System Y: portfolio

Jannes Van Goeye

Robbe Pauwels

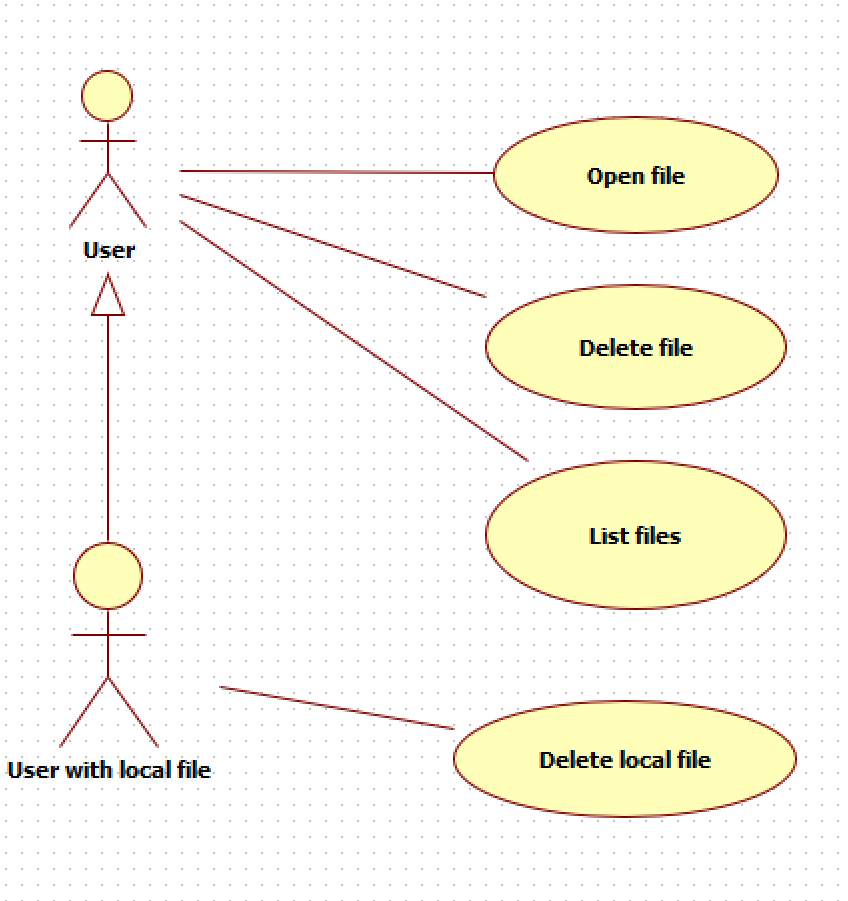
Imre Liessens

# 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. User’s local files are automatically sent to at least one other nodes in de network. Every user can access the files that are stored anywhere in the network, in an easy and transparent (the user doesn’t need to know the inner workings) way.

Minimal configuration should be necessary for new nodes, only a selection of the network interface to use (via it’s associated IP) is necessary.



# 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. In our design of the system we assume the nameserver is always on and reachable.

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

CSV is a very easy data format: it is plain text with every column 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 human-readable 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.

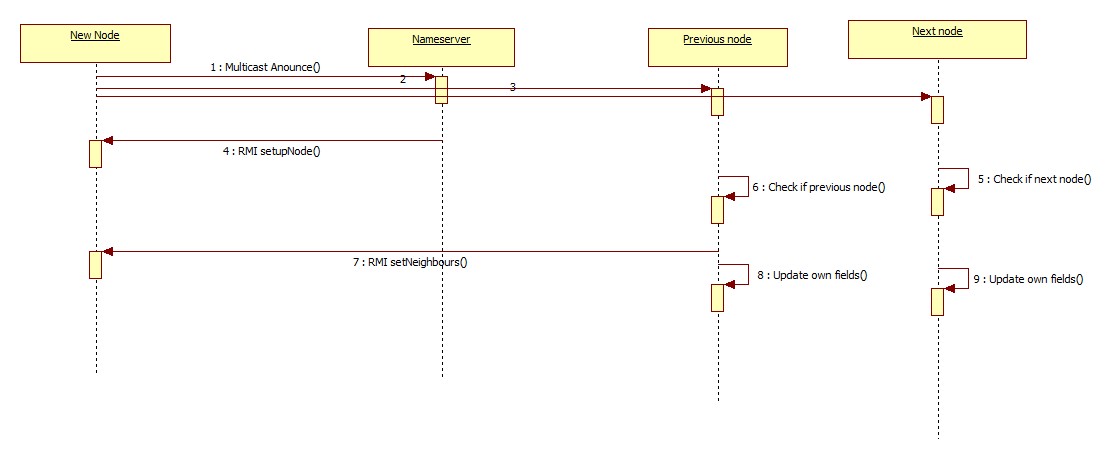
The code is split in two packages: be.dist.common and be.dist.name. The first package contains all classes that are needed in the nameserver as well as the nodes. This makes the distinction between the different classes easier.

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.



# 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).



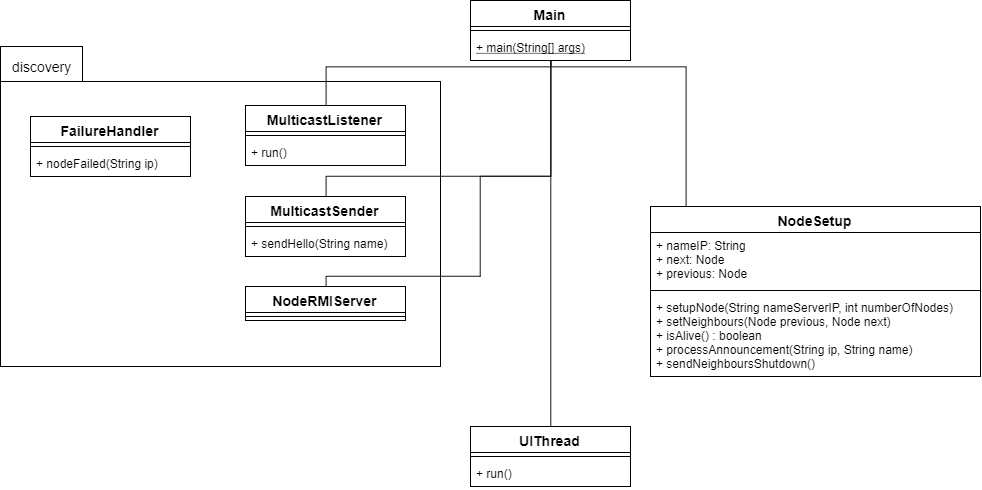
The first announcement is sent via multicast. As internet routers do not forward this type of packet, the discovery process will only work in a LAN. Even the combination of multicast and WiFi-connections might pose problems. So I’s best to only utilize cabled connections. Once the first announcement is sent, the rest of the communication happens trough RMI or TCP (for the file transfers). This has the advantage of a more robust and reliable communication.

It’s quite important that the multicast announcement arrives without problems, as there is no built-in way to check it’s 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 yields the new problem of duplicate announcements.

The following procedure is followed at startup:

1. Node sends a multicast announcement on the network
2. The nameserver responds (via RMI) with some information:
   1. It’s IP address
   2. The amount of nodes currently in the network
3. The other nodes listen for the announcement and update their next and previous as necessary
4. The previous node sends he information about the next and previous to the new node, so it knows it’s place in the network.
5. The nodes can now start replicating the file (new and existing) to the correct locations.

It is important that we implement the different receivers in separate threads, because multiple messages may arrive at the same time. We also need to take care to use the current value of all received setup, because this may change quite often.



# Assignment 3: Replication

The purpose of System Y is to store files in a distributed way, so the replication is a core part of the functionality. When a node joins, where should the files be stored? How should they be transferred?

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

We should also take care to transfer the file name and extension, so that the replicated files can have the correct name. The solution for this is to have some bytes at the beginning of the TCP stream allocated for this 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 a good choice.

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

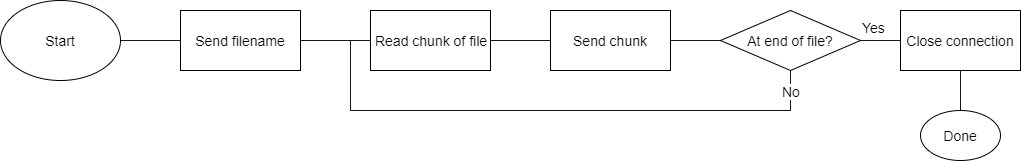
Most of the bytes will remain unused (filled with null characters), but this is a small overhead to pay for the added simplicity. Especially because this happens only once per file transfer. We could mitigate this problem but this would greatly increase the complexity.

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. Should 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:



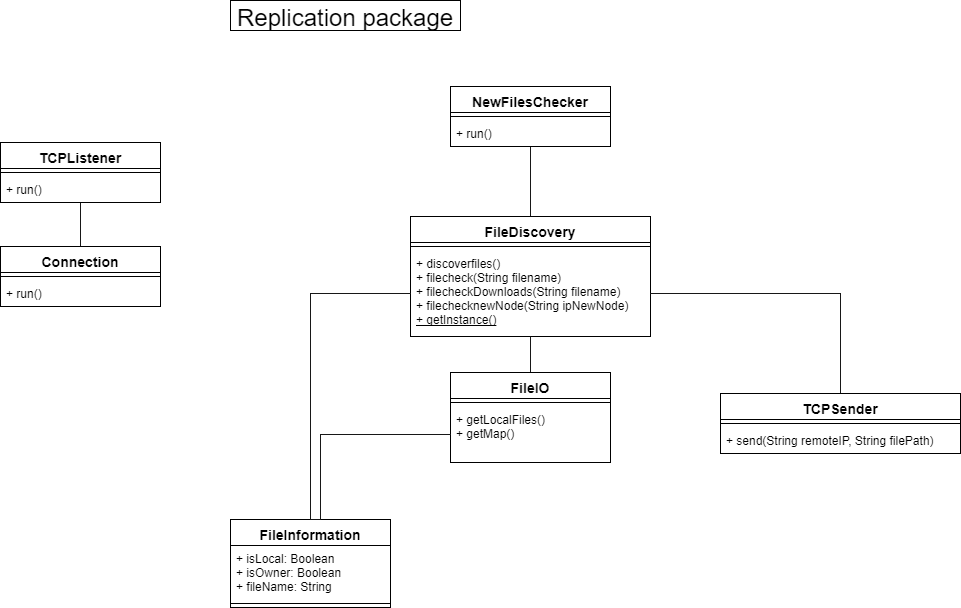
TCP automatically provides the splitting of the file in appropriately sized packets and also protection against failures.

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, which contains the log. This information is necessary for the correct functioning of the shutdown and failure procedures. We don’t need to store this on disk though. The information is always saved on an active node and is transferred at shutdown. When the last node shuts down, the last files are also removed from the system. So there is no longer a need to store any metadata.

When new files are added to the folder of the local node these 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. The actions we take from then are exactly the same as when we start the system. This behavior is implemented in the “NewFilesChecker”-class. This check for new files with a fixed interval of 5 seconds.

We should take care to prevent concureency problems (race conditions, …) while implementing. To prevent this we emloy the “synchronized” keyword on strategic methods. We shouldn’t overdo this, though, as the other threads may need to wait. This could slow our system down.



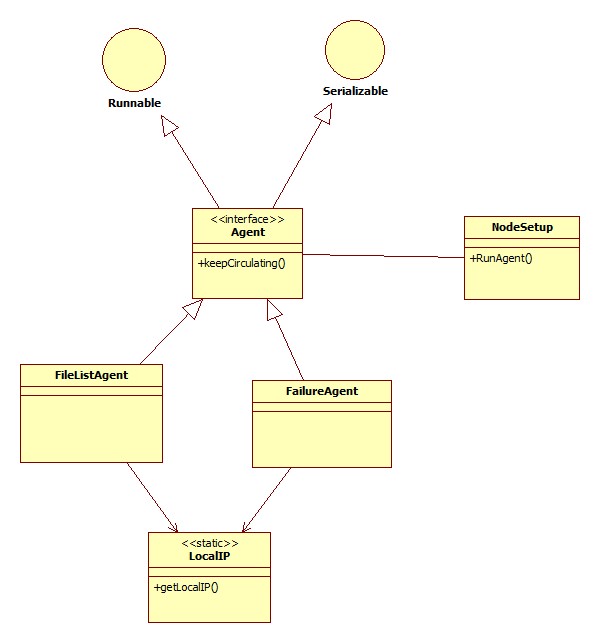
# Assignment 4: Agents

An agent is a piece of code that is passed around in the network. 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 need to know the exact implementation. For this purpose, Java has the Runnable interface.

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

We have designed this infrastructure in such a way, that it may be used for multiple agents. By using a common interface, the nodes don’t even need knowledge of the exact classes that will be used. This makes our system easy to extend.



The class diagram sums it up quite nicely. As you can see every class is associated with at most two other classes, which keeps the system loosely coupled.

# Sources and references

* 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>