Problem Description

My project is a preliminary centralized cloud computing infrastructure that simulates the real cloud computing process on a single machine. The basic idea of the cloud computing infrastructure is that to allow the client to submit some programs, which will be compiled and executed on some high performance worker nodes, and retrieve the output results. The project uses a master/worker approach to fulfill a client's requests. For details of the architecture design, please refer to Architecture Overview part.

User Manual

We will run the project in the Virtual Machine. Go to the root of the project and run make file.

```
      oscreader@OSC:~/OS/project$ make all
```

After this, the directory of the project is as the following tree:

The directory listing of the project is as the following tree:

```
|-- bin
|   |-- deploy
|   |-- master
|   |-- results
|   |-- status
|   |-- worker1
|   `-- worker2
|-- cloud-nodes.txt
|-- makefile
|-- myapp
|   |-- bin
|   |   |-- test
|   |   `-- test2
|   |-- makefile
|   |-- results
|   |-- result.txt
|   `-- src
|       |-- test2.c
|       `-- test.c
|-- obj
|   |-- deploy.o
|   |-- master.o
|   |-- results.o
|   |-- status.o
|   |-- worker1.o
|   `-- worker2.o
|-- src
|   |-- client_deploy.c
|   |-- client_results.c
|   |-- client_status.c
|   |-- master.c
|   |-- worker1.c
|   `-- worker2.c
|-- worker1
`-- worker2
```

The bin file contains all the executable programs. Since this is a simplified version of cloud infrastructure running on a single machine, `worker1` and `worker2` are also in the same directory. `Cloud-nodes.txt` stores the information of the workers, specifically port number in our case.

In my implementation, there are only 2 workers, but it's easy to add however many workers you want. All you have to do is make a copy of `worker1`.c and name is as `worker<#>`.c and change the UDP port and TCP port number and add them in `cloud-nodes`.txt, and in the makefile add the corresponding compilation command. Then open 3 other terminal windows, and direct to the root of the project. In the user manual, I will present how the cloud computing works with 2 workers. In terminal 1, run

```
oscreader@OSC:~/OS/project$ ./bin/worker1
```

In terminal 2, run

```
oscreader@OSC:~/OS/project$ ./bin/worker2
```

In terminal 3, run

```
oscreader@OSC:~/OS/project$ ./bin/master
```

Then in the original terminal that runs "make all", we can submit three requests as a client:

1. Deploy request

   To deploy a program in the cloud, we need to submit the number of replica, makefile path as well as source file paths, for example

   ```
   oscreader@OSC:~/OS/project$ ./bin/deploy 2 myapp/makefile
   myapp/src/test.c
   ```

   In `myapp` folder, `makefile` is located in the root, and source files are located in `src` folder. There is also a `result` folder to store the output results from the deployment. A successful deployment will display the received job tickets as follows:

```
 Job deployment successful - JOB TICKET 0
```

With the job ticket id, we can then check whether the submitted deployment is finished or still running.

2. Status request
   To check if our submitted deployment has been finished, in the command line, we can do as follows

```
 oscreader@OSC:~/OS/project$ ./bin/status 0
```

If the deployment has already been finished, then the console will display

```
 received status: COMPLETED
```

If the job is still running, then "RUNNING" will be displayed. If the job id doesn't exist in the master's record, then "INVALID" will be shown.

3. Results request
   To retrieve the output results of the deployment, we can simply provide a output path as follows:

```
 oscreader@OSC:~/OS/project$ ./bin/deploy 0 myapp/results
```

A successful results looks like the following:

```
 job 0 results retrieved
```

And you will be able to go to `myapp/results` to check the results. Based on the previously submitted value of replica, it will generate an equal number of folders returned by the assigned work nodes. After a successful result retrieval, the `myapp/results` directory will look as follows:

```
|-- myapp
|   |-- results
|   |   `-- 0
|   |       |-- result0
|   |       |   `-- result.txt
|   |       `-- result1
|   |           `-- result.txt
```

where `0` denotes the job id. `result0` means the first received result from the assigned workers, and so on.

Architecture Overview

In this part, I provide a detailed description of the architecture design.

- Initialization:

    - Worker: We first initialize N worker programs. Each worker firstly creates 2 threads. One thread will open a TCP server to take charge of file exchange request. The other thread will open a UDP server socket to be responsible for 1) receiving request of executing clients' programs and 2) updating the worker status and job status. Each process has its own working directory to store the input and output files.
    - Master: After the initialization of workers, we then initialize the master program. The master program creates N threads. In each thread, a UDP client socket is created and sends a message to the a worker's UDP socket, the port number of which is stored in `cloud-nodes.txt`. After receiving confirmation from the corresponding worker (which means that the worker is available), the thread runs a while loop to wait for the update from the worker. In the main thread, the master creates a UDP server to communicate with the clients. It runs a while loop to wait for clients' request. In addition to the above initialization, in the main thread the master also creates 4 lists:
        - the 1st list records the running job by their ticket_id
        - the 2nd keeps track of the ports they use
        - the 3rd list is used to record the number of replicas
        - and the 4th list is used to record whether all replicas have been completed. E.g, 0 means no replica has been completed executing, and if the counter equals the value of replica, then it means that all replicas have finished executing.

      In addition, the master also creates a global variable `job_id`. The master also initializes a linked list to keep track of the usage of the workers. The front of the linked list will be the workers that are least recently used.

- Use case 1: deploy
    - Client: Upon deployment, the client starts the executable program `deploy`. The `deploy` program firstly creates an UDP socket, where the port number is consistent with the master's port number. It then sends a deploy request to the master. Essentially the client sends a string "DEPLOY" to the server, and waits for the response. The master's response will be stored in a list of port numbers that corresponds to M workers, where M is the replica. The client also waits for the master to send the `job_id`, which will then be sent by the client to the corresponding workers. Then the client creates M threads to connect with the workers through TCP sockets. Each thread creates a buffer and a file descriptor to read all files

mentioned in the arguments, and continuously send the buffer to a worker. Each thread waits for workers' confirmation that the file exchange has been completed, after which the thread terminates. The main thread waits for all secondary threads to terminate.

    - Master: when the master's UDP socket receives a request from the client, it first checks the request type. If the request type is "DEPLOY", it then creates a thread that specially deals with deploy request. In this thread, the master first sends the `job_id` to the client, and increment the global variable `job_id`. The update on `job_id` requires a mutex lock to ensure concurrency. Then it checks its linked list to obtain the M least recently used workers. Then the master sends the corresponding port number to the client. In its 4 lists,
        - it inserts `job_id` in the 1st list
        - Inserts [port_1, port_2..] in the 2nd list
        - Inserts `replica` in the 3rd list
        - and finally inserts status counter initialized to 0 to the 4th list. Following that, the master waits for M workers' notification through UDP socket to know that the file execution is completed.
    - Worker: each worker's TCP socket waits for a request. Whenever a request is received, it creates a new thread to deal with the corresponding request type. If the type is "DEPLOY", it creates a buffer and opens a new file with its file descriptor. Worker also receives the `job_id` from the client so that it creates a directory to store all input and output files for the client. For each file being transferred, each worker keeps receiving buffer and writes until the buffer size is smaller than the predefined maximum buffer size `BUFFSZ`. Then the worker will fork and compile received files and execute the programs. The current process suspends its execution until it receives signal `SIGCHILD`. Then the process sends a UDP message to the master.
- Worker finishes execution:
    - Worker: when the child process of the thread is done executing and has generated the output file, it sends the `job_id` to the master through UDP socket
    - Master: when the master receives this message of list. It increments the status counter of the corresponding job in the 4th list.
- Use case 2: status

    - Client: the `status` program uses a UDP client socket, and sends the request and its `job_id` to the master, and wait for its response. After receiving the response, display the status.
    - Master: the master program checks the request type, and creates a thread that deals with “STATUS” request. In this thread, it first checks if the received `job_id` is in the 1st list. If not, it sends ‘INVALID’ to the client. If yes, it then check in the corresponding status in 3rd list. If the corresponding counter equals the number of replicas, it sends “COMPLETED” to the client, and “RUNNING” elsewise.
- Use case 3: results
    - Client: the `results` program uses a client UDP socket first and sends the request and its `job_id` to the master, and waits for the response. If the client receives error message, it displays error and terminates. Otherwise, based on the received port number list, it sends its `job_id` to the corresponding workers and retrieves the result through a TCP client socket. The file exchange procedure is similar to that in deploy part, except it’s the other way around (the worker writes in the buffer and send it to the client).
    - Master: the master program checks the request type, and creates a thread that deals with “RESULTS” request. In this thread, it first checks if the received `job_id` is not in the 1st list. If not, it returns an error message. If the `job_id` is indeed in the 1st list, the master checks whether the corresponding status counter equals to the corresponding replica value. If yes, return “COMPLETED”, and then returns the corresponding port number list to the client. If not, it send “RUNNING” to the client.
    - Worker: the worker’s TCP socket checks the request type from the client. If it’s “RESULTS” type, it creates a thread that exchange the `#job_id` job’s output file through the TCP socket. Again, this file exchange procedure has been described in Use case 1, so we omit the description for simplicity.

Problem Analysis

- Worker:
    - Socket choice:
      The worker uses both TCP server socket and UDP server socket. It uses TCP protocol because the file transfer is safer given no loss of data during transmission. For the communication between the worker and the master,

    since there is not much data to transfer, it will be enough to use UDP socket.
  - concurrency:
    To communicate between the thread that deals with client's deployment and the thread that communicates with the master, I use two global variables: `executed` and `id_to_send`. Upon completion of any deployment, it uses `mutex_lock` to lock the critical resources - the 2 global variables - and updates `id_to_send` and changes `executed` to 1. Then the thread that deals with the deployment uses `cond_signal(&cond)` to notify the UDP server thread to send a notification to the master. In the main thread while loop, it first `mutex_lock`, then `cond_wait` on cond, whenever it resumes execution, it changes `executed` to 0.
- Master:
  - Socket choice:
    The master only uses UDP sockets. It creates one UDP client socket for each worker in its secondary threads. In the main thread, the master creates a UDP server socket to communicate with the clients' request. I don't use TCP for master for the sake of avoiding redundant data transfer. This reduces memory usage for the master and optimizes throughput in terms of handling requests.
  - Concurrency:
    All 4 initially created lists are global. So whenever we need modify their value, we need to use `mutex_lock`. We also have another global list `deployed`, all values of which are initialized to 0. that is used to solve the communication between the UDP client thread and the threads dealing with deploy request. Whenever the master sends the `job_id` to the client, it changes all values of the list `deployed` to 1, and then `cond_broadcast`(&cond_deploy) so that the corresponding UDP client threads can resume working and waits for the worker's update status message.
  - Worker scheduling:
    In reality, using M east recently used (M-LRU) strategy can't ensure that the worker assigned will finish the deployment faster. For example, the worker is working on running some very large program. However, M-LRU still remains one of the most intuitive choice.
- Client_deploy:
  - Socket choice:

    Since the communication between clients and masters is always with very small data, we use UDP sockets instead of TCP sockets. In terms of file transfer, clients uses TCP client socket to avoid data loss.

- Client_status:
    - socket choice:
      The client only receives one string. So a UDP socket is enough.
- Client_results:
    - Socket choice:
      Similar to the socket choices of client_deploy.