

**Design Document**

Homework 1

Adv. & Distributed Programming Paradigms

CSC 3374 – 01

Fatima El Kabir

**Table of contents:**

1. **Overview…………………………………………………………………………………………. 3**
2. **Objective…………………………………………………………………………………………. 4**
3. **Used Commands……………………………………………………………………………… 4**
4. **xFx Protocol…………………………………………………………………………………….. 5**
5. **Code Description…………………………………………………………………………….. 6**
6. **Screenshots……………………………………………………………………………………….. 7**
7. **Overview**

The **Extended File Exchange Application (xFx)** is going to be built using the client/server model. This design decision aligns with a clearly specified system structure. Additionally, we have designed the xFx Protocol, which is structured and clearly defined protocol that will be used to enable efficient communication between the client and the end-user. This protocol acts as the intermediary language to enable easy communication between the client and the end-user, allowing the latter to transmit the necessary information and data to the server-side of the system. The structure of the xFx application is further explained as follows:

A green and white squares

Description automatically generated

1. **Objective**

This homework intends to expand the features of the Fx App, and this expansion will result in the creation of a new application called xFx, which will introduce a set of valuable features to extend the existing functionality of the Fx App. These improvements will provide users the ability to:

1. **Get the list of files that are shareable by the server**
2. **Resume the file download where it stopped, if there was a network connectivity issue**
3. **Save time and network bandwidth by not redownloading an already downloaded file, provided that its local copy has not become dirty/stale**

We will be using Java as the programming language to implement our implementation logic since it offers a strong and flexible platform that aligns well with our development goals and requirements.

1. **Used Commands:**

End users must utilize a set of particular commands, as listed in the table below, for them to interact with the xFx client efficiently. These commands allow users to communicate and interact with the xFx client, resulting in a seamless and simple user experience.

|  |  |
| --- | --- |
| Type | Command |
| Download | “d filename” |
| Upload | “u filename” |
| List | “l” |
| Resume | “r filename” |

1. **xFx Protocol**

**Download:**

* If the client wants to download a file that is not available locally, it must request it from the server. Then, the header for this request should follow this format:
* **download [one space] [file name] [Line feed].**
* Upon receiving this header, the server searches for the specified file, If the file is found on the server, the server responds with the following header:
* **OK [one space] [file size] [Line feed]**
* followed by the file's content.
* If the file is not found on the server, the server responds with the following header and then informs the user through a print statement.
* **NOT FOUND [Line feed].**
* If the file exists locally, it needs to be checked to determine if it's dirty/stale or not. The header for indicating that the file exists in the client share should follow this format:
* **exist [one space] [file name] [Line feed].**
* Upon receiving this header, the server creates an MD5 hash for the file in the server and converts it to a hexadecimal string representation, the client on the other hand prepares its MD5 hash and converts it to a hexadecimal string representation, Then, sends it to the server through the following header:
* **hash** **[one space] [Client Hash] [Line feed].**
* Upon receiving this header, the server checks if the file is dirty or not by comparing the two hashes representations. If the file is not outdated meaning the two hashes are equal, the server responds with:
* **Noneed [Line feed].**
* If the file is dirty meaning the two hashes are different, the server responds with
* **OK [one space] [file size] [Line feed]**
* followed by the updated file content.

**Upload:**

* If the client wants to upload a file, the header should adhere to the following format:
* **upload [one space] [file name] [one space] [file size] [Line Feed].**
* Subsequently, the file's content follows and sent to the server side if the file exists. Otherwise, an error message is displayed informing the user that the file is not available.

**List Files:**

* When the client wishes to get the list of shareable files, it sends the following header:
* **list [Line feed].**
* Upon receiving this header, the server proceeds and checks if the files list is empty or not. If it is empty, the server sends the following header:
* **empty [Line feed].**
* Otherwise, if the files list is Not empty, the server proceeds to send the respective files containing the essential information to list the file names to the client to print them.

**Resuming download:**

* When the client wishes to resume a file download, it should use the following header:
* **resume [one space] [file name] [Bytes Received] [Line feed].**
* After receiving this header, the server prepares how many bytes are left for the file to be fully downloaded, and responds with the following header:
* **acknowledgment [one space] [Bytes Remaining] [Line feed].**
* Followed by the remaining bytes required to finish the download.
* If the file has already been fully downloaded, the server signals this by sending the following header:
* **downloadComplete [Line Feed].**

1. **Implementation Logic (The code is already explained through comments in the source code)**

Once the connection is established between the client and server, the program takes command argument from the user, the command might be “**d**”, ”**u**”, ”**l**”, or “**r**”. if the user specifies the “**d**” command, he/she needs to specify the file name to be downloaded as well. Then the client side will check if the file already exists in the client share. If it does not exist, the file gets downloaded from the server share if it is found. Otherwise, we generate MD5 hash for both files on each side, and check if they are the same version by checking equality. If it is the same file, there is no need to redownload it. In case the files are different, the file will be redownload. If the command specified is “**u**”, the user should specify with it the file name to be uploaded. Then, the file will be uploaded to the server share. If the user specified the “**l**” command, the list of files in the server share will be printed to the user if it is not empty. Finally, if the command is “**r**” plus file name, the client side sees the bytes that are received and sends them to the server side. The latter on the other hand computes the bytes that are left to complete the download and sends them to be appended to file. If the file is already fully downloaded, there is no need to resume anything.

1. **Screenshots**

**Download:**

* **In case of a non-local file:**

**A screenshot of a computer

Description automatically generated**

* **I am going to download Hello.txt :**

**A black screen with white text

Description automatically generated**

* **As you can see, the file gets downloaded to the Client share:**

**A screenshot of a computer

Description automatically generated**

* **In case of a local file**
* **If I try to redownload it again, it will not allow me because it is the same version:**

**A black background with white text

Description automatically generated**

* **In case the file has been modified from either the server share or client share (DIRTY), the file will be updated to reflect the same version, let’s modify Hello.txt**
* **The file originally contains the word “hello”, let’s modify it to “Test Dirty” in client share and see if it is going to be updated**

**A screenshot of a computer

Description automatically generated**

* **Executing the command:**

A black screen with white text

Description automatically generated

* **As we can see, the file has been updated to “hello” the version of server share**

A screenshot of a computer

Description automatically generated

**Upload:**

* **I going to upload File1.txt from client share into the server share:**

**A screenshot of a computer

Description automatically generated**

* **Executing the command**

**A black screen with white text

Description automatically generated**

* **We can see that the file is uploaded into the server share**

**A screenshot of a computer

Description automatically generated**

**If I specified to upload a file that does not exist in the Client share, the following message will be printed: A black screen with white text

Description automatically generated**

**List Files:**

* **Currently I have 3 files in my server share:**

**A screenshot of a computer

Description automatically generated**

* **Executing the command:**

**A screen shot of a computer

Description automatically generated**

**Resume Download:**

* **I first created a resume.pdf file and I tried to cut the connection before the resume.pdf file gets fully downloaded to the client share.**

**A black background with white text and colorful letters

Description automatically generated**

* **As you can see the size of resume.pdf is 2 Ko, in the server share**

**A screenshot of a computer

Description automatically generated**

* **And only 1 Ko has been downloaded to the client share:**

**A screenshot of a computer

Description automatically generated**

* **Executing the command:**

**A black background with white text and colorful letters

Description automatically generated**

* **If we look now into the client share we will see that all the remaining bytes have been downloaded**

**A screenshot of a computer

Description automatically generated**