You are asked to develop a replicator (client) that distributes a large job over a number of computers (a server group) on a single switched LAN (our Linux lab). In this assignment, a large (simulation) job can be divided into a number of small jobs, each of which can be assigned to one machine from the server group for execution. The execution results of the small jobs can be merged once all of them successfully terminate.

**System Architecture:**

```
client      server1    server2    server3 ...
  |            |          |          |
  |            |          |          |
  |    LAN     |          |          |
  |------------|----------|----------|-----
```

The client and servers are running Network File System (NFS) so that user files are visible at $HOME directory. You may want to set up the following environment:

- $HOME/replicate.hosts: a list of (server) hostnames which participate in the simulation. There is no reason why your implementation cannot support up to 10 servers.
- $HOME/replicate_out: the directory that stores the small job execution result.

The simulation program " hyper_link " (binary) is provided. In this assignment, you don't need to know or care what "hyper_link" does, and actually it is a computing intensive (CPU demanding) simulator. The command line arguments of "hyper_link" are job# 1000000 999 1 2 2 100, where the job number determines the number of small jobs in your simulation. To allow the client to run a large job, the job# should be given in a tuple: start, end, and step. For example, the command (from the client) "hyper_link 1 100 1 1000000 999 1 2 2 100" yields 100 small jobs with the job# starting from 1 to 100. Each small job produces a screen output (see example below) at the end (if finished successfully). Your code needs to redirect the output to a file and save it in $HOME/replicate_out. For example (on the server side),

```
./hyper_link 1 1000000 999 1 2 2 100
```

will produce a screen output looks like (it takes approximately 2 minutes on spirit):

```
1:1000000:999:2:0.5:1.125193e+00:2.454346e-04:6.251640e-01:2.205078e-04:0.000000e+00:0.000000e+00
```

**Requirements:**

1. The communications between the replicator and servers are achieved through remote procedure calls in the client-server fashion.
2. A user interface is required for the replicator to control the server. A command line interface will be acceptable. **A (working) graphic user interface (GUI) will impress the instructor and earn up to 20 bonus credits.** Your client interface should at least support the following operations.
   - start a large job. For example: hyper_link 1 100 1 1000000 999 1 2 2 100 (start 100 small jobs with job number starting from 1 to 100)
   - show the current CPU load of a certain server (if the server is active).
   - show the current server status (active or inactive).
   - stop a certain server.
   - restart a certain server.
   - For those who are going to implement GUI, you need to create an icon for each server, and show the server status in the real time, e.g., the CPU load (with the mark of hi-threshold), active/inactive, etc.
   - The hi-threshold and lo-threshold can be set to the pre-determined values (as long as they are reasonable). Alternatively, you will impress the instructor by implementing the configurable threshold values during the run. If that is the case, you have to provide two extra commands that set the values.
3. The replicator has to make sure all small jobs are successfully finished.
   - If a server crashes (or not responsive), the running job (not finished yet) will be killed and rescheduled (at a certain time per your design) for execution.
   - If a server CPU load exceed the preset threshold (the higher threshold), the replicator stops the server (and therefore kills the job).
   - The replicate should keep polling the CPU load of the stopped server. Once the load becomes lower than the lower threshold (a preset value), the server should be reactivated to run the jobs.
   - The replicator can also stop any server (through user interface) if needed. Once happened, the unfinished job will be killed.
   - If a job terminates abnormally (e.g., being killed), the replicator has to reschedule the job execution later.
4. Makefile: you need to provide a Makefile that allows the instructor to compile your code by simply typing "make".
5. Write-up: you are required to write a README document (in txt format) that describes your project design detail and the execution sequence (with the commands). In particular, please explicitly state which part, if there is any, does not work and the possible reasons why that module does not work. For those working modules, please give a brief (in short) sample output.

**Hints:**

1. RPC programming: a brief (Sun) RPC programming introduction is given in the class.
2. CPU load: please check /proc/loadavg for the CPU load information in Linux.
3. Linux signal: the signal mechnism must be used to control the simulation execution at the servers.