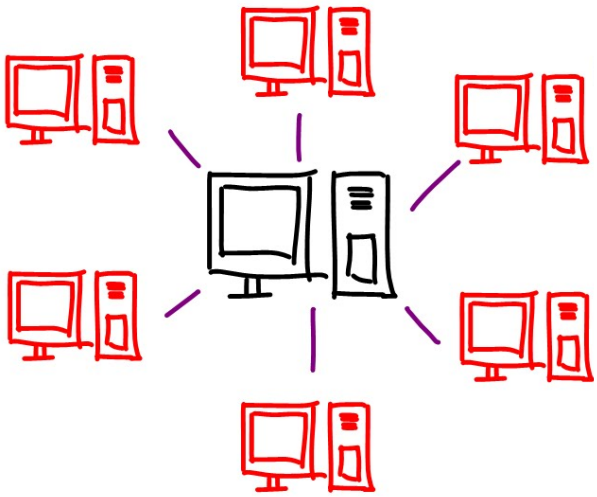
System Y: portfolio



Jannes Van Goeye

Robbe Pauwels

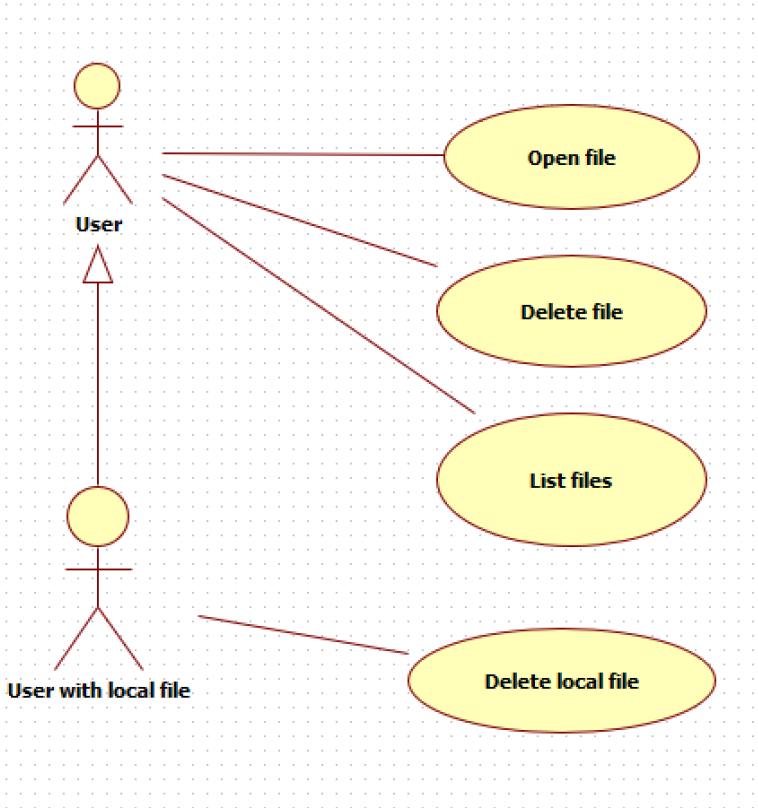
Imre Liessens

Introduction

System Y is a distributed filesystem: it's task is to store files in a network.

It's important that the system is robust and fail-safe. Otherwise, important files could be lost. A user's local files are automatically sent to at least one other node in de network and every other user can access the files that are stored anywhere in that network in an easy and transparent way (the user doesn't need to know about the exact inner workings).

Minimal configuration should be necessary for new nodes. That is, only the choice of which network interface to use (via it's associated IP) is a necessity when it comes to configuring a new node to join the system.



Figuur 1: use-case diagram

Assignment 1: Naming Server

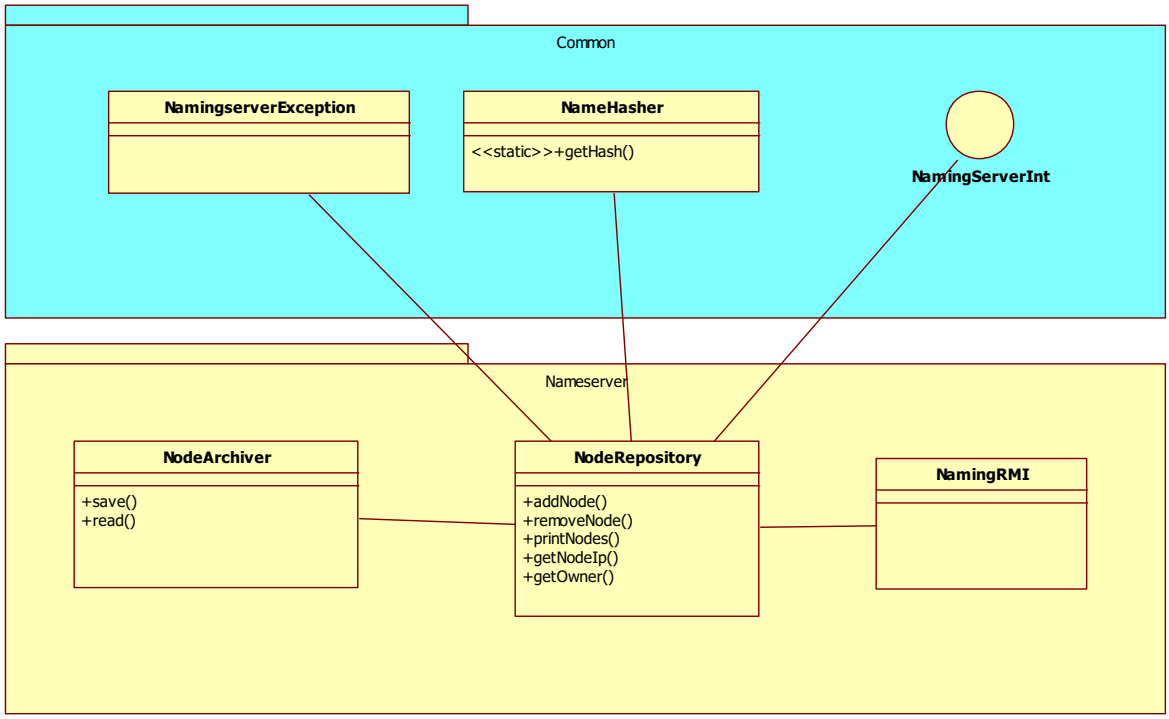
The naming server's task is to keep an overview of all the nodes in the network. It has a supporting role to the nodes and doesn't store any files itself. In our design of the system we assumed the nameserver to always be online and reachable from a node's perspective.

When a change in the node-map occurs, it is also immediately saved to disk in a comma separated list (CSV). This way, the list is maintained even when the namingserver shuts down, or experiences a failure. Then, if we would restart the program, the CSV-file is read and the data map pre-populated.

CSV is a very easy data format to use: it is simply plain text where every column is separated by a comma (sometimes semicolon in the European version, because we use the comma for decimals). A line break announces the next row. It might not be a surprise that csv is a common interchange format for spreadsheet data. It's easily readable (even by humans) and editable which makes debugging easy.

When in memory, the list of nodes is saved in a TreeMap from the Java Collections framework. This collection has some useful properties. It can't contain duplicate keys (hashes from the node name), which is exactly what we want. The keys are sorted in ascending order, which makes searching for the next and previous node when given a hash a breeze.

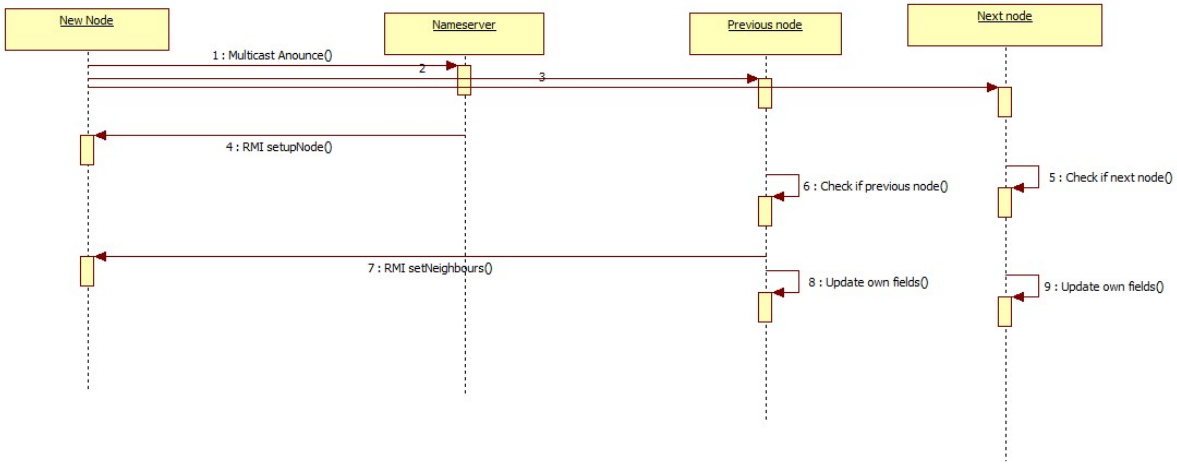
The code is split in two packages: be.dist.common and be.dist.name. The former contains all classes that are needed in the nameserver as well as the nodes. This makes the distinction between the different classes much easier. The common-package also contains a subpackage "exceptions" (not shown) which contains our custom exceptions. These might also come in handy on the node at a later time.



Figuur 2: class diagram

Assignment 2: Discovery & Bootstrap

The discovery and bootstrap process is executed when a new node is started. Its tasks are manifold, lots of things need to be set up. To accomplish this, many classes need to work together, even between different hosts. The function calls are send over the network trough Remote Method Invocation (RMI).



Figuur 3: sequence diagram

The first announcement is sent via multicast. As internet routers do not forward this type of packet, the discovery process will only work on a LAN. Even the combination of multicast and WiFi-connections might pose some unfortunate problems. To avoid these problems, it is recommended to only utilize cabled connections when implementing the system. Once the first announcement is sent, the rest of the communication happens trough RMI or TCP (for the file transfers). The advantage of this is that RMI and TCP are, in general, way more reliable forms of communication in comparison to the previously used multicast.

This is also the reason why we need to specify the IP address while setting up the system. Otherwise the multicast might be send out through the wrong interface (probably to the default gateway).

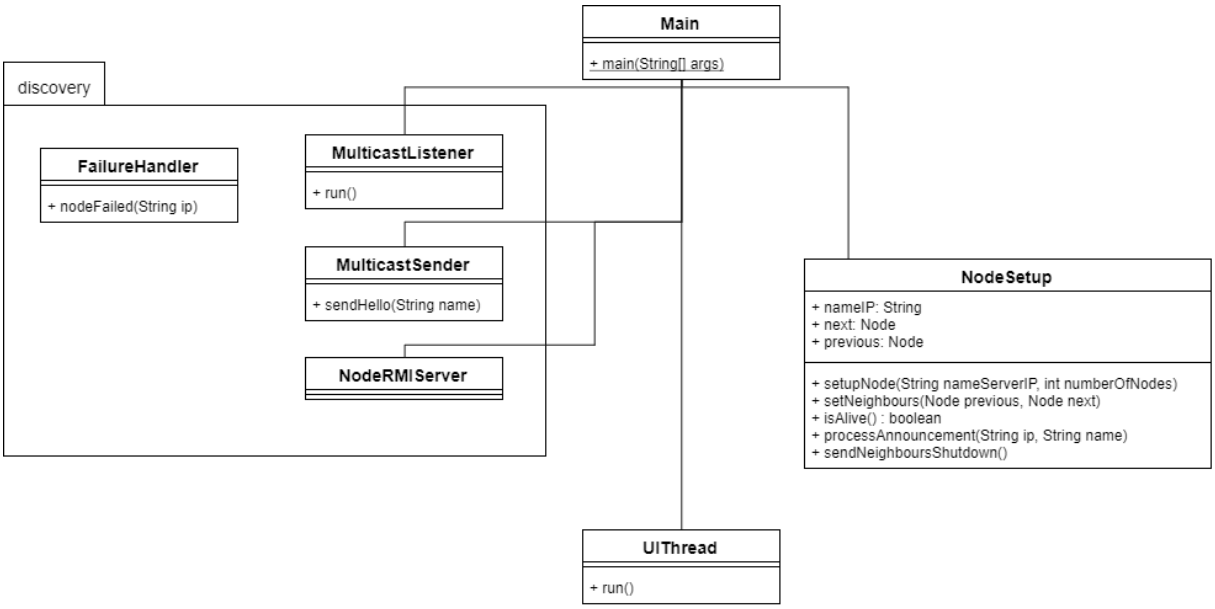
It's quite an important requirement that the multicast announcement arrives without any problems, as there is no built-in way to check its transmission. In a later stadium, we could always add a timeout timer to resend this packet when no setup from the neighbours is received. This however results in the new problem of duplicate announcements.

The following procedure is followed chronologically at startup:

1. A new node sends a multicast announcement on the network
2. The nameserver responds (via RMI) with some configuration information:
    a. It's IP address
    b. The amount of nodes currently on the network
3. The other nodes also listen for the announcement and update their next and previous as necessary

4. The previous node sends his information about the next and previous nodes to the new node, so it knows it's place in the network.
5. The nodes can now start replicating files (new and existing) to the correct locations.

It is important that we implement the different 'listeners' in separate threads. This is because multiple messages may arrive in parallel, and thus, at the same time. On the other hand, we also need to make sure to always use the current values of all received setups because these may change quite often.



Figuur 4: discovery & bootstrap class diagram

Assignment 3: Replication

The purpose of System Y is to store files in a distributed way, so replication definitely is a core part of the functionality of the system. But this functionality brings, with it, a series of new challenges to be overcome. Consider a situation where a new node joins an already existing system. Where then should the files of this new node be duplicated? And how should they be transferred?

Each file should also have some metadata associated with it. This data is always stored at the "owner" node and should also be transferred if ownership changes.

Another thing to consider here is that we would like the files to retain their name and extension when transferred over to another node in the system. To implement this, we simply add some bytes at the beginning of every TCP stream allocated for this exact purpose. As the maximum file name length in most common platforms (Windows with NTFS, MacOS wit HFS+ and Linux with ext4) is 255 characters, this seems like an acceptable choice.

IP & TCP header	255 bytes	[0 - …] bytes
	Filename + extension	File data

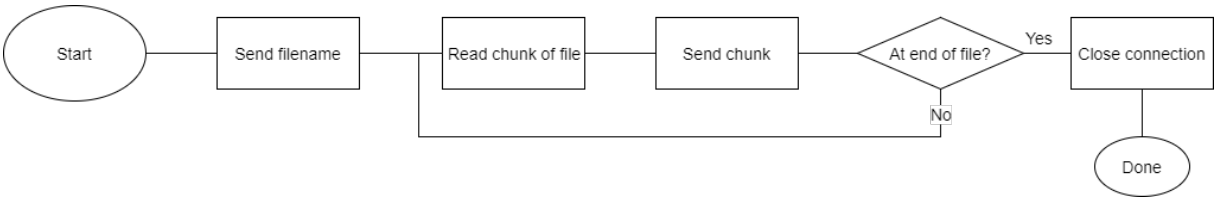
Most of the bytes will remain unused (filled with null characters), but this is a small overhead to pay for the added simplicity. Especially because this happens only once per file transfer. We could mitigate this problem but this would greatly increase the complexity.

Our solution has the big advantage that the filename (and extension) are always kept together with the data. Because multiple files may be “in flight” at a time, this makes it easy to save the file in a correct way at the receiving end. Though, if we ever want to add more information to the file, then this would be better sent via RMI and coupled to the filename.

While testing this part, some files got corrupted. After examining them with a hex editor, it was clear that there was an unstripped null character at the beginning of every file.

While transferring files we should take care to do the process in chunks. We read a piece of the file and send it on its way. Then we do this for the following parts, until the full file is send. If necessary, the IP stack will provide fragmentation to fit the packets on the datalink.

The flowchart illustrates this process:



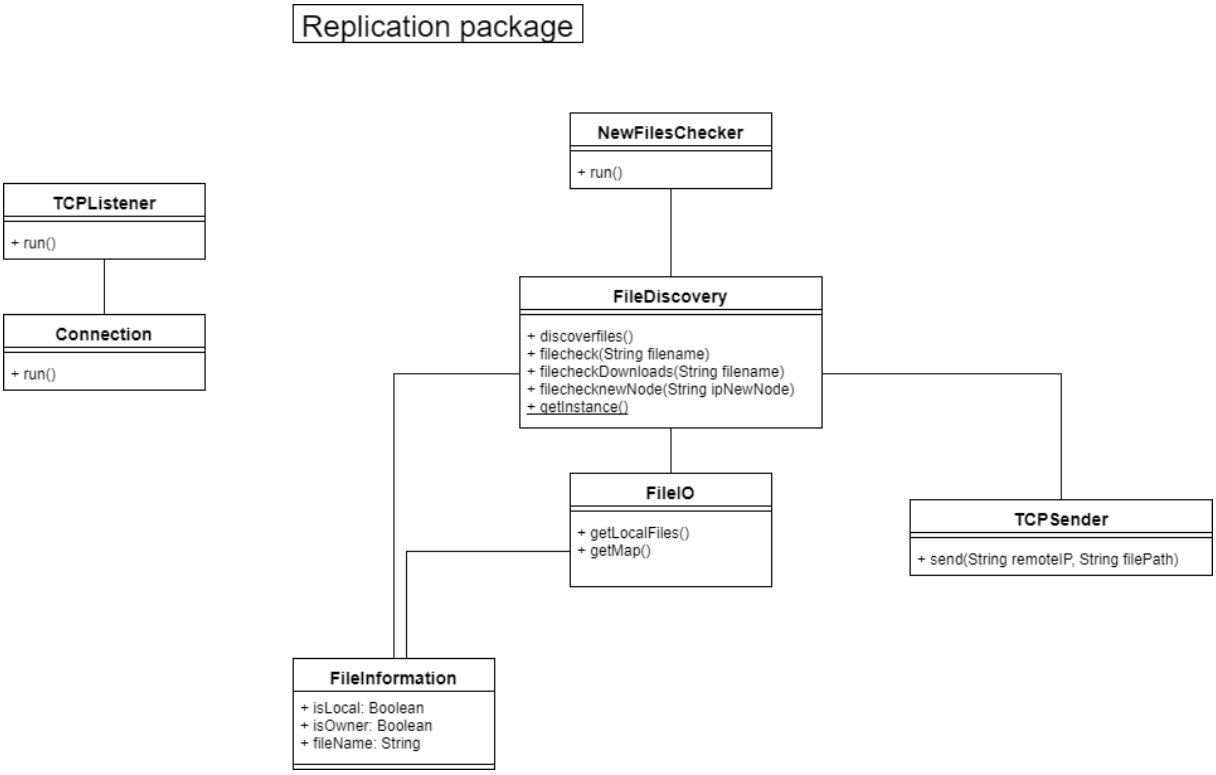
TCP automatically splits the file in appropriately sized packets and also provides protection against possible transmission failures.

The receiver needs to split the data and metadata parts and remove the null characters of the filename. It is important that this happens in a multithreaded manner, because multiple transfers may be in progress at a given moment.

Each file also has some metadata associated with it, which contains the log information. This information is paramount for the correct functioning of the shutdown and failure procedures. We never need to store this on disk though because the information is always saved on an active node and is transferred at shutdown. When the last node shuts itself down, the last files are removed from the system as well. So there clearly is no longer a need to store any metadata.

When new files are added to the folder of the local node, these files should also be replicated. To achieve this, we should check the folder periodically and compare the contents with those of the previous check. This is the easiest way to discern which files are new. From there on out, the actions we take are exactly the same as when we started up the system. All of this behaviour is implemented in the “NewFilesChecker”-class. This class checks for new files with a fixed interval of 5 seconds.

Another important aspect to take into consideration is concurrency. We should always prevent concurrency problems (race conditions, …) while implementing. To do this we employ the “synchronized” keyword on strategic methods. On the other hand, we should always keep in mind that overdoing this may result in other threads having to wait undesiredly long. This on itself, could cause a tremendous slowdown of our system in general.



Figuur 5: replication package class diagram

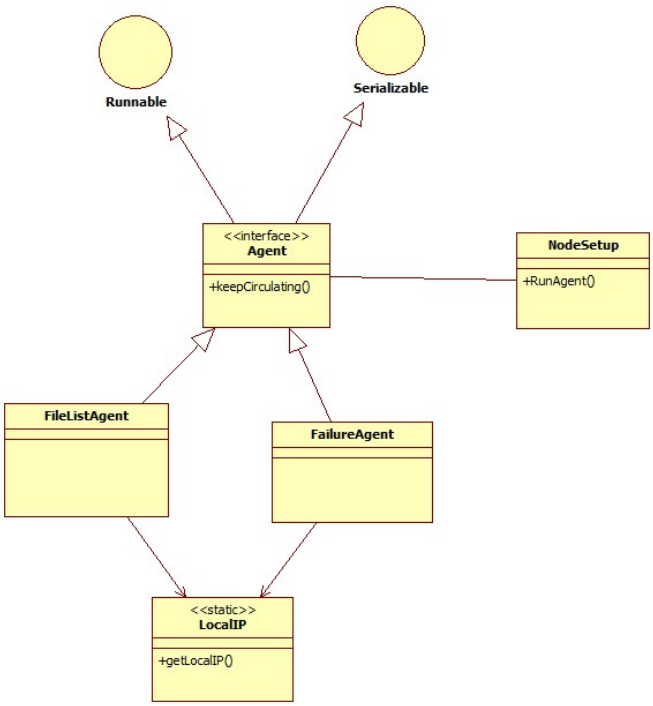
Assignment 4: Agents

An agent is defined as a piece of code that is passed around in the network. An important difference to note here is that, instead of calling methods trough RMI to execute some code, the code itself is passed around. This can also happen via RMI and the Runnable-interface.

Object Oriented Programming lends itself very well to this paradigm, as the objects contain data as well as code. To enable this functionality our class should be Serializable. And using a common interface, the nodes don't even have 'a need to know' about the exact implementation of this feature. For this purpose, Java has the Runnable interface.

We should take to keep the agents circulating when necessary, even when the amount of nodes in the network changes.

This infrastructure was designed in such a way, that it can be used for multiple agents. By using a common interface, the nodes don't even need to know about the exact classes that will be used. This makes our system easily extendable.



The class diagram above sums everything up quite nicely. As you can see here, every class is associated with about two other classes, which keeps the system loosely coupled.

Our system relies on two agents:

- **FileListAgent:** this agent makes a list of all files stored in the whole system. This enables us to create a GUI that lists them all. This agent also keeps the locks that are set. It also keeps the owner of each file, so that we can locate it without having to bother the nameserver.
- **FailureAgent:** this agent is started when a node fails. It searches his files on the other nodes and replicates them again, to restore the redundancy.

Assignment 5: GUI & file download

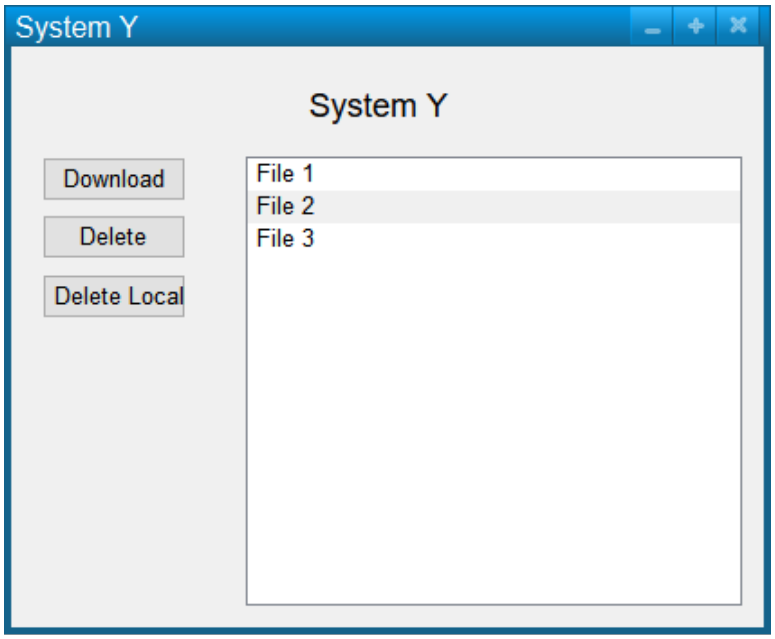
The Graphical User Interface (GUI) might be less important for the functioning of the system, but is fairly critical when it comes to usability nonetheless. It allows the users to download files from the system to nodes that are neither the owner nor the source. In a way, this upgrades our system to classify itself as a file sharing system.

To create this GUI, we used the very popular Java Swing-framework. To ease the design of this GUI and keep our codebase clean, we use the design-tool of our IDE. In our case this is JetBrains' IntelliJ IDEA.

The interface includes a few basic elements:

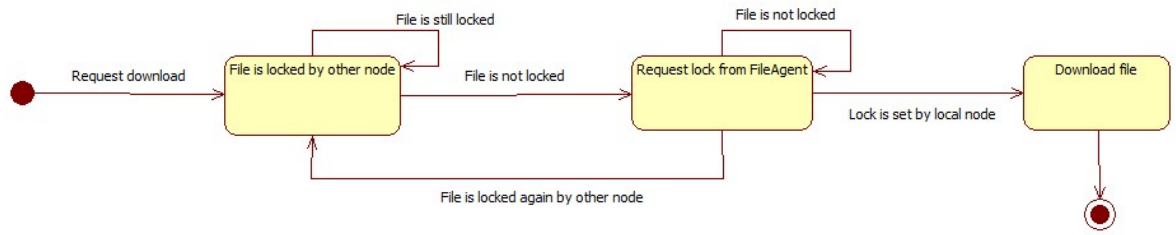
- List of files
- Download button
- Delete buttons
    - For the local file
    - For the whole network

Before we started building the GUI, we made a sketch of our design:



The design contains three buttons that act on the file currently selected in the list next to them. This list is automatically updated by the FileListAgent. Not that a selection of multiple files is not supported, although, in theory, it could be done with some (rather major) tweaks in our code.

The MVC pattern is used by the Swing framework to create a working GUI. While downloading a file, we go through a few states before we can start the transfer. This is shown in the state diagram below.



An improvement to this would be to keep the lock as long as the download is going on (especially for big files). However, this requires us to coordinate between the receiver, FileAgent and GUI at the same time. This makes adding this feature quite difficult. Instead, we assumed there is no need to lock a file after the download has started and the logfiles are updated.

Running System Y

At this point you probably want to try this amazing system for yourself. Instructions on how to do this are included with the code trough the Github-repository. They should get you going in less than 10 minutes. Open the file README.md to get started.

>> https://github.com/iliessens/SystemY

Sources and references

These sources were used to create the system and to write this report.

- Comparison of file systems, https://en.wikipedia.org/wiki/Comparison_of_file_systems#Limits
- Warmer, Jos, and Anneke Kleppe. *Praktisch UML*. Pearson Benelux, 2015.
- Can an interface extend multiple interfaces in Java?, Stack Overflow https://stackoverflow.com/questions/19546357/can-an-interface-extend-multiple-interfaces-in-java
- Designing GUI. Major Steps, Jetbrains manual, https://www.jetbrains.com/help/idea/designing-gui-major-steps.html