RMI project

Activity 4 - Distributed Computing

Edgar Moreno Molina - 54483287J
Daniel Sanchez Solans - 47980551K
December 2019

Contents

List of Figures

1 Introduction

The goal for this activity is to develop a prototype called Mytube using the RMI technology. This system should allow users to remotely upload and download digital contents within an RMI system.

The following document explains all the decisions taken in order to achieve the goal menctioned above.

2 Functionalities

The system can perform the following functionalities:

(i) Clients can **upload** a digital content (for instance: text, music or video) to an RMI Server together with a title, description and an optional password.

(ii) Clients can **search** (locally and globally) the contents doing a partial (or exact) search on titles and descriptions.

(iii) Clients can **download** digital contents stored in the system.

(iv) Clients can **modify** the title of contents that they know the password of (if contents are password protected).

(v) Clients can **delete** contents that they know the password of (if contents are password protected).

(vi) Servers provide a **list** of all the contents locally and globally stored.

(vii) Servers generate a unique key for each content and store the content locally. The actual content is physically stored in a folder named with its unique key and the other information related to the content (title, description and password) is stored in a relational database.

3 Main decisions

3.1 Database and content storage

Each server will have its **own database** whose name can be defined when executing the server with *args*. If the database doesn't already exist, it will be created and so will a table named "content" in the database which will store information related with the contents: **key (Primary key), title, description and password**.

The **keys** for each content follow the **format "*x*-*y*"** where x is an integer identifying the number of content in the database and y is another integer that corresponds to the session server identifier. The session server identifier is provided by the central server when the server starts and lets the central server know its start. Each connection to the central server will have a unique server identifier, the central server will keep a file named *lastServerIdentifier.txt* which contains the identifier assigned to the last server that connected to the central server, hence **each content globally stored has a unique key**.

Having a **database** allows for an **easy implementation of partial searches** and overall **quick search**.

Regarding **content storage**, each content is stored in a **folder named** with the format ***x*-*y*** explained above. This folder is contained in a folder named */contents* at the root of the server project. For instance, content *test.txt* with key *1-4* would be stored in */contents/1-4/test.txt*.

3.2 Password protecting files

A client uploading a digital content will be able to protect a content with a password if wanted. The password will restrict the **deleting and renaming operations**.

At first we decided to restrict the content download with password as well, but as as the professor pointed out this was not specified in the instructions and we decided to not protect the download with password. That means all clients can download all files.

3.3 Central server

In case a client wants to download a digital content not stored in the server the client is connected to, a search on other servers is needed (also known as *global search*). One of the approaches is to have a central server which keeps track of all the servers up and running so it can forward download requests to the other servers that might own the content. The same approach can be used to search contents that are not locally stored on the server the client is connected to.

3.4 Global search

Our approach to global search is quite simple; in the case a client wants to search a content, either by its title or description, the central server will be asking the server with the most number of contents first, then the second server with most contents and so on. This way is likely that the servers that get asked first will actually own the content.

3.5 Exiting when can't proceed

When a server stops running the clients connected to it should stop running as well. That is because errors will occur if the server is not running when the client attempts to make a petition to the server. Our decision is to end the clients when the server disconnects and to stop the servers when the central server stops running. This procedure takes place automatically using the RMI facilities.

However when a client stops running, the server does not stop running, it does notified however that a client has exited. Just like when a server stops running, the central server doesn't but it does get notified that a server has exited.

4 Classes and UML class diagram

4.1 Classes

4.1.1 Server

This is the main class for the server part, it performs the following operations (sequentially ordered):

- Connects to the central server by looking up the central server object on the RMI registry started by the central server. This will allow for interactions with the central server such as global search and global download.
- It starts a RMI registry and binds the ServerImplementation object to the RMI registry.
- It defines a shutdown behavior which allows: notifying the clients connected to it and the central server that the server is about to exit, unbind the registry and unexport the ServerImplementation object.

4.1.2 ServerImplementation

This is the class that defines the stub for the client and the core part of the project. It defines and manages the contents *sqllite* database.

It contains all main RMI operations that a ServerImplementation object can perform triggered by the clients. Such operations are:

- Uploading a content: saves the content's related data to the database and saves the actual content in the server.
- Downloading a content: checks if the server has the content first, if not it calls the central server so it can perform a global search.
- Searching a content (either by title, description, partial title or partial description): calls the central server to perform a global search.

It also contains all RMI operations that a ServerImplementation object can perform triggered by the central server:

- Exit the server: if the central server exits, then the server notifies all the clients so they can exit, then disconnects from the database and exits.

4.1.3 CentralServer

This is the main class for the central server part, it performs the following operations (sequentially ordered):

- It starts a RMI registry named *"MyTubeCentralServer"* and binds the CentralServer-Implementation object to the RMI registry.
- It defines a shutdown behavior which allows: notifying the servers connected to it that the central server is about to exit, unbind the registry and unexport the CentralServerImplementation object.

4.1.4 CentralServerImplementation

This is the class that ServerImplementation objects will call when a global search or global download is required. It will call all the Servers currently up and running sequentially and once one of them reports a valid result it will return it to the Server who triggered the operation.

In order to implement an efficient download and search the central server will call the servers with more contents stored first.

4.1.5 Client

The main goal for this class is the first to deal with the user input and then trigger the necessary operations by using the stub and calling the ServerImplementation operations.

The main operations it performs are:

- Contacting the server by looking up the ServerImplementation object previously binded by the server (which is then casted to ServerInterface).
- Letting the server know it has connected so that the server can then report when exiting.
- Constantly asking the user for operations to perform which will call the stub remote methods on the server side.

4.1.6 ClientImplementation

This class serves the purpose of clients exiting when the server exits. The operation is triggered by the Server using the RMI facilities.

4.1.7 ContentDatabase

Contains all the operations needed to manage and perform the diverse operations for the contents related information on a *sqlite* database. It also can create the required database and table inside it if they're non existent.

4.1.8 DigitalContent

This class represents the information related to the contents which are: key, title, description and password. This class is needed to easily pack and unpack the information of the contents. For instance, when a server gets a list contents request it is convenient to return a list of Digital contents (*ArrayList<DigitalContent>*) and then when the client gets that list can easily call the *toString* method to easily format and output the adequate information related to those contents.

4.1.9 Output

This class serves the purpose of printing formatted and colored messages depending on the type of message: information, warning, error.

4.2 UML diagram

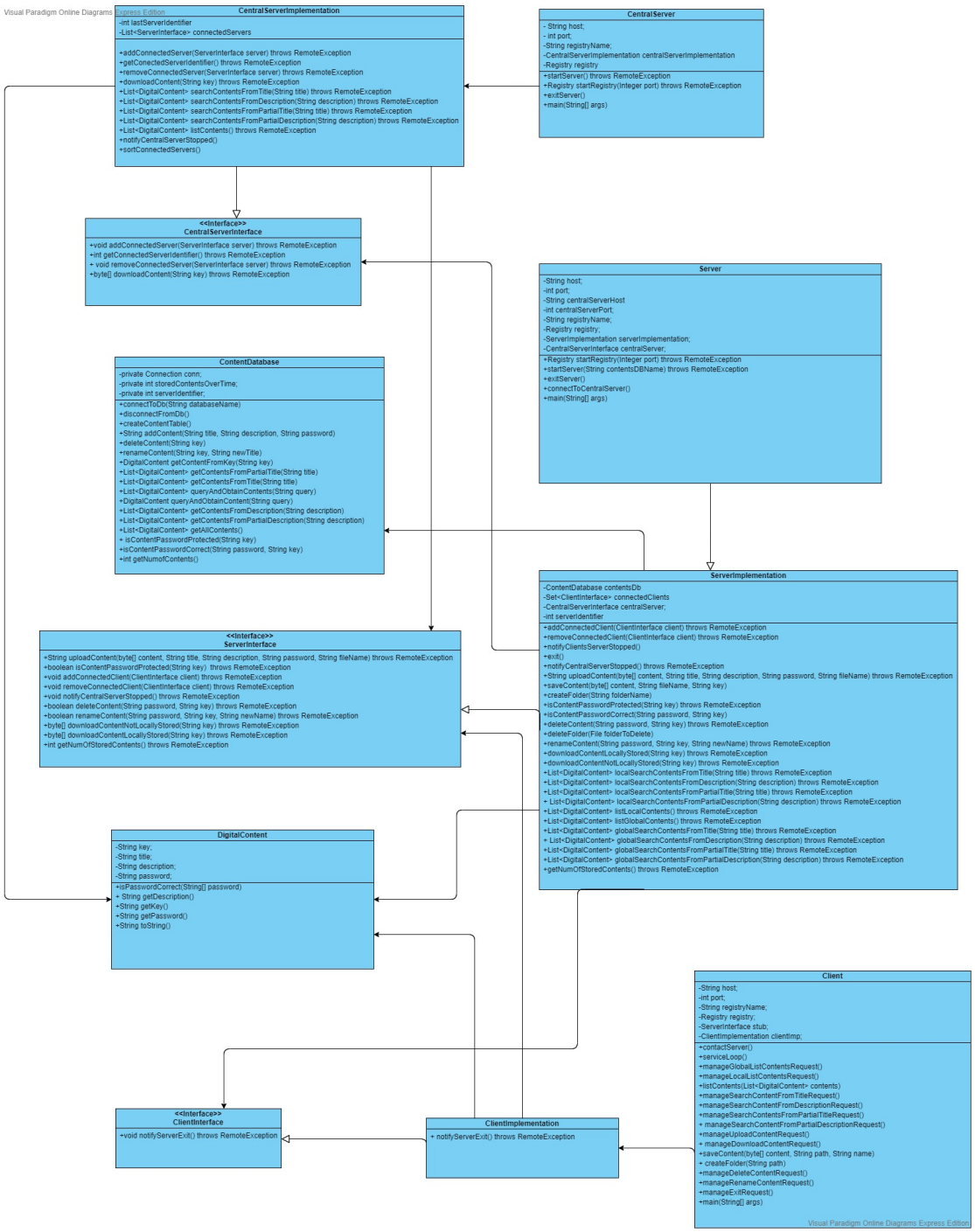

Figure 1: UML diagram

5 Sequence diagrams

The following section contains the sequence diagrams for the operations of global search, uploading and downloading a content.

5.1 Upload

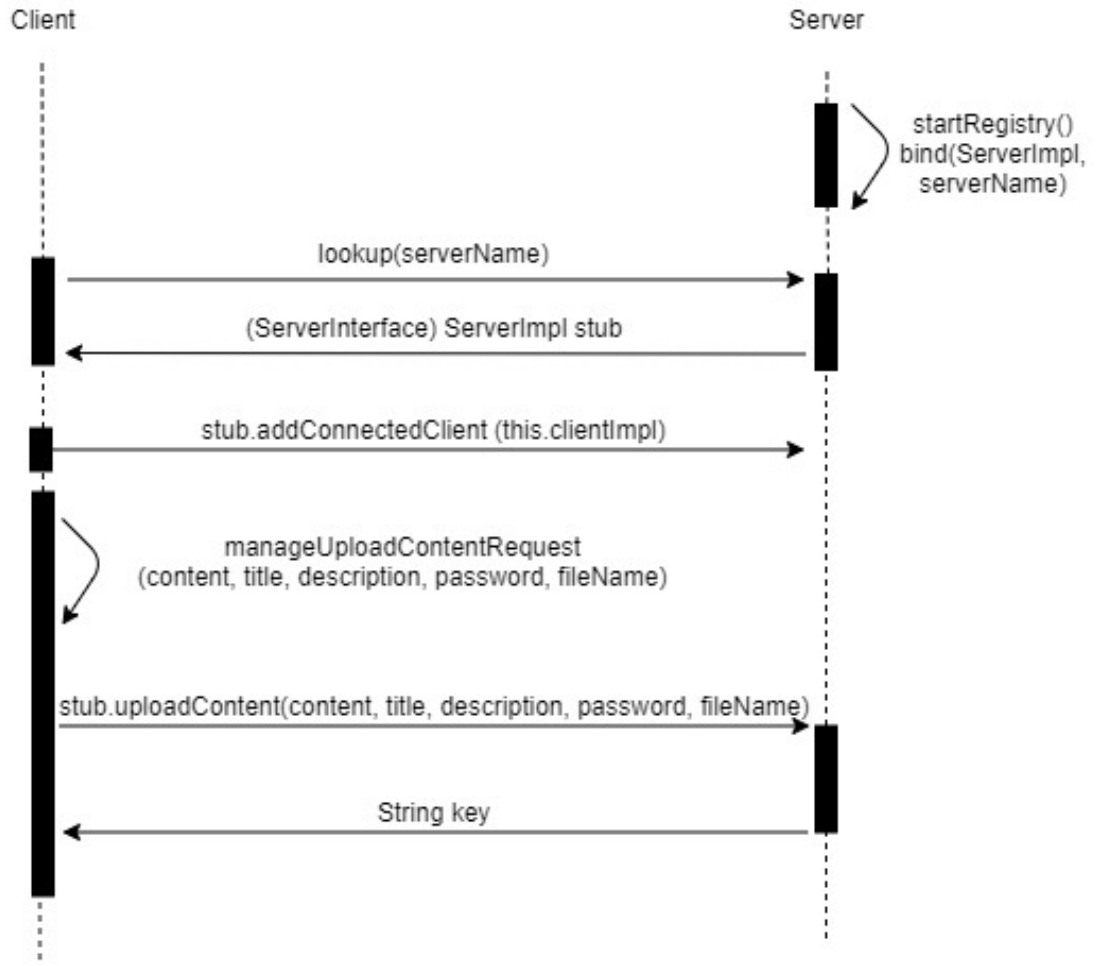

Figure 2: Upload operation sequence diagram

5.2 Download

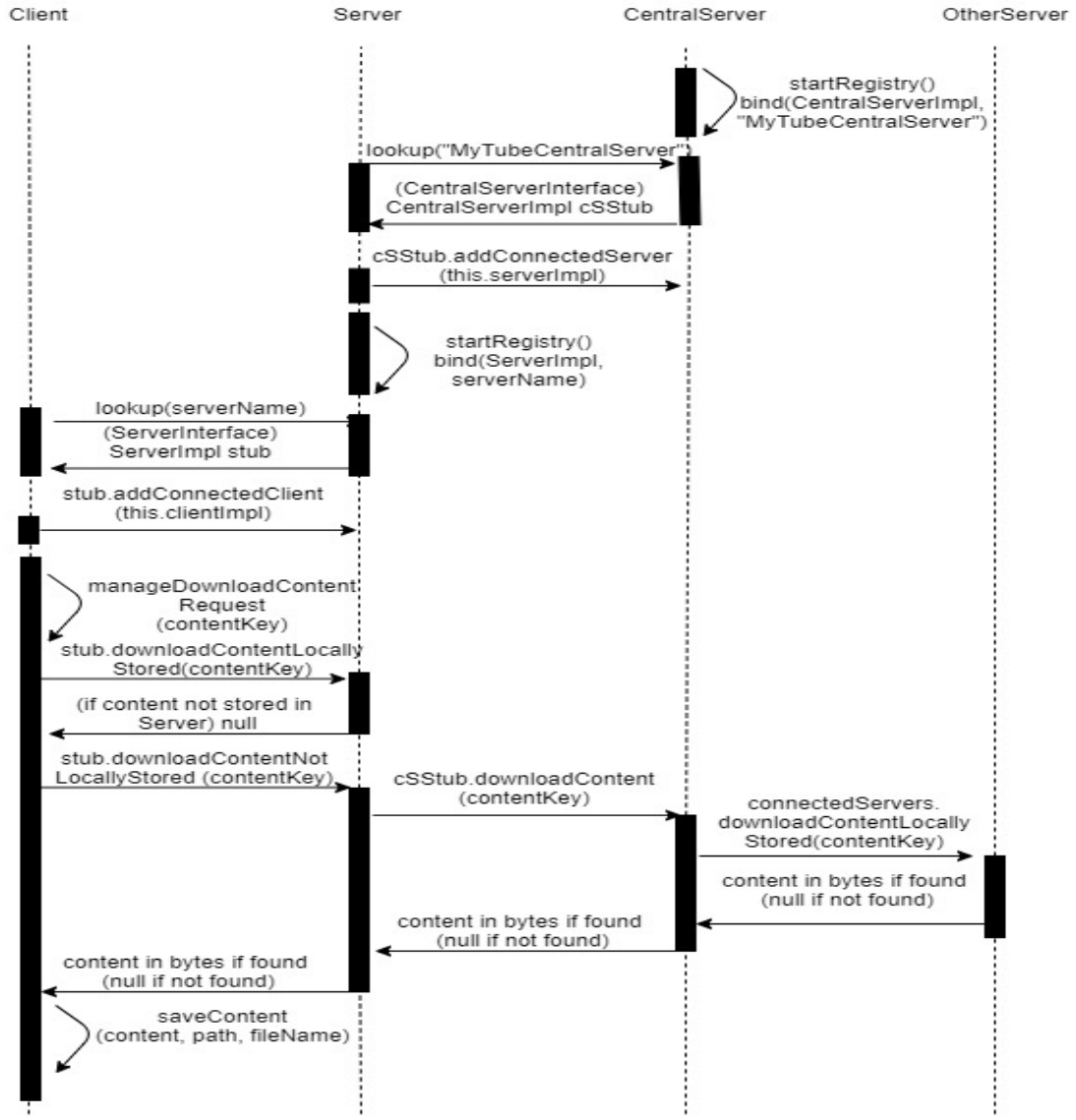

Figure 3: Download operation sequence diagram

5.3 Global Search

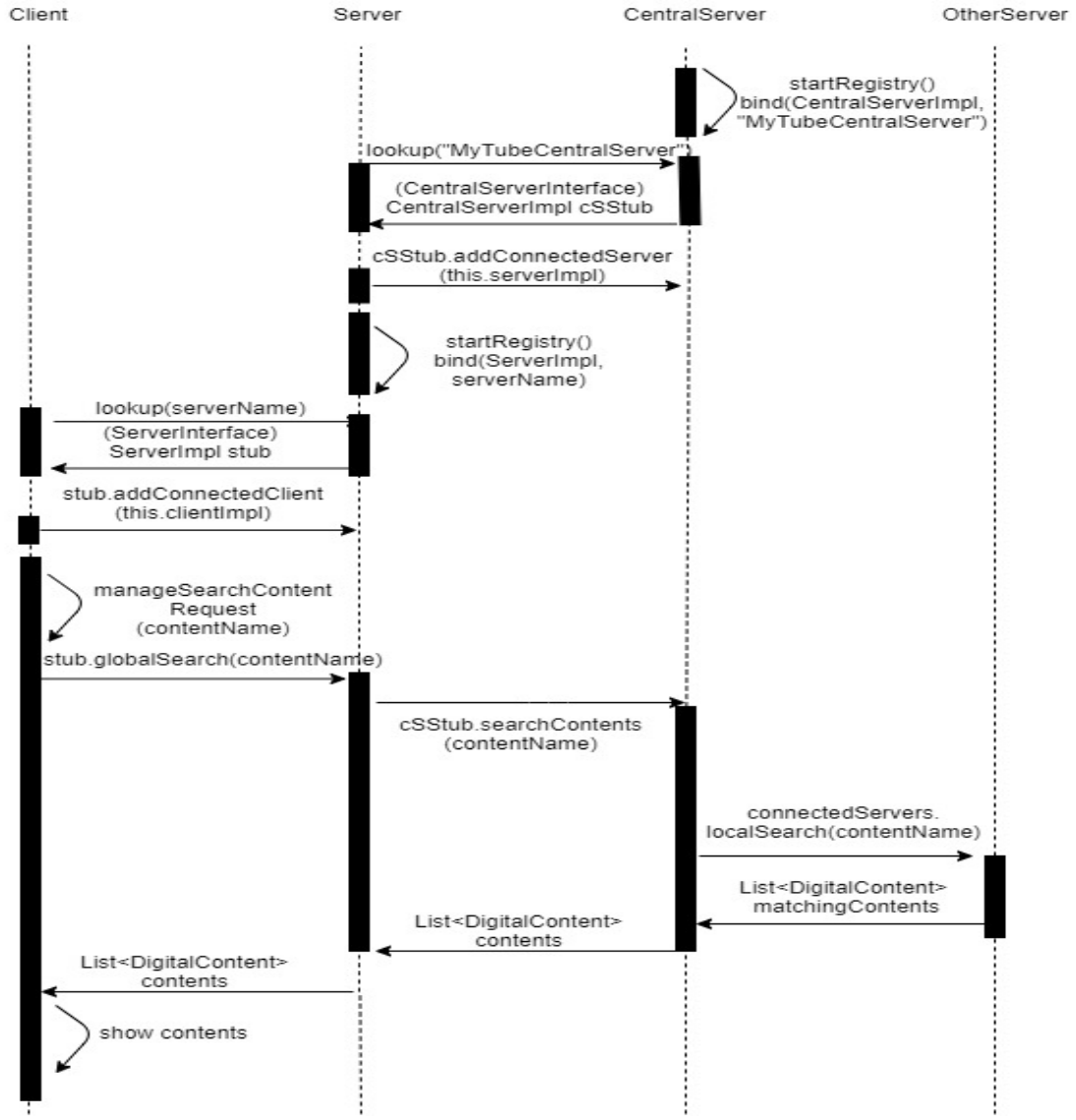

Figure 4: Global Search operation sequence diagram

6 Considerations

6.1 Unique key for all the contents in the system

All contents in the system will have a unique key thanks to server identifiers and content identifiers, this is discussed in section *3.1. Database and content storage*.

6.2 Physically storing the contents

The contents will be stored in folders as explained in section *3.1. Database and content storage*.

6.3 Reduce the number of transfers of digital contents

When a client searches for a content, only the information about contents matching the search will be returned (but not the actual content) using the DigitalContent class which allows for 1 transferral of an object or a list of objects containing all the relevant information about the content being searched. When a client attempts to download or upload a content only one matching content (or none if non content is found for the download operation) will be transferred.

6.4 Efficient global search

Our approach to an efficient global search is explained in section *3.4. Global Search*.

7 Execution

7.1 How to correctly execute

This section explains how to correctly execute the project in sequential order. Each of the following can be opened as separate projects on an *IDE* such as *IntelliJ IDEA*: CentralServer, Server and Client projects and can be ran separately. In order to properly execute the project, the central server must be ran first, then the server/s and finally the client/s. For the **server part it is important to properly configure as dependency the *sqlite-jdbc-3.8.10.1.jar* which resides in the folder *contents***. This can be configured under *File* > *ProjectStructure* > *Modules* > *Dependencies* in *IntelliJ IDEA*.

7.1.1 Central server

First of all, we need to run the central server. The main class is called *CentralServer.java* which can be ran as ***Usage:*** *<**host**> <**port**>*, where:

- <**host**> (*mandatory*): refers to the host where the central server will start the RMI registry (the central server's address).
- <**port**> (*mandatory*): refers to the port where the central server will start the RMI registry (the central server's port).

For instance in case we wanted to run the central server's RMI registry at a local machine and port 1098 we would pass as arguments: *127.0.0.1 1098*. The following Figure shows how we would need to set it up in *IntelliJ IDEA*.

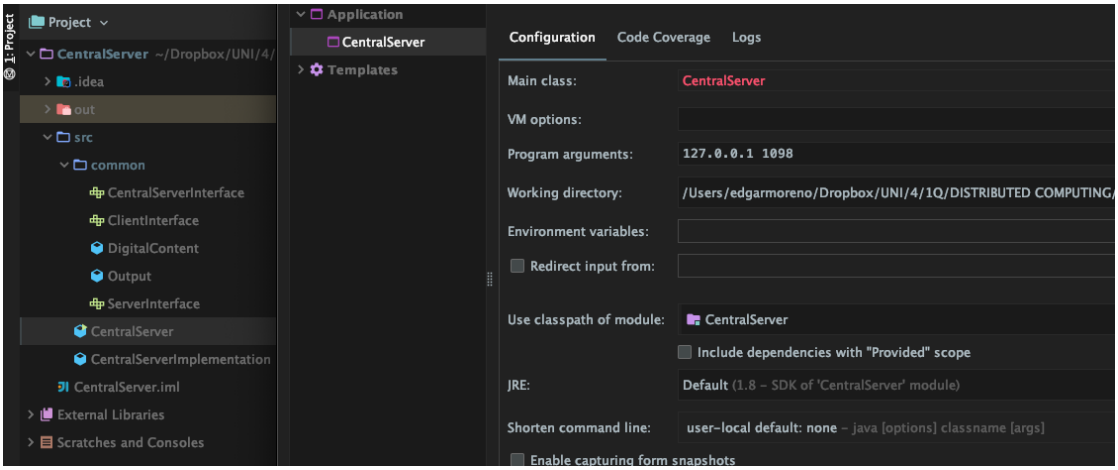

Figure 5: How to run the central server in *IntelliJ IDEA*

Another scenario would be if we wanted to connect to the central server from outside the local machine; if we wanted to contact the central server from a machine inside the network we would have to pass the internal IP address of the machine the central server will be run at as argument. In case we wanted to contact the central server from outside the network where the central server is, we would have to pass the external IP address of the machine the central server will be run at.

7.1.2 Server

Once we have the central server up and running we can connect to it by running a server or multiple servers. The main class is called *Server.java* which can be ran as ***Usage: <host> <port> <central_server_host> <central_server_port> [contents_db_name] [registry_name]***.

- <**host**> (*mandatory*): refers to the host where the server will start the RMI registry (the server's address).
- <**port**> (*mandatory*): refers to the port where the server will start the RMI registry (the server's port).
- <**central_server_host**> (*mandatory*): refers to the port of the central server hosting the RMI registry whose CentralServerImplementation object will be looked up.
- <**central_server_port**> (*mandatory*): refers to the port of the central server hosting the RMI registry whose CentralServerImplementation object will be looked up.
- [**contents_db_name**] (*optional*): refers to the name of the database where the server will store the contents information.
- [**registry_name**] (*optional*): refers to the name of the object ServerImplementation to bind on the registry started by the server at host:port. "*MyTube*" by default.

For instance, in case we wanted to run the server's RMI registry at a local machine and port 1097, bind an object named *MyTube* to it, create a database named *contents* and finally connect to the central server started in the previous section we would pass as arguments: *127.0.0.1 1097 127.0.0.1 1098 contents MyTube*.

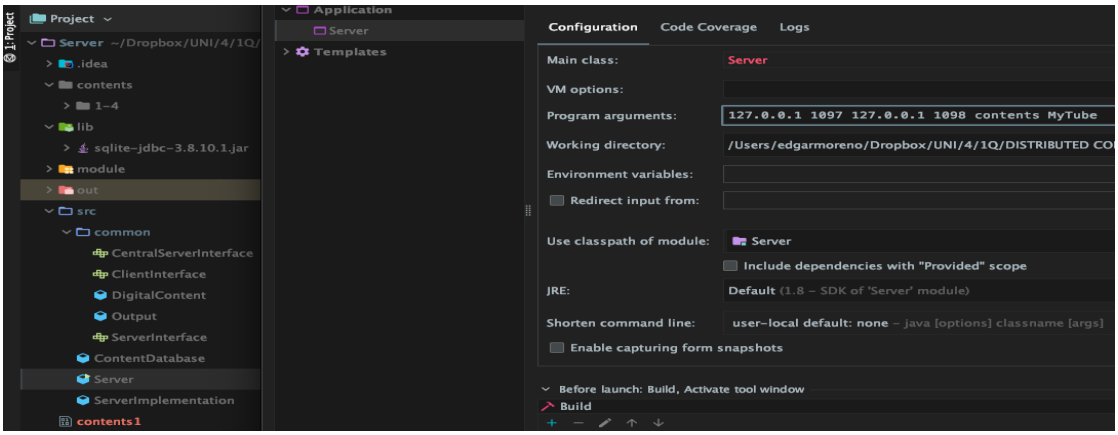

Figure 6: How to run the server in *IntelliJ IDEA*

7.1.3 Client

Once we have the server/s running we can run the client/s. The main class is called *Client.java* which can be ran as ***Usage: <host> <port> [registry_name]***, where;

- <**host**> (*mandatory*): refers to the server hosting the RMI registry.
- <**port**> (*mandatory*):: refers to the port of the server where the RMI registry has started.
- [**registry_name**] (*optional*): refers to the name of the object ServerImplementation to lookup on the registry which was binded by the Server. It is "*MyTube*" by default.

For instance, in case we wanted to connect a client to the server started in the previous section we would pass as arguments: *127.0.0.1 1097 MyTube*. The following Figure shows how we would need to set it up in *IntelliJ IDEA*.

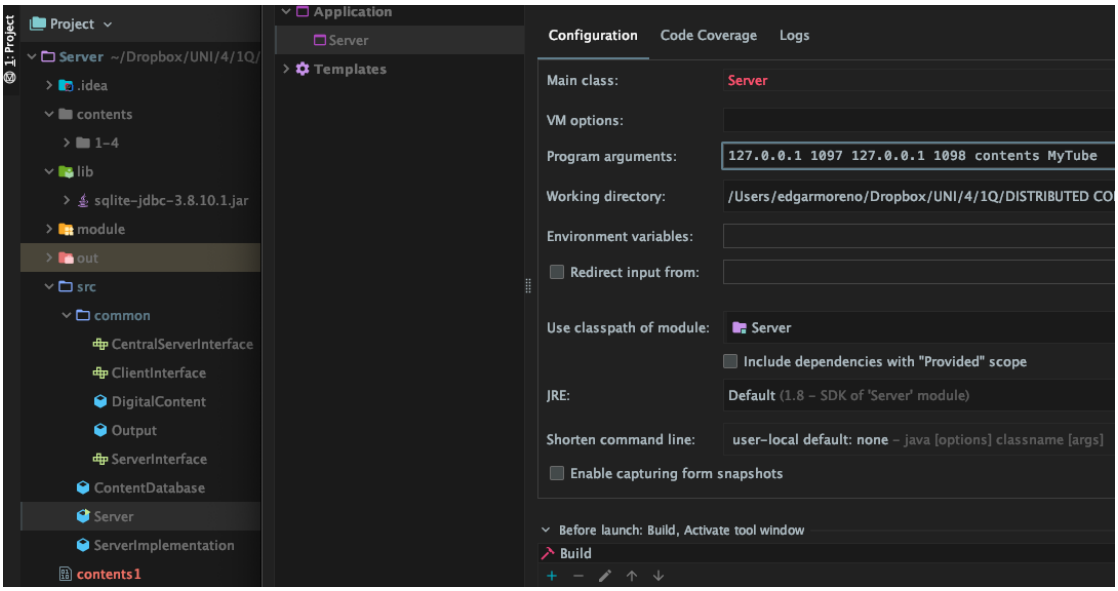

Figure 7: How to run the client in *IntelliJ IDEA*