Operating Systems - Fall 2018 - Final Project

Cloud Computing Infrastructure

Introduction

Cloud computing allows users to deploy and run their own services on top of a network of computers (cloud nodes) that they share with other users. The rationale is similar to that of a shared-bike service such as Ofo or Mobike: the company leases resources (bikes for Ofo, computers/nodes for cloud providers) at a low cost, and provides the infrastructure for provisioning these resources when they are requested. In the case of clouds, users may run their programs on multiple nodes without worrying about the management nor the maintenance of the hardware infrastructure. Further than that, users rarely even know where the programs that they launched are being executed. From their own device, they can request the deployment of their code on the cloud and wait for the results of its execution.

For this project, you will design and implement your own cloud computing infrastructure.

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 (`results`)*

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.

A deployment call has the following format:

```
$deploy <#replicas> <makefile> [<source_files>]+
```

where `<#replicas>` represents the number of replicas the user wishes to start on different cloud nodes, `<makefile>` is the path to the makefile for compiling the user application, `[<source_files>]+` is a list containing one or more paths to the source code of the user application, each separated by a whitespace.

A successful deployment request would look like so:

```
$deploy 3 myapp/makefile myapp/source1.c myapp/source2.c
Job deployment successful - JOB TICKET 3385
```

on what condition will it succeed
1. **<#replicas> == nbFiles**
2. **the files do exist**

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 status inquiry call has the following format:

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

where `<#job>` represents the job ticket returned to the user after a successful deployment request.

Results retrieval

A *results retrieval* uses a job ticket as argument to return 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:

```
        $results <#job> <directory-path>
```

where `<#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 a set 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. **check status first**

Cloud architecture: master and workers

For this project, you will use a master/worker approach: a single master server handles user requests from clients, dispatches job tasks to worker server nodes, and monitors the jobs' progress. The software that manages your infrastructure will therefore consist of three programs:

1. a *client* through which users can deploy their applications, check their status, and retrieve their results,
2. a *worker server* which runs client application replicas deployed on its local node,
3. and a *master server* which manages client requests and dispatches application tasks on worker nodes.

You may assume that the topology of the cloud network is static: the locations of all the worker server nodes that compose your cloud are fixed and known in advance. The list of these locations is accessible as a text file called cloud-nodes.txt and located at the root of the directory you submit.

A second assumption is that the cloud nodes are heterogeneous in terms of hardware and software. That is why it is necessary to upload and compile the source code of user programs on the cloud nodes before running them.

Finally, your programs must use multi-threading for concurrency.

Submission format

You are expected to provide both your source code and a report.

Report

The report should be *at most 20 pages long*, in *PDF format* only.
Here is its suggested structure:

1. Problem description (ie. a short explanation, in your own words, of what your software does).
2. A mini-user manual (how to compile your source, which files are important and where they are located, how to start the program and what behavior to expect at runtime.)
3. Architecture overview (data structures and variables used, implemented operations and how they work, with diagrams to represent relationships between variables and processes)
4. Problem analysis (explanation of your design choices, of the problems you encountered and how you solved them.)

The report must not contain your code.

Code

The code submission must comply with the usual assignment submission guidelines, as detailed in the worksheet for assignment 01. Additionally, your makefile should comprise a compilation directive called `cloudapps` which generates all of your binary executables (client, worker server, and master server), ready to be run.

You must submit your work, your code as a gzipped archive and your report in PDF format, before 11.55pm on December 16th, 2018. There can be no deadline extension.

Evaluation criteria

Completeness

The submitted software provides all of the expected features: client access to services, multi-threaded concurrency, and load balancing of the tasks over the cloud nodes.

Design

The proposed design specification is logically correct, with a focus on concurrency and efficiency, proper handling of corner and edge cases, no known boundary errors, and no redundant or contradictory conditions.

Implementation

The submitted code compiles and runs smoothly, it matches perfectly the design specification, it reliably produces appropriate results for all inputs tested.

Clarity

The report explains and details clearly the overall design and the inner mechanisms associated with each feature. The code provides adequate comments for all major functions, variables, or non-trivial algorithms. Code formatting and indentation aid readability.

Note that the assessment focuses on your report (70%), not on your code (30%). Work on your design choices first, and document them properly. Then plan the data structures and variables you will use; document all this properly too. Then, and only then, work on your implementation; keep it as simple as possible. It is far better to submit an incomplete (but functional) software with a detailed and accurate report, than to submit a full-fledged cloud infrastructure ready for commercialization with a flimsy two-page README file. A well-written report that explains why the application crashes will get you a better grade than an application that works perfectly without any explanation... Finally, once your implementation is finished, prepare a manual that allows the average user to use your software (testing it on a fellow student who does not major in CompSci should do the trick).