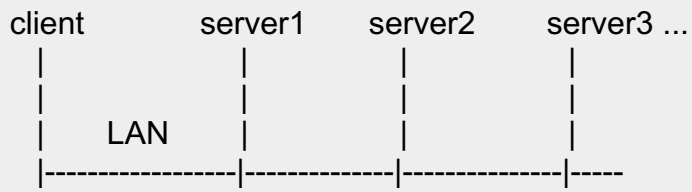
A Simple RPC-based Distributed Computing Platform

Description:

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:



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 5 servers.
- $HOME/replicate_out: the directory that stores the small job execution result.

The simulation program "hyper_link" (binary) is provided and should always be executed on the servers (the client computer should never execute this simulator binary). 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 operations at the client (replicator):

1. Your client program should enter a replicator console (shell) which is ready for user's commands;

2. 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). The server needs to redirect the (screen) output to the output file "replicate_out" (on the server side) since the client cannot see the server's screen.

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

The operations at each server:

All the servers should be running before a user enters commands on the client side. The server should enter an infinite loop and wait for the command from the client. Upon the reception of a command, the server should fork a child process which execute the command (with arguemtns) provided by the client. For example, if the first string of the command is "hyper_link", then the server will create a child process which subsequently executes the hyper_link binary, with the provided arguments from the client. The main process of the server should immediately wait (instead of wait for the termination of the simulation) for the next request from the client.

Guidance (where do I start?):

A good starting point is to replicate the example given in the SUN RPC slides. That is, in addition to hyper_link, your client also support commands like "squaring", or anything else, such as "multiplication", "exponentiation", and so on. In that case, start with a single server in your "replicate.hosts" file. You can reuse the code provided in SUN RPC slides to implement your "squaring" command. Note, to support the "squaring" command on the server side, you are required to write a squaring binary exectuable that performs a squaring operation on the client provided value. Similarly, if you wants to implement other operations (commands), the corresponding binary should be created on the server side.

Question: My question is that we can't write the executable file, right? it would be generated while we are compiling the code .To support the squaring operation ,do we need to right the executable file?

Answer: It really means that the squaring operation is performed at the server side. The binary described is part of the server code, which is a function that performs the operation.
Question: I understood stopping a job and rescheduling a job but, What it really means to stop a server? If I stop a server execution by a command from the client, how should I restart it from client because there won't be any communication between the client and server right?
Answer: When you stop a server, the job that is running on that server is killed, but the server is still running the RPC server process (means ready to take the next job). The reason to stop a server is because the server's CPU load is too high. When the server is back to idle, the server should be automatically utilized by the client to complete the remaining jobs (if there are).
NOTE: in the SUN RPC slides, the remote printmsg example requires the server function printmessage_1_svc() open "/dev/console", which is mostly not writtable by a regular user. A simple remedy is to skip this part and directly write to the screen as below:

```
int *printmessage_1_svc(char **msg, struct svc_req *rqstp) {
  static int result;

  fprintf(stdout, "%s\n", *msg);
  result = 1;
  return(&result);
}
```

Requirements:

1. The communications between the replicator (client) and servers are achieved through remote procedure calls in the client-server fashion. **You can only use C programming lanuage to complete this project. Your implementation should not rely on any extra library (to compile your code). Your code must compile by using gcc installed on the Linux workstations in FH 133.**
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.

Grading Criteria:

1. (30 points) A successful implementation of a base RPC client-server platform. Basic functions such as "squaring" or "multiplication" can successfully run between the client and the server. Note, the commands should always run on the server side (never on the client side)!

2. (60 points) Successful implementation of the RPC based replicator distributed computing system with up to 5 servers.

a) (30 points) a base replicator system with at least 2 servers, and the hyper_link simulation can successfuly complete, even if CPU load, server stop/restart, threshold, are not supported;

b) (10 points) support of the server CPU load information;

c) (8 points) support of the server stop/restart function;

d) (8 points) support of the CPU load threshold;

e) (4 points) support up to 5 servers.

3. (10 points) Good documentations: including a makefile and a README file.

4. (20 bonus points) GUI design as described above.

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.

Submission:

1. Create a folder and name it as your group number, concatenated with "_p2", e.g., group1_p2 (all lower case).
2. Copy all your source code to the above folder (clean your source code folder and remove all binary files). Please do not submit any binary code! Only the client, server source code as well as *.x file are required.
3. Provide a Makefile such that the instructor only needs to type "make" to generate the RPC client stub, the RPC server stub, and compile all the client and server binaries.
4. Edit a README file (in plain text format only) and provide the following information:
    - Group member names
    - Member contribution distribution (e.g., 50%-50%)
    - Design details
    - Compiling instruction and execution sequence (with commands)
    - A sample test run
    - Please explicitly state which part, if there is any, does not work and the possible reasons why that module does not work.
5. Log in grail, go to the parent director of the folder you created, and run (suppose the your folder is "group1_p2")
   $ turnin -c cis620w -p p2 group1_p2
   If there is no error message, your submission is successful.
6. Your most recent submission will always automatically overwrite the previous one