Peer to Peer file sharing using CORBA

Ugaitz Amozarrain
Mikel Iturbe
Ibai Osa
Unai Perez

June 14, 2011

Abstract

The report goes through the building of a P2P application using CORBA. Following the unified process, diagrams with increasing maturity level are sketched chronologically, starting with the idea of what the program should do (writing the user case diagram and the IDL file) and finishing with actually how the program does it (case diagrams). The start of work is in the Methodology section, in which the start (first diagrams) and the way of working are explained. After that, the broader presentation section comes, with screenshots of the intended program and different sequence diagrams, with the aim of showing the interaction of the different parts of the distributed applications as clear as possible, instead of focusing in the software engineering side of the problem. Furthermore, each diagram has a small explanation to expand the understanding of processes.Finally, in the appendices, a mention of design patterns and their application in the project is made, along with the bibliography.

Contents

List of Figures

Introduction

A P2P program, essentially, is an application that is used to share content through the network. Its most important difference from other downloading methods (FTP servers etc.) is that there is no central node in which content is downloaded from and uploaded to. The content flows through a decentralized network of nodes, clients, who simultaneously work as servers (or seeders, because they seed content) and downloaders (leechers). The most famous of them is eMule or aMule (depending on the Operating system used).

To illustrate different steps in the development phase, different kinds of diagrams have been used, starting from the user case diagrams in the very beginning until the fully-detailed class-diagram, all of them constructed using UML (the Unified Modelling Language). To make possible having different ends working at the same time, the resulting application is a distributed one, built using a Corba (Common Object Request Broker Architecture) implementation in the Java language, JacORB.

The authors, Arrasaten, in Arrasate the June 14, 2011.

1 Methodology

Being the already cited problem the creation of a P2P (Peer to Peer) application, it is vital to first identify the aspects that a P2P program normally fulfills, in order to work correctly. A P2P program, essentially, is an application that is used to share content through the network. Its most important difference from other downloading methods (FTP servers etc.) is that there is no central node in which content is downloaded from and uploaded to. The content flows through a decentralized network of nodes, clients, who simultaneously work as servers (or seeders, because they seed content) and downloaders (leechers). They are also servers in the P2P network, but they have a different role:

Servers Servers are used to identify and contact different clients between them. They do not take part in the actual process of downloading or uploading content, that is the clients' job. If the server infrastructure falls down, it can be replaced with a new one; no data is lost.

Clients The core relies on the clients. Those are the responsible for downloading and uploading content, all the heavy traffic is done between clients, the server only helps meeting clients with similar interests. Normally, clients can stop and resume processes, and hold different processes at the same time (both uploads and downloads). More advanced features consists of bandwidth usage and a virtual crediting system to give priority to clients who share most.

Thus, if we leave the more advanced features aside, we have already defined what our program should do, with its two main parts: servers and clients. The server (a main one is enough to build a prototype) is the responsible to put clients together and clients are responsible for moving content. The client is the one an average user would use, so it is the one we will focus on. The server needs no user interaction.

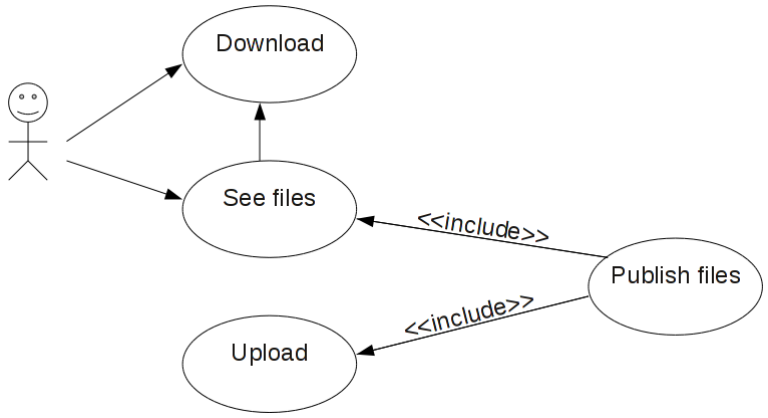

Figure (1): *User Case diagram of the P2P client*

The diagram no. 1 shows the User Case diagram of the client, from the point of view of the user. When the user starts the program, receives from the server the list of the files available (the ones that other users have published) and chooses which one of the files wants to download. Without the user's consent, the files he or she is sharing are sent

to other clients and if other clients want one of the files the user has, the upload of the content will also start automatically. Different uploading and downloading processes can be carried at the same time.

It has to be mentioned though, that the user can control the content he or she wants to upload, but not through this application. All the content the user wishes to share is stored in a directory called `ongoing`, so the user can use the Operating System to put or remove different files in the directory, but as said, it cannot be done using the program. Likewise, all the downloaded content is stored in a directory called `incoming` and once the file is downloaded, it is the user who places the file in the directory he or she wishes to.

Before continuing with in-depth analysis and design of the application, it has to be cleared how many connections are there going to be. It has been noted before, that the main connections (the ones sharing content goes through) are between clients and that clients only turn to servers to know what files are being shared. Once the desired file has been selected and download starts from another client (or various, depending on the case) the server just keeps updating the downloadable file list, based on the clients that connect and disconnect.

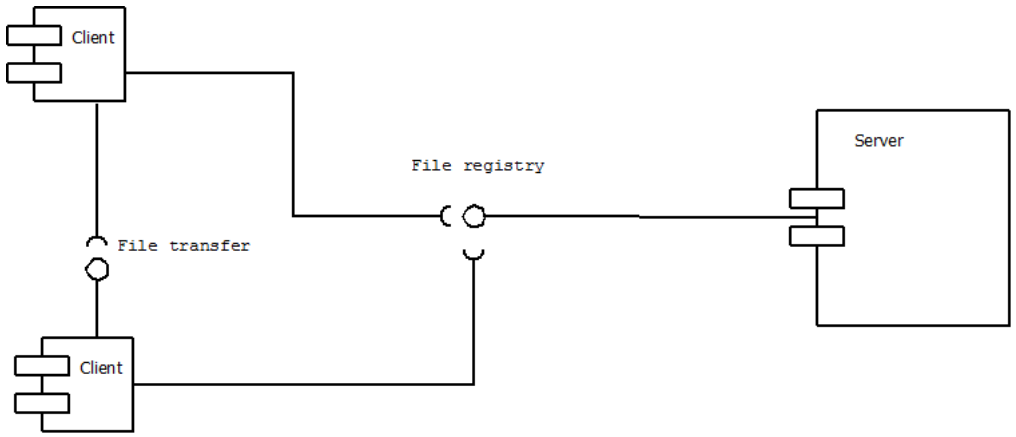

Figure (2): *Connections in the P2P system*

What has been told previously is summarized by the figure no. 2. The diagram shows how the clients connect to the server to register files (and to receive the last version of the list showing the available files), and once they have selected what file to download, the server puts in contact the two clients (here the simplest case is shown, but more clients can be involved in the process), the seeder (the one who uploads) and the leecher (the one who downloads) start transferring content, but they still maintain contact with the server, to update the file list and to have the possibility to connect to other clients.

Finally, it is important to note that although general diagrams of the working programs have been done *before* the application was developed, technically deeper diagrams that come later have been outlined *after* the application was developed, in order to be as accurate as possible. For the sake of cleanness, and to ease things as much as possible to the reader, diagrams that were outlined in order to guide the development of the program have not been included in this report, as those diagrams have gone through numerous changes from the original version until the final one.

Thus, programming started early enough to make non-structural changes to the main design, but the main aspects of the program remained the same. The IDL file, for instance, was defined early in the analysis stage and luckily, there has been no need to change it all along the development phase:

```
module erabilgarriak{
    struct FileData{
        string name;
        long long size;
        string hash;
    };
    interface DownloadFile{
        typedef sequence<octet, 1048576> Part;
        FileData getFileData();
        long getPartCount();
        long getPart(in long numPart, inout Part part);
    };
    interface Server{
        typedef sequence<FileData> FileDataArray;
        typedef sequence<DownloadFile> DownloadFileArray;
        boolean register(in DownloadFile file);
        boolean deregister(in DownloadFile file);
        boolean getLista(inout FileDataArray files);
        boolean getFile(in FileData data, inout DownloadFileArray files);
    };
};
```

All the development has been done using VCS (Version Control System), to make development as easy as possible and to minimize possible version collisions. Parallel to the development, documentation has also been done step by step, using VCS as well, taking advantadge of the capabilities of LaTeX to be used in one of those systems. In the next section, all the application is explained, using different diagrams and screenshots. Even those are used to explain the correct functioning of the program, they can be seen as the next step in the unified process of the creation of the Peer to Peer program.

Until now, only the main lines of the program have been set. Once the main components and actions of the program are defined, it is necessary to jump to the next layer: what has been done, using the methods already mentioned, to achieve the program explained in the user case diagram (figure 1).

2 The product

2.1 Layout

2.2 Working process

In this section, the main functionalities of the program are explained, one layer deeper that has been shown in the Methodology section. Here, in-depth schemas are in place, using mainly Sequential UML diagrams. It has to be mentioned, though, that the text in those diagrams are in Basque language, the same language the codification of the program has been made. That way it will be simpler to compare the diagrams with the code rather than translating function and object names into English just for the diagrams. But, on the contrary, all the explanations of the diagrams are done in English. Also, the bulk of the diagrams focuses on the client, because it is the part where the real work is done. As said, servers just put clients together. Not all the code is featured in the diagrams (graphical interfaces etc. have been left apart), and just the distributed nature of the application is shown.

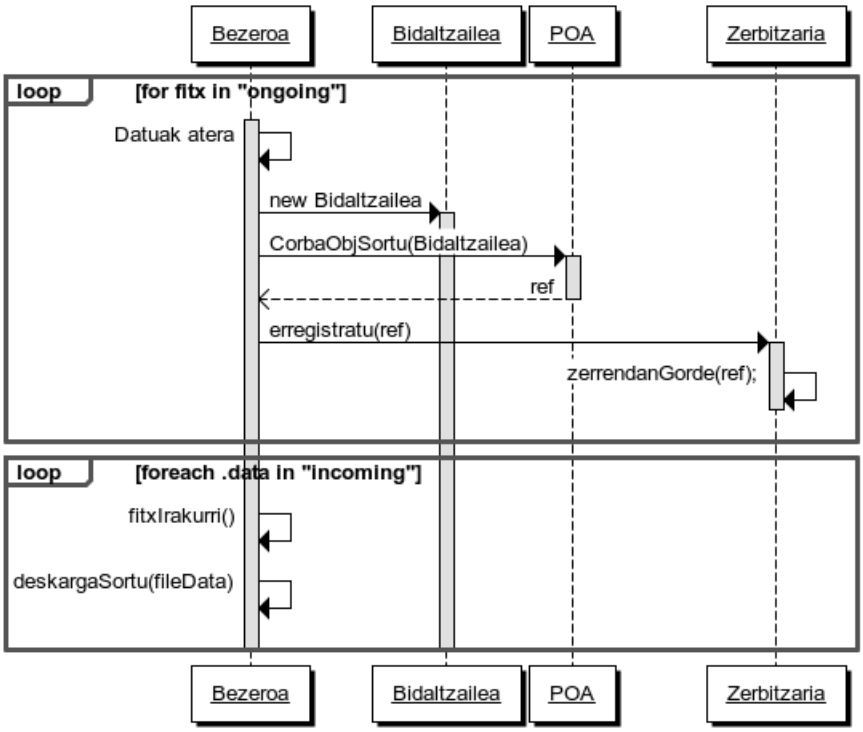

Figure (3): *Start of the P2P client*

In the figure no. 3 it is possible to see what the client does when it is started. First, extracts the identification data of the file (file name, hash as defined in the IDL, look section 1), after creates the `Bidaltzailea` object, whose responsibility is to upload the content the user is sharing. Finally, it creates a Corba object with it and registers it in the Server. This process is repeated with all the files in the `ongoing` directory.

Figure no 4 shows the development of the download (`Deskarga`) process. Each download is initialized with one `SeedChecker` and one `Gordetzaile` object and it is

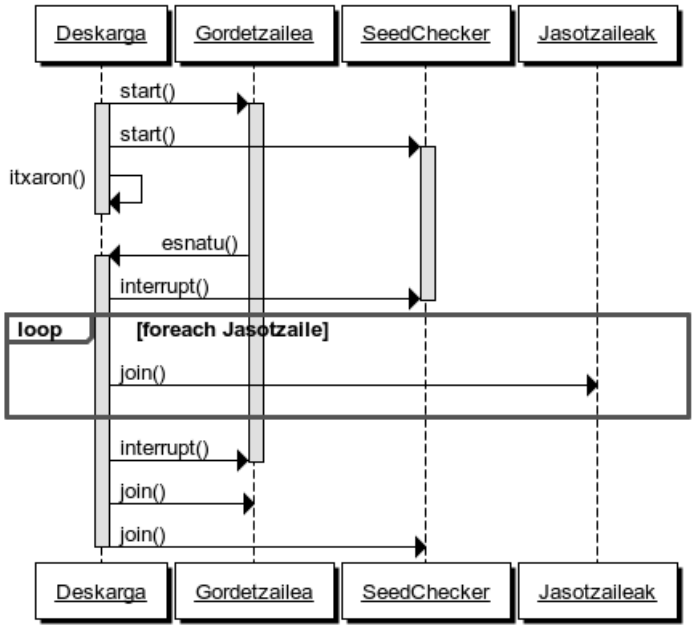

Figure (4): *Run of the* `Deskarga` *thread.*

the `Deskarga` object who starts the `Gordetzaile` and `SeedChecker`, and when the `Gordetzaile` tells the `Deskarga` that it is awake, that is, the download is complete; interrupts the `SeedChecker`, waits to all the `Jasotzaileak` threads (each one downloads the content from a different client), interrupts the `Gordetzailea` and waits until all the remaining threads are finished.

The process of retrieving the seeds from the server is done by the `SeedChecker` thread (figure no. 5) and each . Starts retrieving the seeds from the server, with the data extracted from the file (see the `Datuak atera` process in figure no. 3). The server creates a Buffer list of the seeders and returns it to the client. The process described next is done using a mutual exclusion semaphore, to make sure that the objects mentioned are not manipulated at the same time.

The checker checks all that the seeders actually all the seeders exist or that are not in the `SeedLista` object, because that means that a file is being downloaded from them at the moment. As shown in the diagram no. 4 the client creates a `Bidaltzailea` for each file, which is, in fact the seeder from the file is downloaded. Thus, having a seeder in the `SeedLista` means that the client is already downloading the same content from it and therefore is removed from the buffer.

Once the duplicated and inexistent seeders have been put away, for each seeder from the now clean buffer the system checks if the active seeder number (each active seeder has an active `Jasotzaile`) is less than the maximun one (`JasoMax`) and if it is, the `SeederChecker` registers the seeder in the `SeedLista SeedErabLista`. This last object is used to store the seeders that are actually in use, to prevent creating a new `Jasotzaile` with a seeder already in use. A `Jasotzaile` is created with a seeder that is in `SeedLista` but is not in `SeedErabLista`. After the creation of the Jasotzaile, the checker adds the newly created one to the `JasotzaileLista` list. But, if the active seeder number exceeds the maximun one, the new seeder is just registered in the `SeedLista` not in the

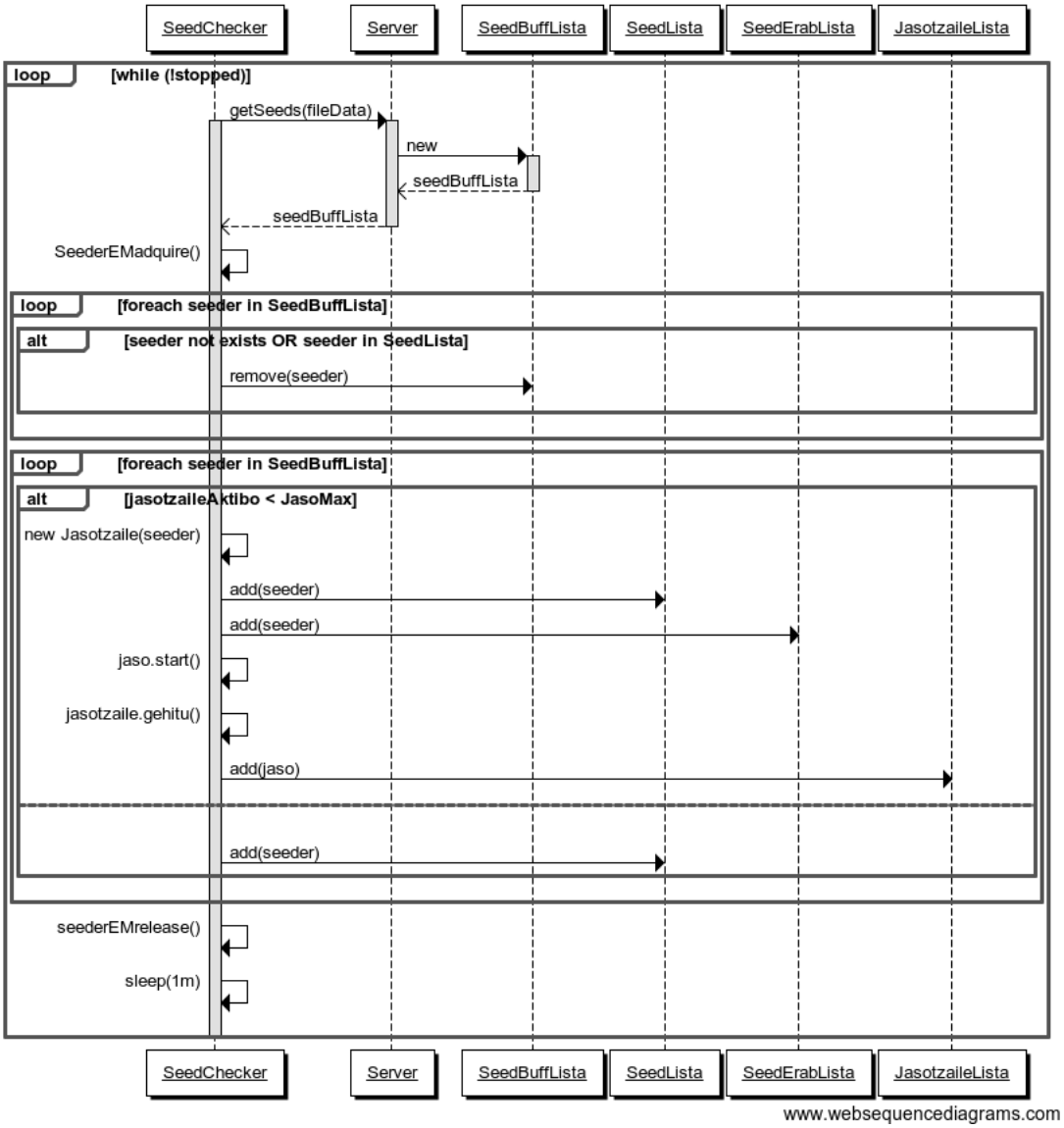

Figure (5): *Run of the* `SeedChecker` *thread.*

SeedErabLista. Finally, the semaphore opens again and the process sleeps until is rerun again.

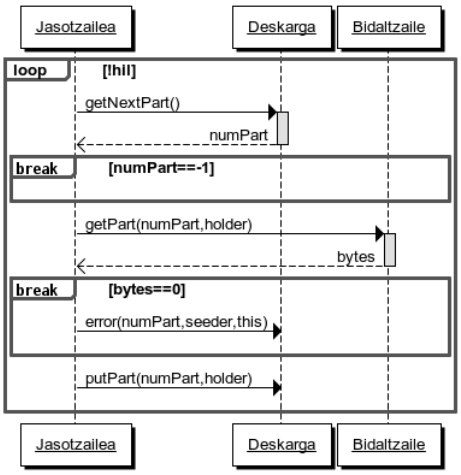

Figure (6): *Run of the* `Jasotzailea` *thread.*

The next process, concerning the retrieving of files, is done by the thread `Jasotzailea`, as shown in figure no. 6. It has already been mentioned how a `Jasotzaile` is created by the `SeedChecker`. Them the newly created object asks for the number of the next downloadable part. If that number is `-1`, the thread is interrupted. If it is not, asks to the `Bidaltzaile` of the seeder the next part of the file. If the part received has a size of 0 bytes, the thread files and error and starts the loop again. Unlike the previous break, this break does not interrupt the thread, just starts the loop again. If some bytes are received, the `Jasotzailea` just forwards those to the `Deskarga` object. All this it is done if there is no exceptional error which changes the `hil` value to `true`.

The process of storing the file is controled by the `Gordetzailea` thread (shown in figure no. 7), which basically writes what has been received. All the process is regulated by a mutual exclusion semaphore, so no concurrence problems occur. The object is created with the data retrieved by the other client (see the `Datuak atera` process in figure no. 3). It basically writes received parts to the file (`Fitxategia`) and writes to a

`.data`

(`datuFitxategia`) file what has been the last part downloaded, so the download can be resumed in the same place if the process is interrupted (in case the network goes down, for instance). After, if the downloaded part number is the same as the number of parts the file has, the semaphore opens, the auxiliary

`.data`

file is erased and the `stopped` variable is set to true, so the process does not continue. If there are parts still to store, the process just repeats again.

The last process shown in this report (figure no.8), is the one that affects the sending of files. Unlike the previous ones, this process is not started with an explicit `start()` method, Corba itself executes it in a separate thread, as the reincarnation of the methods

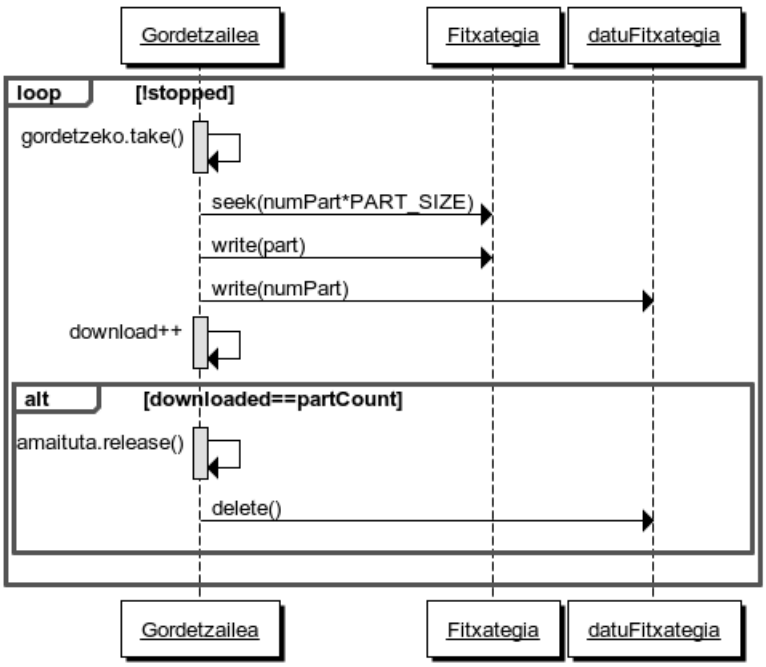

Figure (7): *Run of the* `Gordetzailea` *thread.*

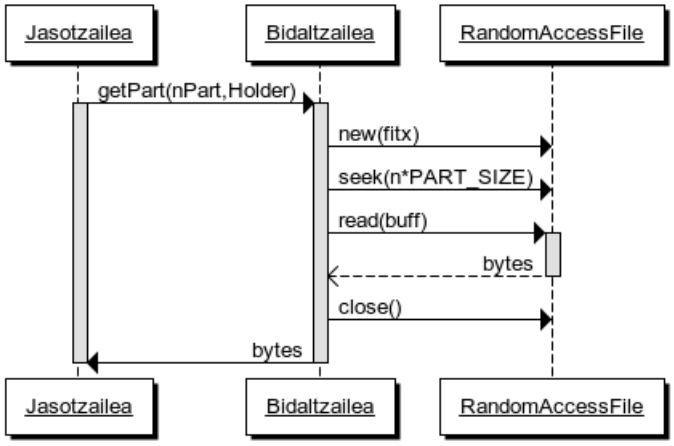

Figure (8): `Bidaltzailea` *sending files method.*

stated in the IDL file. So, when a `Jasotzailea` asks for a new part, the `Bidaltzailea` reads the bytes concerned and sends them back. The reason to use RandomAccessFiles is that the file can be read simoultaneously and can be chosen which part is read, using the `seek()` function.

With this six processes, the main functions of how our P2P program shares the files is explained, leaving secondary explanations aside.

2.3 Program structure

To achieve the functionalities presented in the previous section, it is necessary to go down to the code, to see how it is arranged and how different methods interact with each others. But, in this case, leaving the code apart, class diagrams are going to be shown, the last step of the unified procedure of this program. In those, methods and variables appear, ordered by the class they belong.

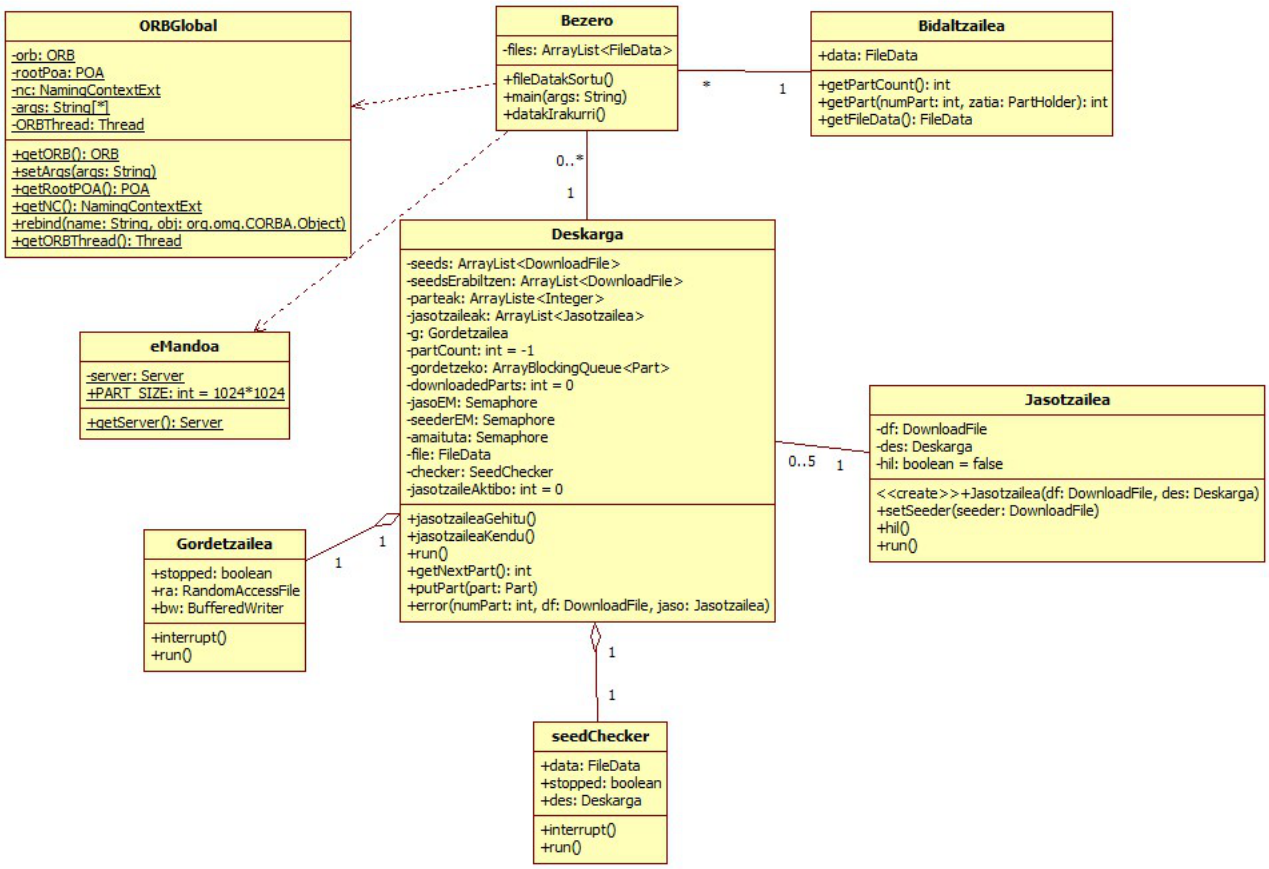

Figure (9): *Class diagram of the client*

In both diagrams, all the methods mentioned in the previous section appear ordered, as well as the cardinality of the relation between the classes that host them. From those clases, `Bidaltzailea` is the reincarnation of the `DownloadFile` defined in the IDL and `Zerbitzari`, the one of `server`.

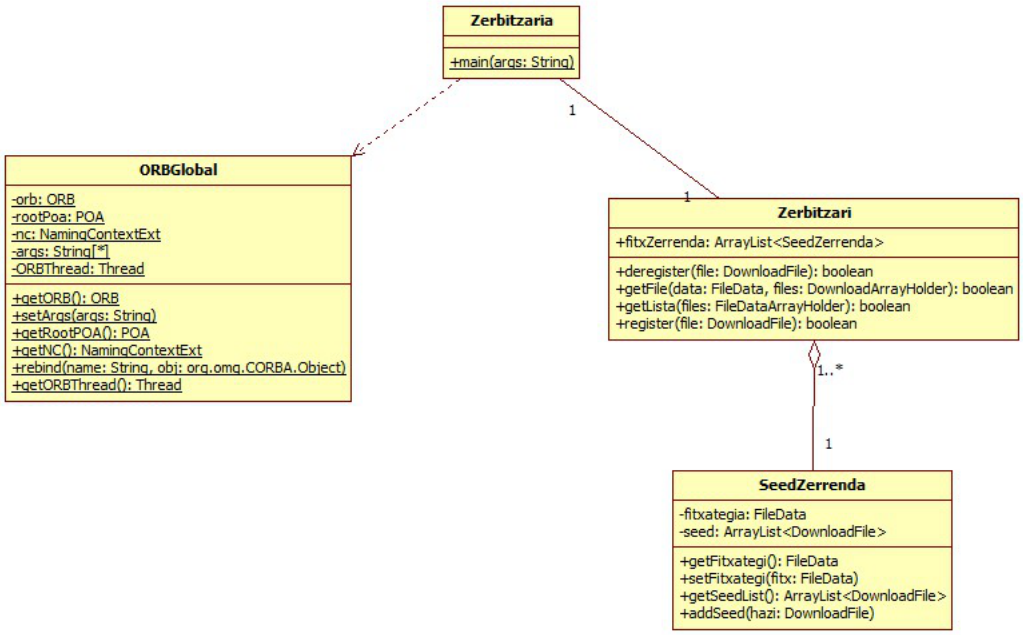

Figure (10): *Class diagram of the server*

Appendices

A Design patterns

Introduction

Design patterns usually are pieces of design that can be easily reusable, and that has been created with that objective in mind. They can be considered as a general template, so it makes possible to adapt them in similar situations. Mostly, they deal with relationships between different classes, without being too explicit with code. They are different types of design patterns:

Algorithm strategy patterns

Computational design patterns

Execution patterns

Implementation strategy patterns

Structural design patterns

* Algorithm strategy patterns addressing concerns related to high-level strategies describing how to exploit application characteristic on a computing platform. * Computational design patterns addressing concerns related to key computation identification. * Execution patterns that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization. * Implementation strategy patterns addressing concerns related to implementing source code to support

1. program organization, and 2. the common data structures specific to parallel programming.

* Structural design patterns addressing concerns related to high-level structures of applications being developed.

In our case, an implementation design pattern is used, which is implemented in the `Globalak` class. This class can be exported to almost any other CORBA application, achieving a functioning distributed application powered by Corba with minimal effort.

B Bibliography

THE JACORB TEAM. *JacORB 2.1 Programming Guide* [online]. The JacORB Team, 2004.
http://jmvanel.free.fr/corba/doc/jacorb-ProgrammingGuide.pdf [Last consultation: 6-13-2011]

GITHUB. *GitHub:Help* [online]. GitHub, 2011.
http://help.github.com/ [Last consultation: 6-13-2011]

GAMMA, Erich; HELM, Richard; JOHNSON, Ralf; VLISSIDES, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley Professional, 1994.