**To run the program:**

**Initial run:**

* Run *“pyro4-ns”* to start naming server
* Run *“py server.py”* to start one server (please make sure it says Server0 running)
* Run *“py server.py”* to start second server (please make sure it says Server1 running, otherwise restart the pyro4-ns and repeat the process)
* Run *“py server.py”* to start third server (please make sure it says Server2 running)
* Run *“py frontend.py”* to start front end
* Run *“py client.py”* to start client

**To kill a server:** (in my implementation, it’s the equivalent to setting the server “offline” or “overloaded”)

* just close the cmd window running that server

In my implementation, every time a request is made, the pyro4 naming server is checked to see if any changes have been made. So, **when a server is killed, it is removed from pyro4 naming server** and from the dictionary of servers.

**To bring a server back on:**

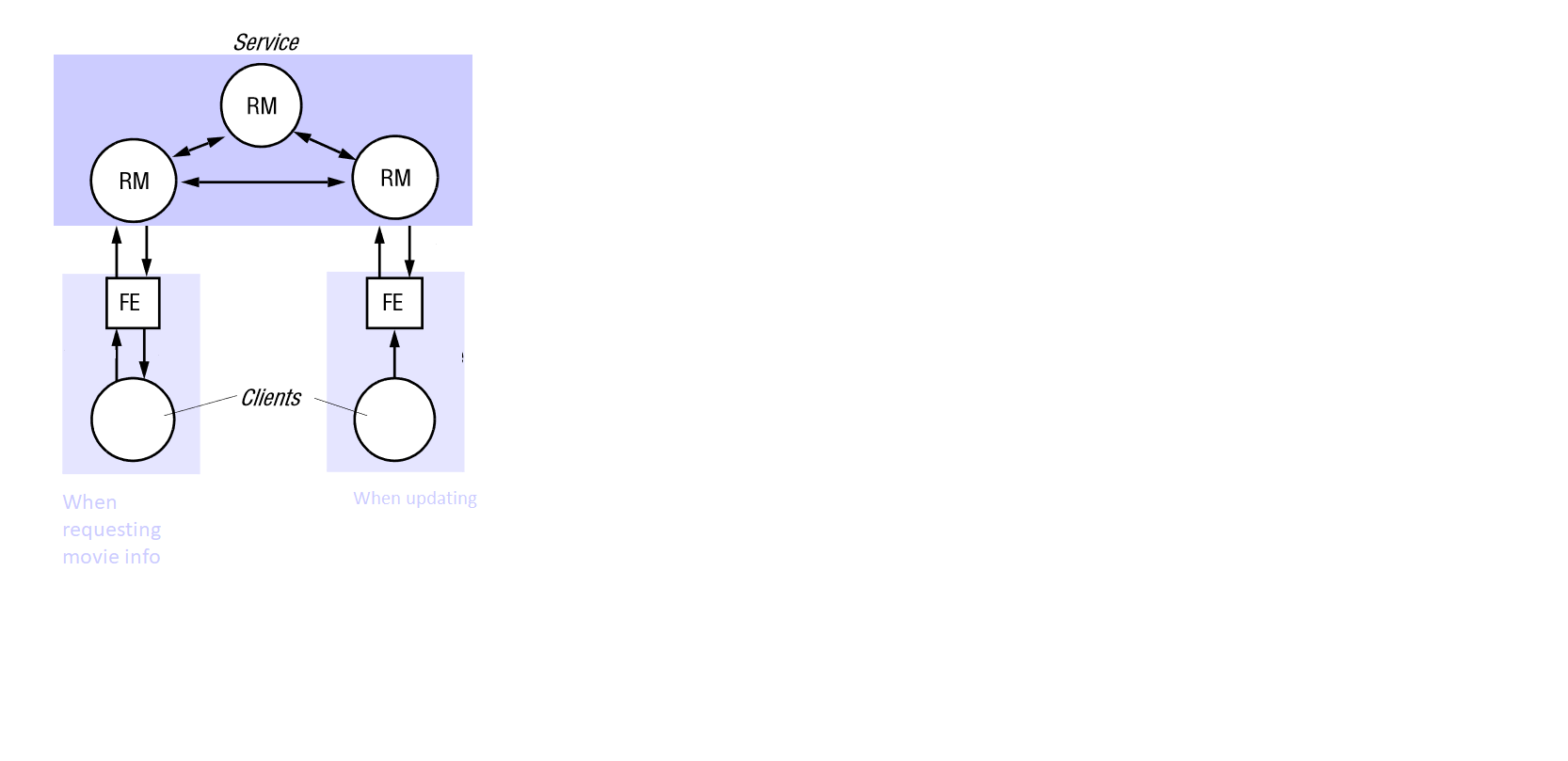
* run server.py

**N.B.** *it's* ***dynamically*** *set to take the name that is missing in the naming server and brings it back up*

e.g. when killing Server1 and having Server0 and Server2 running, when running “server.py” again Server1 will be brought back up.

Each server links to a different .csv file called “*ratings0.csv*”, “*ratings1.csv”,* “*ratings2.csv”* in respective order of server index running. For testing purposes, **these files are empty** just with the column headings set for you to observe the robustness of the implementation.

**System Architecture**



Client input fields

Client input fields,

FE timestamp,

Update ID,

Server index

Updated FE timestamp,

acknowledgement

**Gossip**

Executed Host Log,

Server index

Client input (movieID)

Average(movieRating)

MovieID,

FE timestamp

•The update request takes a bit **slow** to process, because my implementation checks every time the NS server and if it detects that a server that has been switched off from command line or another server has been added, it will make the necessary amendments to the dictionary of available servers.

•The server is chosen randomly for each request from the dictionary of available servers.

• The status of the server is allocated randomly every time server.getStatus() is called with a **10%** probability of being offline and **10%** probability of being over-loaded.

• The **gossiping** works in the following way:

Once the server has collected several updates (in my implementation =2) it will send gossip messages to all the servers available.

A gossip message contains *server index* and *executed log* of the server sharing data.

• When a request is processed, if the FE timestamp is equal to the host server timestamp, it will proceed with making the necessary changes; otherwise it will request updates forcedly from the available servers. If no servers are found as “available”, the server will just set its timestamp equal to the server timestamp. This is when data loss occurs.

If you want **to test the robustness** of the implementation, I would recommend testing with shutting down the server from command line (by closing the window). And then bringing it back up (see instructions on page1).