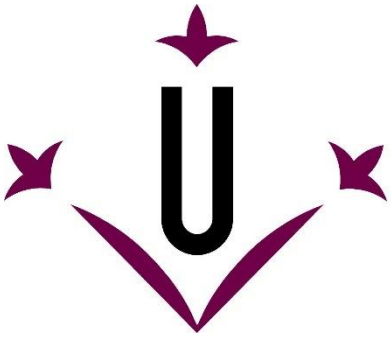


Universitat de Lleida
Escola Politècnica Superior

Bachelor's degree in Computer Engineering

Subject:

DISTRIBUTED COMPUTING

Project 1 – RMI

Authors:

Camara, Florin

Taló López, Pau

Professors:

Josep Lluis Lerida Monso

Eloi Gabaldón Ponsa

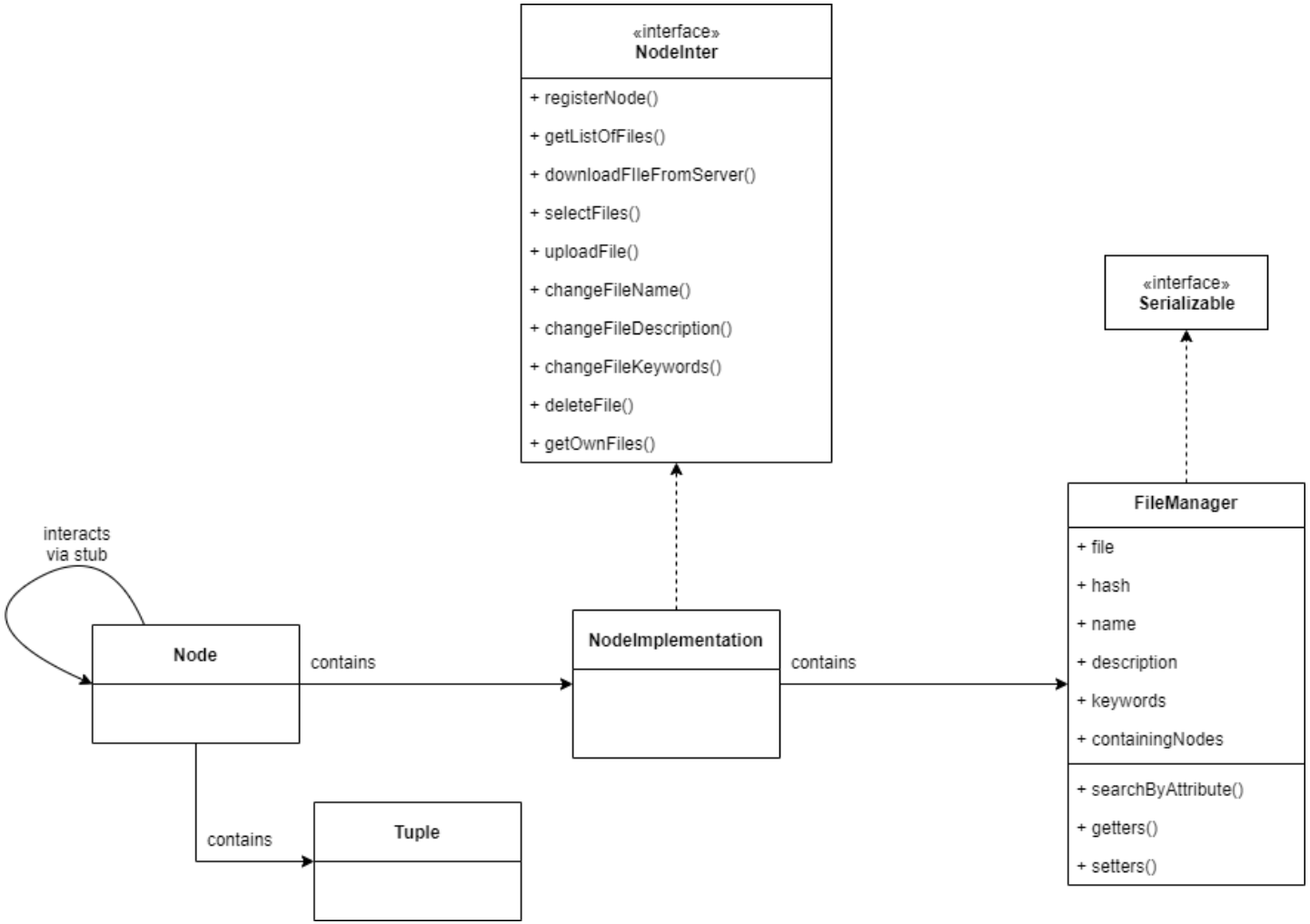
Jorge Gervás Arruga

15th of December, 2021

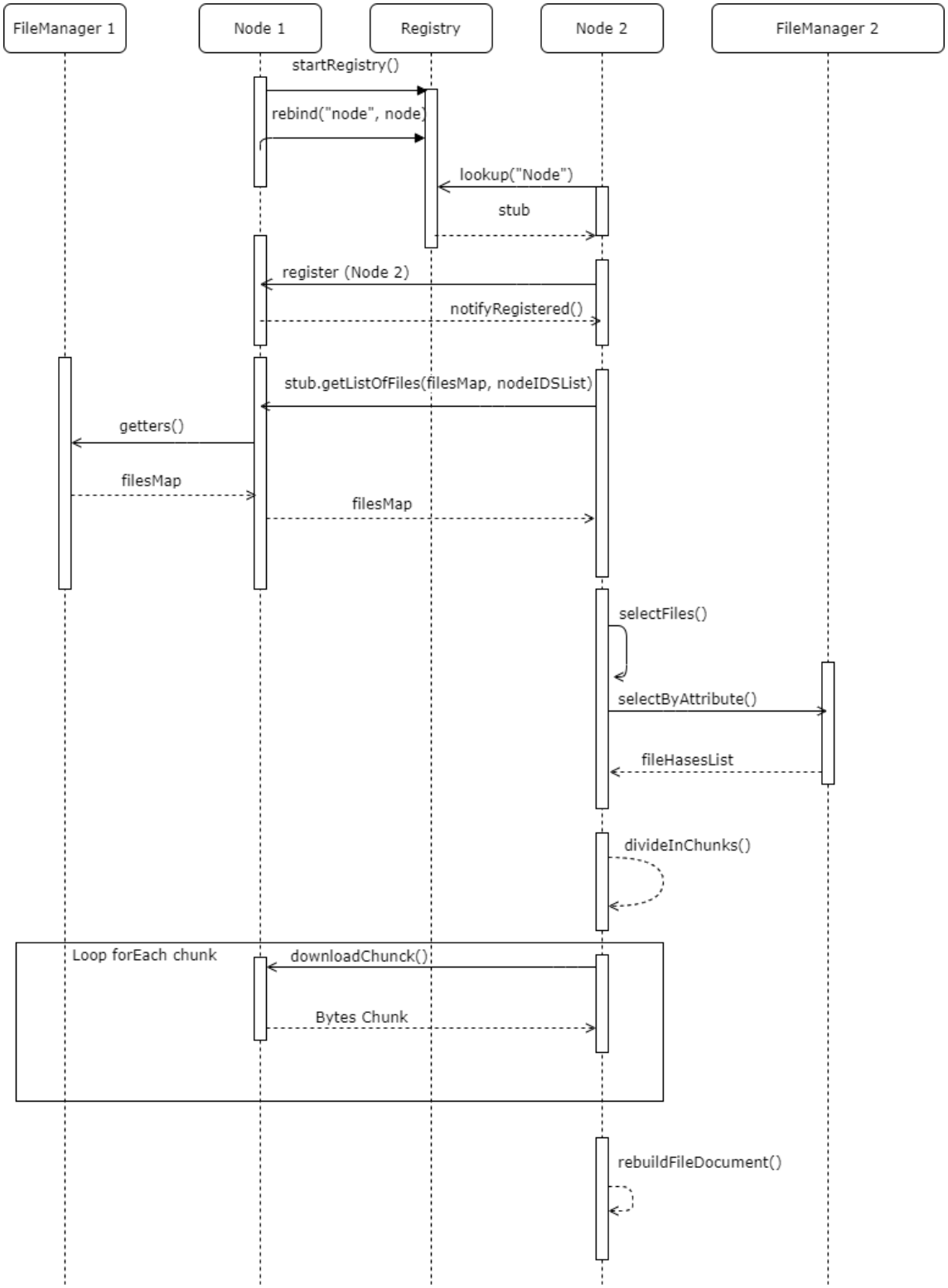
Course: 2021 – 2022

Contents

1. UML CLASS DIAGRAM



2. SEQUENCE DIAGRAM



3. DESIGN DECISSIONS

Technology

First of all, the technology we have decided to use in order to develop this project is Java RMI, although not apporting any real help in developing a peer-to-peer network, this technology was introduced in class and seemed easy to use, specially when stablishing a connection to another peer as long as we know its IP address and exposed port.

Architecture

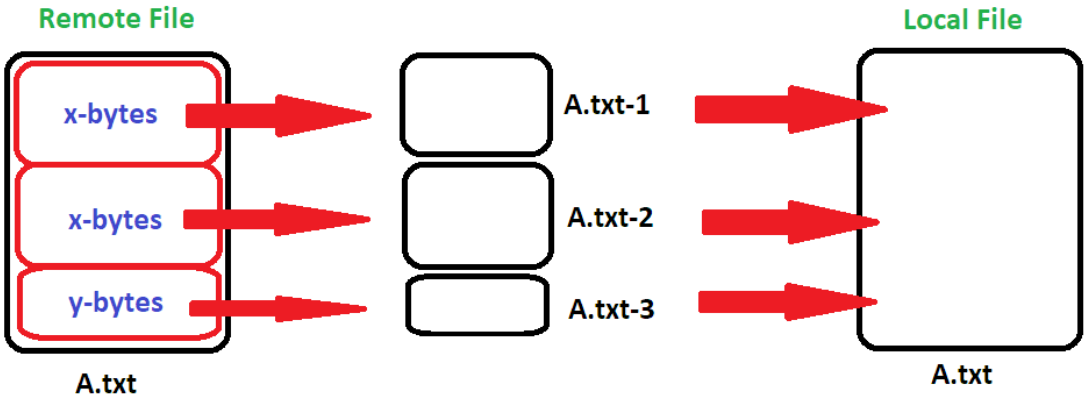
When we first started reading this projects statement and after identifying the main functions that a peer should be able to do, the first decision we took was to create an auxiliary class named 'FileManager' in order to, as its name implies, handle all the file managing a peer would like to do (i.e., setting a name, description or keywords).

Of course, a part from the 'FileManager' there is also a main class named 'Node', which is the one in charge of the flow of the program. This class named 'Node' contains an inner private class named Tuple, which is in fact a representation of an object similar to Python tuples, a pair of values. This class is only used to store other peers IPs and ports in a Map as the key just in case there are nodes with the same IP addresses but different ports. Furthermore, we also have a class named 'NodeImplementation' implementing the 'NodeInter' interface, which is in charge of defining the main functions a peer should be able to carry out in the network.

Considerations

Physical file transfer and workload balance

In order to download digital contents from a peer to another, we decided to split the files in different chunks of 1MB or maybe less only for the last chunk. The requesting node would use a 'parallelStream( )' to request a certain chunk of the file directly to the serving node, without passing this value through any other nodes in the network. 'Collection.parallelStream( )' allows us to parallelize the requests of file chunks to the same source, making the application more efficient downloading files. Each stream will download the file data to a different temporal file, and when all the streams are done downloading, the main thread remains in charge of recomposing the file from those temporal file fragments. These fragments name come with the identifier of the chuck position corresponding to the source's original file, so if a file A.txt has 3 chunks, the temporal file fragments would be the following: A.txt-1, A.txt-2, A.txt-3.



If there are more than one peer that own the requested file, in order to balance the requests among the network, we have implemented a Round-Robin chunk assignment queue between the different owners. Moreover, if the requesting peer tries to request a file that it already has, the request will be ignored, and a detail output will pop up.

Efficient search

As far as efficient search goes, we decided to give each peer a unique random ID generated by the 'UIID' class. When a peer requests the list of files among all the peers in the network, this is done via requesting the all the 'FileManagers' information recursively calling this same method among all the immediately connected peers. This method has a list of IDs as argument, and whenever a targeted peer receives this call, it first checks if its own ID is contained in the list of IDs before recursively calling this method to its neighbours. This way we can rest guarantee that there will never exist an infinite request loop.

Content location transparency

The 'FileManager' class provides this type of transparency, since the requesting peer can select any file from the list of files among the network, previously requested, by making a partial search on either their hash code, name, description or keywords. So, the requesting peer never really gets to know the specific location of this file, neither the peer containing it or its specific path inside that peer.

Security

The only security measures we took were to allow a peer to edit the information about a certain file only if its ID is contained in the owner IDs list of the 'FileManager'.

4. RUNNING THE APPLICATION

In order to correctly run this application, the user must first decide to either start a new peer-to-peer network by starting an isolated peer or to start a peer connected to an already existing network.

If the user opts for running an isolated node the program arguments should be:

```
port-to-expose path-to-folder
```

Where path-to-folder corresponds to the path to the folder where all files inside it will be listed as downloadable and where in the future all requested files will be downloaded.

On the other hand, when opting for a connected peer, the application arguments should be:

```
port-to-expose path-to-folder host-ip host-port
```

Where host-ip and host-port correspond to the IP address and port of the peer to connect to respectively.

From here on, an easy-to-follow menu with the proper options will pop up to guide the user to managing its local files or those ones across the network as we can see in the next example of how to download a file:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrain
RMI registry cannot be located
RMI registry created at port 1099
Connected to Node 192.168.1.131: 1098
Client registered, waiting for notification

--------------------------------------
|                                    |
|        WECOLME TO P2PNetwork       |
|                                    |
--------------------------------------

WHAT DO YOU WANT TO DO?
1 - UPLOAD FILES
2 - SEARCH & DOWNLOAD FILES
3 - EDIT YOUR FILES
4 - DELETE YOUR FILES
5 - EXIT

2
Which file do you want to download?

--------------------------------------------------------------------------------
Hash: 8c260cf0ce4818807e4a3c7222ef7280e50c4c2d1ab46544919f68e7936ba21a
Name: [CP2008_IZZ_Montage.mp3]
Description: []
Keywords: []
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
Hash: 229bc04c6382ef44c4f94927630f4bcdf6073b16a3b09b94c96d4b0090635d51
Name: [wallhaven-ymo2ek.png]
Description: []
Keywords: []
--------------------------------------------------------------------------------

Enter attribute to search by:
1 - Hash code of the desired file
2 - Name
3 - Description
4 - Keywords
2

Enter value to search in 'name'

wallhaven

Downloading file 'wallhaven-ymo2ek.png' ...
Download Completed!


Your files have been updated:
```

5. CONCLUSIONS & POSSIBLE IMPROVES

To conclude, we can say that RMI, although being a language dependent technology, it is pretty easy to use and useful for small sized projects since it belongs to the Remote Method Calling paradigm and provides less abstraction than other paradigms like Web Services.

In our honest opinion, we think that making this project peer-to-peer oriented rather than making it client-server oriented, like all the example codes we have seen in class, has added a bit of difficulty, messing with our ideas at some point. Even though a peer could be seen as a client and a server at the same time, which sounds as a simple idea, but, at least in our case, we have faced some difficulties implementing it.

For possible improves of this project, we could consider using a local database instead of the current 'FileManager' class. This would make the application far more scalable while keeping a similar execution flow and even improving data consistency.