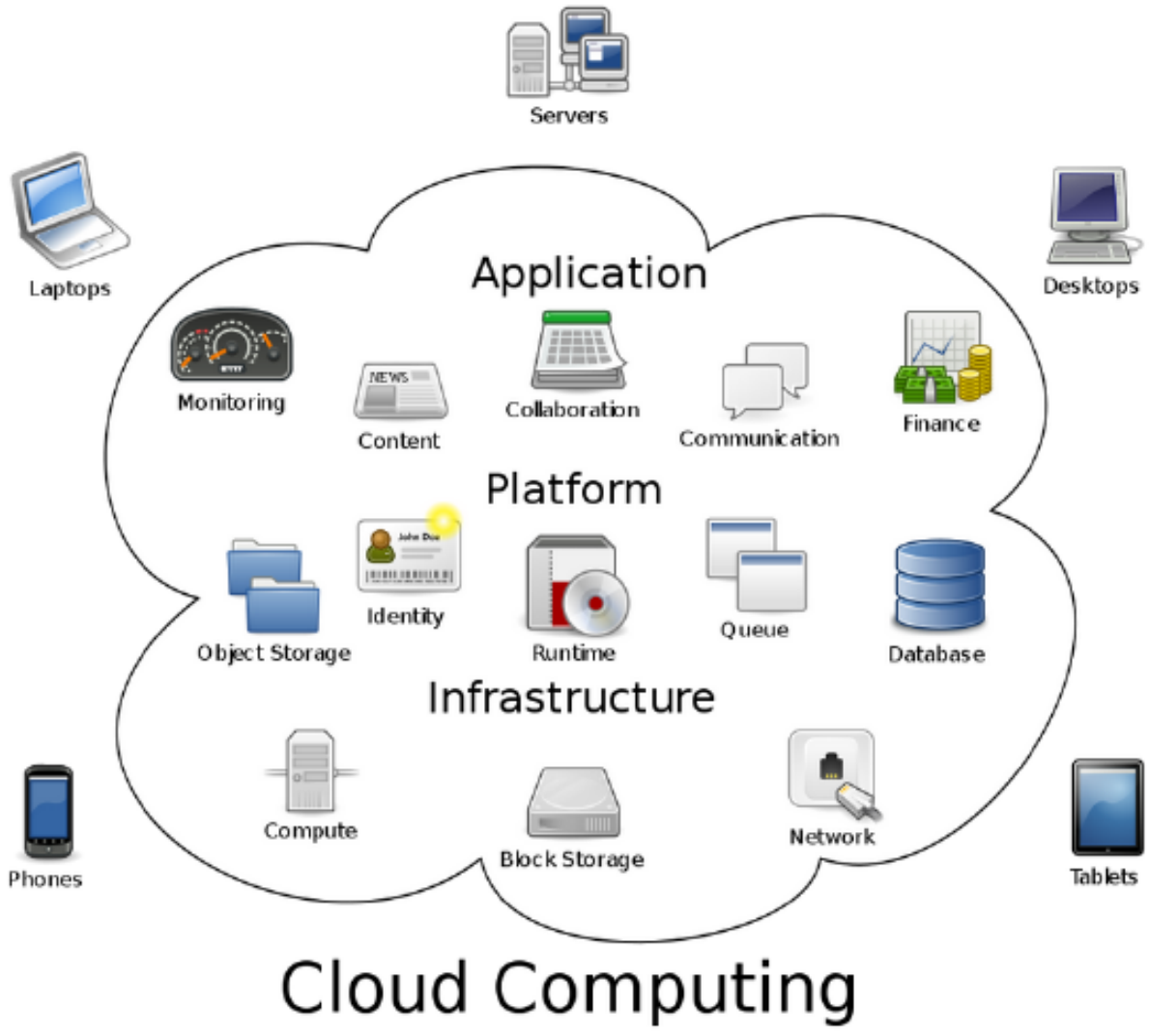
Bosen Yang by570@nyu.edu

Cloud Computing Infrastructure



Problem Description

This project is to create a centralized computer network/distributed system where we have a master server managing a bunch of computational resources (worker servers). A client program can depend on the computational power of those resources to run some code. To do this, the client will have to send the source code to the master server, and the master server will decide where in the computational resources those files will be executed. Additionally, a client can ask the master server for code execution status and the execution results.

User Manual

Please run this program in **macOS** rather than Linux in the virtual machine, because the whole program is written and tested in macOS.

Compilation

The user can compile all the code at once by calling make in the terminal. However, the Makefile file contains no rules for running any executables. The clt_file folder contains all the files created and needed for the client. The wkr_file folder contains all the files created and needed for the worker server. The mst_file folder contains all the files created and needed for the master server.

Initialization

Before any client makes a request, all servers, worker and master, should already be launched. We'll launch the worker server first and then the master server, where the latter should have the information of the former and connect them automatically.

A worker server launch call has the following format:

```
$bin/worker <portnumber>
```

In our case, there will be 5 worker servers in total, with the portnumber of range from **8400** to **8404**, inclusively. Therefore, the user will have to open 5 terminal windows and launch 5 worker servers respectively with the format above.

A successful worker server launch call would look like so:

```
$bin/worker 8400
```

```
Worker server successfully launched
```

After we launch all the worker, the user should open another terminal window and launch the master server.

A successful master server launch call would look like so:

```
$bin/master
Master server successfully launched
```

Cloud services

Users start their own client program in a terminal to interact with the cloud, and can then enter text requests. A user can make three different kinds of requests:

i. *deployment (deploy)*
ii. *status inquiry (status)*
iii. *results retrieval (result)*

Deployment

A *deployment* request starts a job in the cloud by providing the source code of a program (including a makefile) to be uploaded, compiled, and run on the cloud. Users can ask to deploy replicas of the same application concurrently on multiple nodes, at most one replica per cloud node. A deployment request returns a job ticket: a unique integer value that represents the job for future requests: status inquiries and results retrieval.

The client should send a compressed file of tgz format to the master server, same as the format of our homework submission. We trust the client will send the right thing, with everything well arranged in the tgz file and the presence of a Makefile file.

A deployment call has the following format:

```
$deploy <hostname> <#replicas> <tgzfilename>
```

where <hostname> is the string format name of the master server (the first argument to put into the getaddrinfo function), <#replicas> represents the number of replicas the user wishes to start on different cloud nodes, < tgzfilename> is the path to the tgz file (including both the path and name).

A successful deployment request would look like so:

```
$deploy localhost 2 clt_file/yang.bosen.by570.osc.03.tgz
Job deployment successful - JOB TICKET 3385
```

After this call, in the folder of master server files called “mst_file”, we can find a source file called “sourceFile-<#job>” and a folder called “stdoutFolder-<#job>” of stdout files, each returned from one of the worker servers. For example, for the example deployment request above, we can find in folder mst_file the corresponding source file called “sourceFile-3385” and the stdout file folder called “stdoutFolder-3385”.

As for the worker server, we’ll find the source file, the folder where we extract the source file into and the stdout file corresponding to this replica. The way of naming is similar, so I won’t talk more about it.

Status inquiry

A *status inquiry* uses a job ticket as argument to return the status of that job: COMPLETED, RUNNING, or INVALID. A job is COMPLETED when all of its application replicas have terminated their execution. A job is RUNNING when not all of its application replicas have terminated their execution but the job is successfully deployed. A job is INVALID when no client has ever received this job ticket.

A status inquiry call has the following format:

```
$status <hostname> <#job>
```

where <hostname> is the string format name of the master server (the first argument to put into the getaddrinfo function) and <#job> represents the job ticket returned to the user after a successful deployment request.

A successful inquiry call would look like so:

```
$status localhost 846
JOB#846 – RUNNING
```

A unsuccessful inquiry call would look like so:

```
$status localhost 855
JOB#855 – INVALID
```

Results retrieval

A *results retrieval* uses a job ticket as argument to return a tgz file containing the outputs of all the program replicas pertaining to a job. To make things simpler, we will only consider the text output on stdout.

A results retrieval call has the following format:

```
$result <hostname> <#job> <directory-path>
```

where <hostname> is the string format name of the master server (the first argument to put into the getaddrinfo function), <#job> represents the job ticket returned to the user after a successful deployment request, and <directory-path> is the path of the directory where the output files must go. If the specified directory does not exist before the call, then it gets created as a result of the call. Calling a results retrieval on a completed job shall return tgz file of text files, one per application replica, in a directory specified by the user. Calling a results retrieval on a job that is not completed shall return an error message.

A successful inquiry call would look like so:

```
$result localhost 125 clt_file
Result retrieved: clt_file/stdoutTar-125.tgz
```

A unsuccessful inquiry call would look like so:

```
$result localhost 855 clt_file
Task not COMPLETED: Undefined error: 0
```

After this call, in the folder of master server files called “mst_file”, we can find a compressed file called “stdoutTar-<#job>.tgz” of all the stdout files, returned in the deployment step. For the client, we can find a file with the same name and content in the directory specified by the client. This means we successfully send the output to the client.

Architecture Overview

This section will explain the data structures and variables used, the implemented operations and how they work. This section will only include the important ones, because the naming and use of other variable are very obvious while reading the code.

Deployment

- **General view**
  In a *deployment* call, the client will first build connection with the master server. Next it specifies its request, in this case, deployment. Then it send an integer value of how many replicas it wants. After that, the client sends the compressed source file and waits for the assigned ticket number after sending. After the client receives the ticket number, it prints it out and exit.

  In the master server side, the master server creates a thread to handle each new connection of whatever request. In this case, after receiving the number of replicas, it first select the worker servers to execute the replicas and connect with each of them. While receiving packets of the source file from the client, the master simultaneously send the packets to each of the worker. After receiving the whole file, the master returns the job ticket to tell the client to exit. Then it enters the last stage to wait for the worker to return the outputs.

  As for the worker server, it creates a thread for every new connection. So each thread is only in charge of one replica from one client. Thread will fork and call exec functions to call functions like tar and make.

- **int service**
  This variable appears in all deploy, status and result files and has the same meaning in them. It's what is first sent to the master server after a connection is established, specifying the type of the request (0 for deployment, 1 for status check and 2 for result retrieval).

- **int ticket**
  This variable appears in both master server and deploy files and is used in command line while making status inquiry and results retrieval. As the name suggests, it represents the ticket number of a certain client. When we launch the master server, it's value is set to 0. Each time a client makes a deployment request, its current value is returned to the client and gets incremented by 1 after that.
  This variable a critical as accessed by many threads, so we need a mutex while modifying it. Also, each thread will have a local variable (named **local_ticket** in the master server) to store the ticket value they're assigned.

- **int nodes[NB_NODE]**

This variable is the list of ports from which we can connect the worker server and macro NB_NODE means the total number of workers server we'll have (change this value in header file server.h to have more worker servers starting from port number 8400). Because I wrote and tested the program only on my machine, these worker servers should have different.

- **int nodes_counter**
  In my implementation, the worker servers are assigned in a circular order. Since we store the information (port number) about worker servers in an array, we'll only have to locate the next worker to assign. For example, for the array of 4 ports called nodes[], nodes_counter is set to 0 at the beginning. A client makes a request of 3 replicas and we assign nodes[0], nodes[1] and nodes[2] to it. The value of nodes_counter is then set to 3. The next client makes a request of 2 replicas and we assign nodes[3] and nodes[0] to it. The value of nodes_counter is then set to 1.

- **int sock_worker[nb_replicas]**
  While the master sends source file to workers, it builds connection with each one of them. This variable stores the port number for each connection.

- **int sc, scom**
  sc stands for the connection socket and scom stands for the communication socket in both worker and master program.

Status Inquiry

- **General view**
  In a *status* call, the client will first build connection with the master server. Next it specifies its request, in this case, status. Then it send an integer value of the desired ticket number. After that, the client waits for an integer reply from the master specifying the job status.

  In the master server side, the master server creates a thread to handle each new connection of whatever request. In this case, after receiving the ticket number, it checks the job status and return the integer encoded status value.

- **int ticket_status[MAX_CONNECTION]**
  ticket_status is an array of integers representing the job status. Since we start assign job ticket from 0, the index of this array can correspond to the job of this ticket number. There are potentially at most MAX_CONNECTION deployments so that we give this array with equal size.

Result Retrieval

- **General view**
  In a *result* call, the client will first build connection with the master server. Next it specifies its request, in this case, result. Then it send an integer value of the desired ticket number. After that, if the ticket number is valid, the client waits for the compressed file containing all output files to be put into the desired directory entered in terminal line. Otherwise, the client will be notified that the job is invalid and prints an error message.

  In the master server side, the master server creates a thread to handle each new connection of whatever request. In this case, after receiving the ticket number, checks the status and send it back. If the job is completed, the master compressed the folder of output files (obtained by the ticket number entered) and send it back to the client.

Implementation Analysis

Design choices

- **Network Protocol**

I'm using TCP protocol for all the communication among participants in the network. Simplicity of programming is part of the reason. But more importantly, in our case, every piece of information is important (corrupted data can lead very bad results of the program) and TCP can provide better protection of the communication content. Additionally, thanks to the increasing hardware capacity (faster computational power, as predicted by the Moore's law, and faster network speed), the cost of TCP is much lower compared to the past (mainly time cost).

- **Scheduling Policy**

  In my very intuitive implementation, the worker servers are assigned in a circular order. Of course this method can guarantee that every worker are assigned almost equal amount or works. However, considering the fact that some replicas only take a short time to run while others can take long, cases can happen where we assign a replica to a heavily burdened worker while there are idle workers.

  This scheduling policy can be improved by monitoring the workload and assign job firstly to more free workers. This method only requires an extra array recording the workload and a function to locate the most free worker.

- **Multi-threading in communication**

  Another important point was where to use multi-threading. Of course the master server is where we really have to apply multi-threading because it's so burdened with all kinds of requests in this centralized structure. The master creates a thread for each new connection so that the server can handle different requests concurrently. We can do the same for worker server, because in this way a worker server can take multiple replicas from different client (one thread is in charge of one replica).

- **Flaw with multi-threading in my implementation**

  I used some exec functions in my code. My strategy is that let a process fork and have parent to wait for the child calling exec to run the source code. The flaw appears when the parent process waits. The wait function blocks the whole process, all the threads. So the thread in charge of network connection will also be blocked, causing network connection to be postponed after the execution of the exec. One easy and efficient solution was to create processes rather than threads at where we will call exec functions, like in result retrieval or for the worker server. I'm only stating this strategy rather than implementing it because I thought of this only after I completed coding the whole program and don't want to recode and debug anymore. Probably I'll do it in a 2.0 version.

- **Storage of output files**

  In my implementation, all the output files will be sent back to the master server, which only increases the burden of the master. This mechanism can be improved if we don't send back the output files. Instead, when a client performs result retrieval, the master only send back the information about assigned workers (IP address, port number), and then the client will itself connect those worker server to retrieve the result.