of the connection information, which is necessary for things like the "chat" feature, or the mutual acknowledgement between the mobile devices and the central computer.

It's necessary because If our client delivers a request to the server in a stateful protocol, it will expect a response of some sort, which is the behavior specified in the exercise. If it does not receive a response, it will send the request again.

6. Protocol / Message Types

```
struct package {
      int package_id;
      int weight;
}
```

- Size in Bytes = 4+4 = 8 B

```
struct request{
      // opcode 1
      short opcode;
      int identifier;
      // opcode 2 (sends also the opcode)
      Int id_client;
      char signature [64];
      int products_delivered [64];
      // opcode 3 (sends identifier and opcode again)
      int products_undelivered[128];
}
```

- Size in Bytes = 2 + 4 + 4 + 64 + 64*4 + 128*4 = 508B

```
struct server_response {
        // Response of opcode 1
        char delivery_address[256];
        char customer_name[128];
        int customer_id;
        struct package packages [128];
        //response of opcode 2 and 3
        int err_code; // response for opcode 1, 2 and 3. 0 is OK
}
```

- Size in Bytes = 256 + 128 + 4 + 128*8 + 4 = 1024B = 1 KB

```
struct chat_message{
        int id_source;
        int id_destination;
        char message[1024];
}
```

- Size in Bytes = 4+4+1024 = 1032 B

EXPLANATION:

- `package`: struct that stores the package information.
    - `package_id`: identifier of the package.
    - `weight`: weight of the package.
- `chat_message`: structure that will contain the necessary information for a message sent by a courier or the server. The server will have id 0 and the clients will have an id ranging from 1 to the max number of carriers active in their workday.
    - `id_source`: id of the sender.
    - `id_destination`: id of the receiver of the message.
    - `message`: contents of the message.
- `request`: struct with the contents that can be requested in each of the transactions that are made in the system.
    - `opcode`: operation identifier.
        - `opcode = 1`: request asking for a delivery list.
        - `opcode = 2`: request sending the package delivered information.
        - `opcode = 3`: request delivering information about end of delivery journey.
    - `identifier`: ID of the carrier.
    - `id_client`: ID of the receiver of the package.
    - `signature`: signature of the client.
    - `products_delivered`: array sending a list of packages' IDs delivered to a client. The array will only have the first index (`products_delivered[0]`) with a value different from 0 if only one package has been delivered.
    - `products_undelivered`: sends a list of the undelivered packages' IDs to the central computer.

- `response`: struct with the data that will be sent as a response from the server to the client as a result of the previous request:
  - Data for the response of opcode 1:
    - `char delivery_address[512];`
    - `char customer_name[128];`
    - `int customer_id;`
    - `struct package packages [128];`
  - `int err_code`: this will work as the response for opcode 2 and 3 but also for opcode 1. In this case, if it's 0 it means that the communication and method went OK, but will take the value 1, 2 or 3 if there is any issue with one of the requests.

The way it works is we have a struct for everything and then the fields that are required for that request and needed to process it on the server side are managed by the methods with their own error handling.

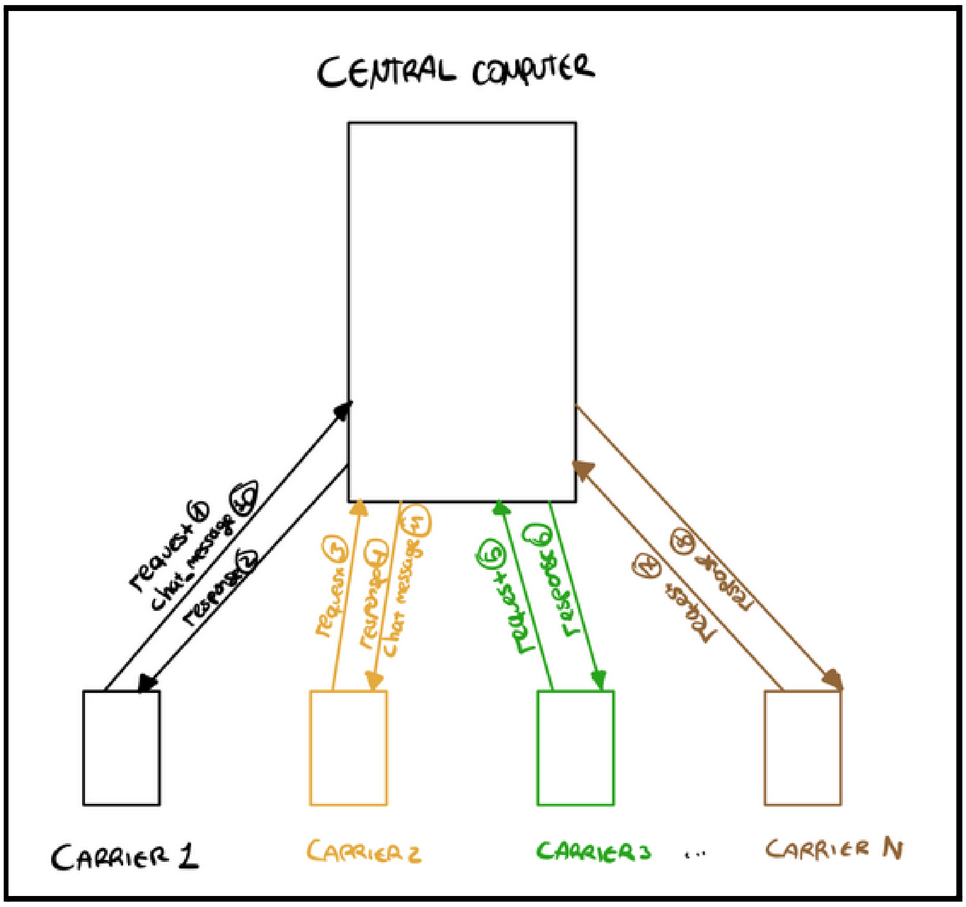

Figure 3. Diagram of the communication

7. Security

Security for this system is necessary if we want to avoid intrusions that could potentially imply data theft or problems with integrity of data, which might allow an attacker to modify products, swap them, change them, etc. There could also be data leaks from clients information if this is not taken care of.

Therefore, it’s crucial that good security is implemented. On the server-side, checking that the clients connecting are legitimate, protecting the database, and fixing vulnerabilities for the system not to be exploited. On the client-side, taking care that the communication is not being eavesdropped or that you are connecting to a “fake” server, sending the information to a third party.