# Programming Assignment 1

Ying Lu (lu000097@umn.edu), Ke Wang (wan00802@umn.edu)

## Design Document

In general, for this programming assignment, we implemented a simple RPC compute framework for implementing image-processing. The framework receives jobs from a client, splits a job into multiple tasks, and assigns tasks to compute nodes which perform the computations. The client will send a job for image processing. The framework uses two different scheduling policies which are Random and Load-balancing to assign tasks to compute nodes.

- **Shared files**
  The client, server, and compute nodes will all have access to the following files:
  - machine.txt: contains address information for server/client/nodes
  - config.txt: contains policy information and load probability of each node

- **Thrift IDL files**
  - imageProcessing.thrift:
    - defines a RPC service for the client and the server
    - Service name is ImageProcessing
    - There is one function inside the service:
      `double processImages(1:string folderName)`

  - computeNode.thrift:
    - defines a RPC service for the server and compute nodes
    - Service name is computeNode
    - There is one function inside the service:
      `double singleImageProcess(1:string fileName)`

- **Client**
  The client sends a job to the server which is the folderName which stores data to be analyzed. When the job is done, the client will get the result of the total elapsed time for the job it sends.First the client will process the machine.txt to get the server's address . Then it connects to the server and sends a request to the server via a RPC call using that address.

- **Server**
  The server receives the job sent by the client. Inside the main function, we set the hostname of the server to be "0.0.0.0" to allow for remote connections. The handler is `ImageProcessingHandler` which contains the implementation of `processImages(folderName)`. It does the following steps:
  (1) Firstly, after receiving the folder name, the main thread splits the job into multiple tasks which correspond to the `def split(folderName)` function in the

server.py. The split function will process the folder, extract all images into a list and return the image list. After this step, we can see each image as a task. We used a `queue` to save all unfinished/rejected tasks.

(2) Secondly, the server assigns each task to a compute node. It creates 4 threads in total so that each thread will pick up a task from the `tasksQueue`. By doing this, each task can run in parallel which saves completion time.

Before running the thread, the server will process the config.txt to get the current scheduling policy and process machine.txt to get all nodes' addresses. Then the server will choose the compute node based on the policy. After deciding on the compute node, the thread will pop out a task from the `tasksQueue`, which then connects to the chosen compute node via RPC call by passing the task(image filename). If the task is rejected under load-balance policy, it will push back rejected tasks into `tasksQueue`. Here is the detailed implementation of each scheduling policy and its corresponding method in the server.py:

- **Random -** `def randomSelectNode()`
  The server will assign tasks to any compute nodes randomly chosen. Delays are injected according to the load probability assigned to each node when executing the image processing in the computeNode.py.

- **Load-Balance** - `def BalanceSelectNode()`
  The server keeps a `nodesTasks` dictionary which maps the node name to its task completion status `[sentTasks, rejectedTasks]`. For example, it will look like this:

```
{
   "node_0": [5,3],
   "node_1": [3,0],
   "node_2": [6,2],
   "node_3": [1,1]
}
```

  If there are nodes receiving no tasks, the server will randomly choose one of those nodes to assign a task to it. If all nodes have at least one sentTasks, the server then chooses the compute node based on the rejection ratio of each node. In order to avoid all threads sending tasks to the node with the lowest rejection ratio, we first find the node with the largest rejection ratio and then randomly select one from the remaining nodes to spread the load appropriately.

(3) After all threads finish completing tasks (`tasksQueue.empty() == True`), the server will print the elapsed time of running all tasks and returns the elapsed time back to the client.

- **Compute Nodes**
  Each compute node runs a multi-threaded Thrift Server (`TThreadedServer`) to accept and execute multiple tasks. Compute nodes will process the config.txt to extract the corresponding load probability and policy. We use random numbers to simulate load probability for rejecting tasks and injecting loads:

  ```
  num = random.randint(1, 10)
  if (num >= 1 and num <= loadProb*10):...
  ```

  Then each node is maintained to handle the image process. The image process is maintained by the function `singleImageProcess(fileName)`.

  Inside the main function, we set the hostname of the node to be “0.0.0.0” to allow for remote connections. Inside the `Class ComputeNodeHandler()` we have the defined function `def singleImageProcess()` and the following helper functions:

  1. `def processConfig()`: Initialize the policy for the node computation, we can choose, a. “random” and b. “load-balance”. The function will get specification from the `` `config.txt` ``. Where we can change the policy and the probability for each node manually.
  2. `def checkReject()`: We used the random number to simulate probability for rejecting tasks (if ***`loadProb`*** is in the random range), Which is specified from the `` `config.txt` ``. If rejected, then the compute node will return immediately.
  3. `def InjectDelay()`: We used the random to simulate and to check if the load is injected or not. Sleeping 3 seconds before executing if the load is injected.
  4. `def imageProcessTime()`
     Inside the `def ImageprocessTime()`, we use functions start and end to catch each time processing time, and then `cv2.cvtColor()`, `cv2.GaussianBlur()` and `cv2.Canny()`. The `cv2.cvtColor()` method is used to convert an image from one color space to another. So that we can convert to grayscale for the next stage. It is used to block the noise of the picture and get better detection results. The `cv2.gaussianblur()` function applies Gaussian Smoothing on the input source image. So that we can blur the image for better edge detection. All this prep works for the better solution provided for the edge detection function: `cv2.Canny()`. After detection we can store the image result back to the directory ‘`output_dir`’. And the function will return the catched time back.

  After the image processing is done, it will save processed images in the data/output_dir and return the elapsed time to the server.

## **User Document** - How to run the service

Here are the detailed steps of how to run each component and how to use the service. All the commands should be run in the terminal.

**STEP 1: Setting the config.txt and machine.txt**

Before running the service, you should modify the config.txt and machine.txt based on your choice. Two policies are {`load-balance`,`random`} config.txt should look like below:

```
policy : load-balance
node_0: 0.1
node_1: 0.5
node_2: 0.2
node_3: 0.9
```

- **Run in the Localhost**

  If you want to run the service on the localhost, first navigate to the proj_dir and change the machine.txt to the following:

  ```
  node_0 127.0.0.1
  node_1 127.0.0.1
  node_2 127.0.0.1
  node_3 127.0.0.1
  server 127.0.0.1
  client 127.0.0.1
  ```

- **Running Remotely**

  If you want to run the service on the localhost, first navigate to the project folder and change the machine.txt to the following:

  ```
  node_0 kh4250-08.cselabs.umn.edu
  node_1 kh4250-03.cselabs.umn.edu
  node_2 kh4250-06.cselabs.umn.edu
  node_3 kh4250-02.cselabs.umn.edu
  server kh4250-05.cselabs.umn.edu
  client kh4250-01.cselabs.umn.edu
  ```

**STEP 2: Running compute nodes, server and client**

After setting the config.txt and machine.txt, you are able to run the service through one of the following methods:

(1) Run the following command in a separate terminal by order.

Start node_0: `python3 computeNode.py 0`
Start node_1: `python3 computeNode.py 1`
Start node_2: `python3 computeNode.py 2`
Start node_3: `python3 computeNode.py 3`
Start the server: `python3 server.py`
Start the client: `python3 client.py`

(2) Using grading.sh to run via ssh

After setting the config.txt and machine.txt, you can use the following script to run the service. Open the terminal and navigate to the project folder where you should be able to see a grading.sh file which will automate the running process. You should set the ssh before running the script. Then you can run the following command:

```
source grading.sh
```

You should be able to see images are processed and saved in the data/output_dir in the project folder. You can modify input images by adding more images in the data/input_dir folder.

**Here are the Assumptions for the service:**

- There will be a single job at a time to simplify the system implementation.
- Client, Server, and Compute Nodes are sharing the same directory.
- The job sent by the client can be the pathname for the input_dir which contains the images for which the edge detection filter needs to be applied.
- All communications in this project are synchronous.
- There will be no faulty nodes during a job execution.

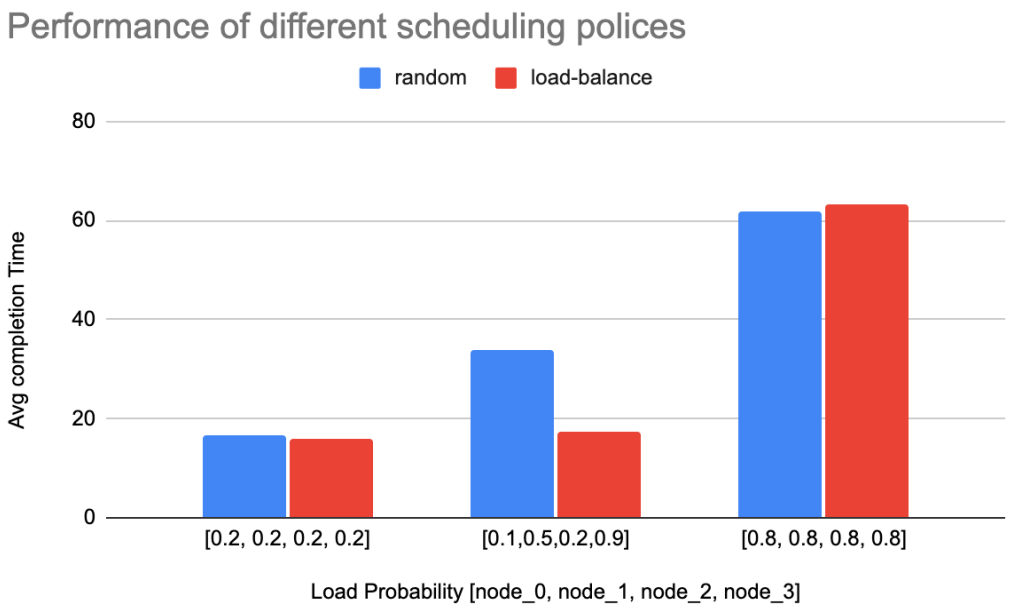
# Testing Description

- **Test cases**

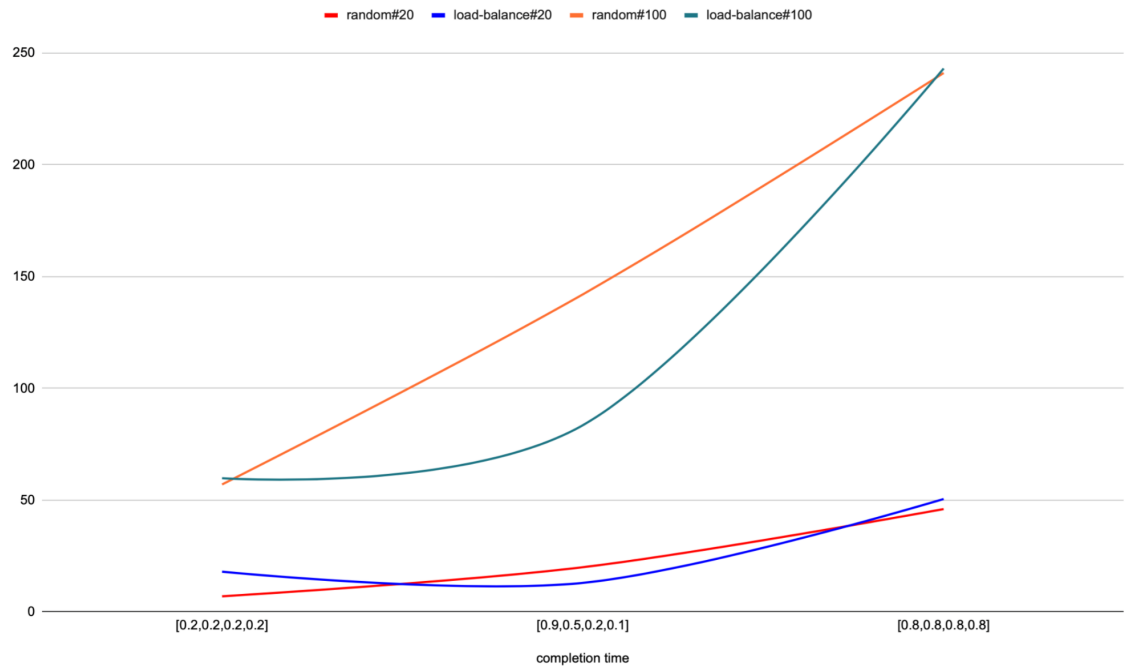
- *Negative cases:*
  When input_dir is empty, there is no output in output_dir. The result is as expected. When the load is low ([0.2,0.2,0.2,0.2), the performance shouldn't be bad. The result is as expected by looking at the following result table.

- The following are all test cases we tried. We tested three different load probabilities (equal low, equal high, all different) under two scheduling policies. a) [0.2, 0.2, 0.2, 0.2] b) [0.8, 0.8, 0.8, 0.8] c) [0.1, 0.5, 0.2, 0.9].

| Load Probability | Random(avg time) | Load-Balance(avg time) |
|---|---|---|
| [0.2,0.2,0.2,0.2] | 16.83 | 16.08 |
| [0.1,0.5,0.2,0.9] | 33.99 | 17.19 |
| [0.8,0.8,0.8,0.8] | 61.99 | 63.32 |




*"The average time difference between random policy and load-balance processing 100 images"*




*"We can see that when processing more images(20 vs 100), the final results will show a **larger difference** between the load-balance policy and the random select policy."*

## Performance evaluation results of the system

- Situation1:
  Due to the [0.2, 0.2, 0.2, 0.2] & [0.8, 0.8, 0.8, 0.8] are both the policy with the same load probability in each node, hence under this circumstance, we would expect the average completion time of "load-balance" policy would be close to the average completion time of "random" policy. One reason for the similar formance may be that all nodes have the same probability so load-balance policy can't really find the "best" node when it assigns tasks, it also spend some time inspecting each node's status which may take more time than randomly policy. It may also take more time for load-balance policy since the node can reject the task, which may cause more time.

  We do see that performance of using low load probability is much better than using high load probability regardless of policies.

- Situation2:
  Under the circumstances that the probability of [0.9,0.5,0.2,0.1] which is different between each node, we would expect the "load-balance" is always better than the "random". Due to the reason that we can always choose the nodes with most likely not reject the image. Hence there's definitely optimization inside total time consumption. We can also see that the advantage of using load-balance policy is more obvious when processing more images.