Applied Cloud Computing
Uppsala University – Autumn 2020
Report for Lab 3

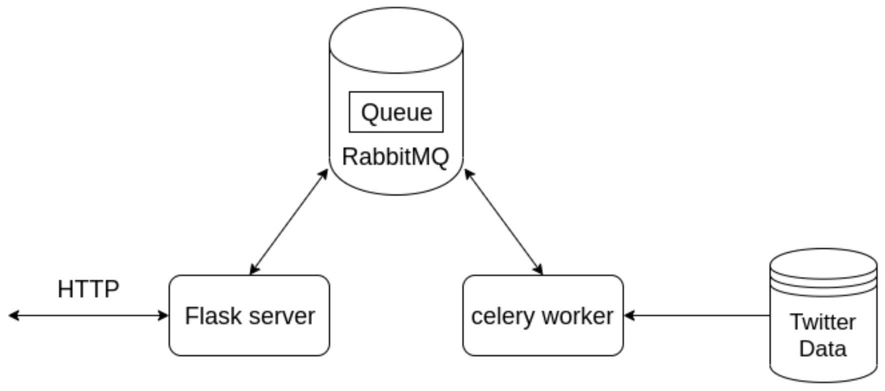
Li Ju

9th October 2020

I Task 1:

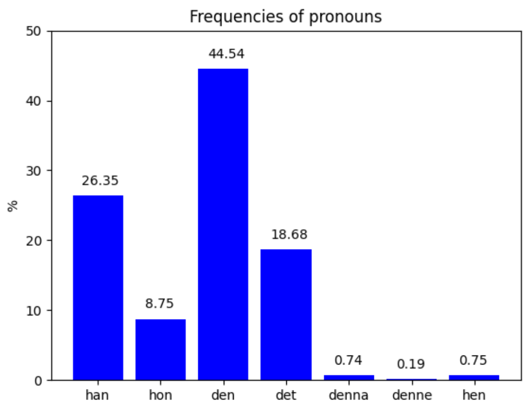
For the task 1, a Celery/RabbitMQ based program to analyze a twitter dataset should be deployed on a single virtual machine, and a REST API should be deployed that the program can be provided as a SaaS (Software-as-a-Service) and be accessed remotely.

Architecture Firstly the architecture of the application is shown as following. Clients use HTTP API to access the flask server first. Then the flask server will submit the requested task to the RabbitMQ server and it will forward the task to a celery worker. Then the celery worker will read data, analyze them and return results.



Work pipeline of each worker Each worker will receive a file name assigned by the broker. The worker will read the json file from local storage as a list of dictionaries (only lines in even numbers will be read in). Then all items in the list whose "retweeted_status" is None will be selected and "text" of these items will be appended to an empty list of strings. Here a result dictionary is created: all target pronouns are keys of the result dictionary and value of each key is the number of the pronoun appears (initialized as 0). For each sentence in the list of texts, the sentence will be split to a list of words (strings). And the list of words will be traversed to count how many times each target pronouns appear and update the result dictionary. After we traversed all texts, the result dictionary will be returned and passed. After all requested files are analyzed, all result dictionaries will be summed by keys and the final result will be returned via flask api.

Frequencies of pronouns With counts of all target pronouns, a frequency chart of pronouns can be plotted with matplotlib and shown as following.

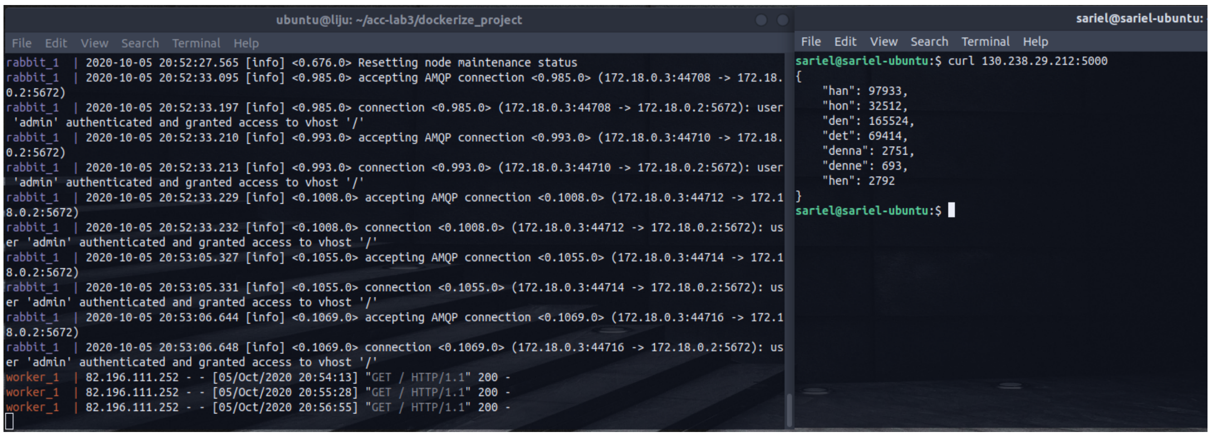


Scalability The scalability of the application (more workers can be elastically deployed) is of vital importance (as we will see in Task 2). There are 101 raw twitter data files and all files should be analyzed for each query. In my implementation, each time the flask server is requested, it will push 101 celery tasks to the job queue and each worker is only working on one file for each task: we let celery schedule tasks. After all tasks are done and 101 dictionaries with statistics of pronouns in each file will be collected and summed by the flask server. Finally the flask server will forward the summed results back to clients. This ensures the scalability of the application: with more workers, celery is able to schedule all tasks and balance loads of those workers.

Portability Further, to improve the portability of the application, the application is containerized in Docker. The application is separated into two parts: RabbitMQ micro-service and Flask/Celery service. The code can be found here.
To get service started,

1. firstly we need to "git clone" the repository and install "docker" and "docker-compose";
2. because raw data is out of file limit of GitHub (100MB), to deploy the containerized project properly, we also need to copy all raw twitter data files in "dockerize_project/project/data/";
3. run "docker-compose up -d" in "dockerize_project/" directory;

Then the service should be able to be accessed by "curl ⟨public ip⟩:5000" on any machine remotely. The running status (to show status, run "docker-compose up" instead of "docker-compose up -d") and the access result are shown as following.


ubuntu@liju: ~/acc-lab3/dockerize_project
File Edit View Search Terminal Help
sariel@sariel-ubuntu:
File Edit View Search Terminal Help

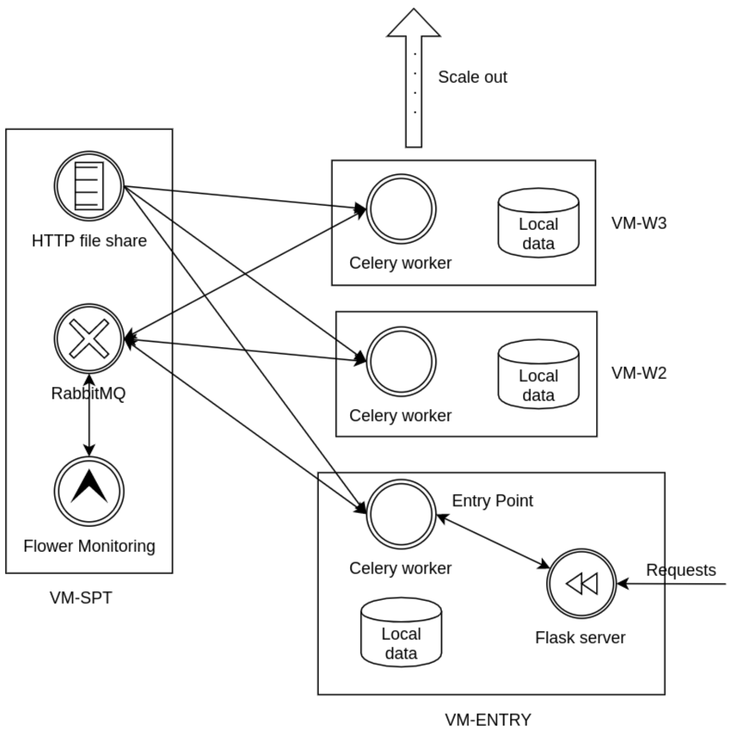
```
rabbit_1  | 2020-10-05 20:52:27.565 [info] <0.676.0> Resetting node maintenance status
rabbit_1  | 2020-10-05 20:52:33.095 [info] <0.985.0> accepting AMQP connection <0.985.0> (172.18.0.3:44708 -> 172.18.
0.2:5672)
rabbit_1  | 2020-10-05 20:52:33.197 [info] <0.985.0> connection <0.985.0> (172.18.0.3:44708 -> 172.18.0.2:5672): user
 'admin' authenticated and granted access to vhost '/'
rabbit_1  | 2020-10-05 20:52:33.210 [info] <0.993.0> accepting AMQP connection <0.993.0> (172.18.0.3:44710 -> 172.18.
0.2:5672)
rabbit_1  | 2020-10-05 20:52:33.213 [info] <0.993.0> connection <0.993.0> (172.18.0.3:44710 -> 172.18.0.2:5672): user
 'admin' authenticated and granted access to vhost '/'
rabbit_1  | 2020-10-05 20:52:33.229 [info] <0.1008.0> accepting AMQP connection <0.1008.0> (172.18.0.3:44712 -> 172.1
8.0.2:5672)
rabbit_1  | 2020-10-05 20:52:33.232 [info] <0.1008.0> connection <0.1008.0> (172.18.0.3:44712 -> 172.18.0.2:5672): us
er 'admin' authenticated and granted access to vhost '/'
rabbit_1  | 2020-10-05 20:53:05.327 [info] <0.1055.0> accepting AMQP connection <0.1055.0> (172.18.0.3:44714 -> 172.1
8.0.2:5672)
rabbit_1  | 2020-10-05 20:53:05.331 [info] <0.1055.0> connection <0.1055.0> (172.18.0.3:44714 -> 172.18.0.2:5672): us
er 'admin' authenticated and granted access to vhost '/'
rabbit_1  | 2020-10-05 20:53:06.644 [info] <0.1069.0> accepting AMQP connection <0.1069.0> (172.18.0.3:44716 -> 172.1
8.0.2:5672)
rabbit_1  | 2020-10-05 20:53:06.648 [info] <0.1069.0> connection <0.1069.0> (172.18.0.3:44716 -> 172.18.0.2:5672): us
er 'admin' authenticated and granted access to vhost '/'
worker_1  | 82.196.111.252 - - [05/Oct/2020 20:54:13] "GET / HTTP/1.1" 200 -
worker_1  | 82.196.111.252 - - [05/Oct/2020 20:55:28] "GET / HTTP/1.1" 200 -
worker_1  | 82.196.111.252 - - [05/Oct/2020 20:56:55] "GET / HTTP/1.1" 200 -
```

```
sariel@sariel-ubuntu:$ curl 130.238.29.212:5000
{
    "han": 97933,
    "hon": 32512,
    "den": 165524,
    "det": 69414,
    "denna": 2751,
    "denne": 693,
    "hen": 2792
}
sariel@sariel-ubuntu:$
```

II Task 2

In task 2, the scalability performance of the system will be tested. However, the implementation in task 1 only works in one single machine and can hardly be scaled out. To improve the scalability of the system, it is redesigned and re-implemented.

Architecture The architecture of the improved system is shown as following.

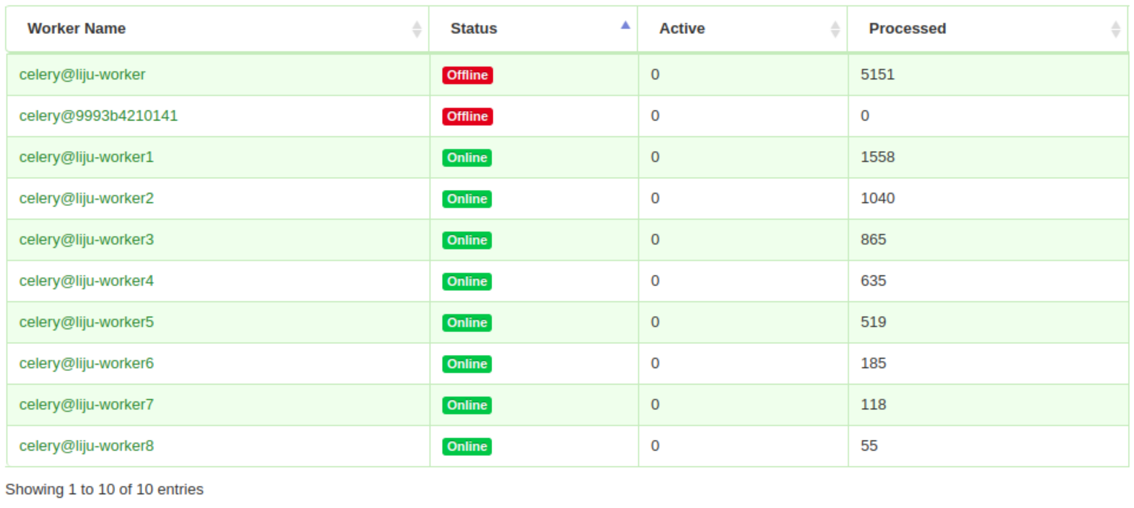


1. In the machine named "VM-SPT", 3 services are deployed: RabbitMQ, HTTP file sharing and Flower. The HTTP file sharing is used to store the raw twitter data, so that when new workers are deployed, they can copy a local duplicate of raw data from this HTTP server automatically, otherwise one must scp this to each new instance. RabbitMQ is the broker of all celery workers. And flower is used to monitor all celery workers connected to the RabbitMQ and their status.

2. In the machine named "VM-ENTRY", a celery worker services is deployed and connected to the RabbitMQ server in VM-SPT. Also a copy of raw data is stored when the VM is initialized with cloud-init. A flask server is deployed in this machine as well, as an entry point to the celery service: HTTP requests from clients will be forwarded to celery from this machine and final results summing up will be done here as well.

3. In other machines named "VM-W*", only celery workers will be deployed together with a local raw data duplicate.

This architecture is highly scalable comparing with initial version. Still it can be optimized with more entry points and more nodes to sum all results from workers up. But for this assignment (up to 8 workers), this architecture can work fine. The running status of the system is shown as following.

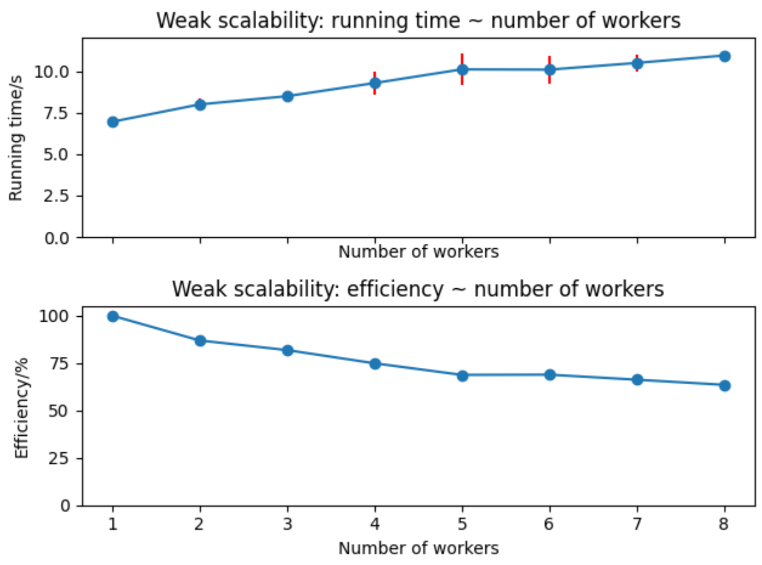


Worker Name	Status	Active	Processed
celery@liju-worker	Offline	0	5151
celery@9993b4210141	Offline	0	0
celery@liju-worker1	Online	0	1558
celery@liju-worker2	Online	0	1040
celery@liju-worker3	Online	0	865
celery@liju-worker4	Online	0	635
celery@liju-worker5	Online	0	519
celery@liju-worker6	Online	0	185
celery@liju-worker7	Online	0	118
celery@liju-worker8	Online	0	55

Showing 1 to 10 of 10 entries

Automation & Portability To improve the automation and portability of the system, all services are deployed based on Docker containers and all VMs can be automatically deployed with cloud init scripts without logging in any machines. The scripts used to create and initializing VMs can be found here, and dockerized contextualization scripts can be found here.

Analysis of Scalability Weak scalability of the system is analyzed, increasing number of celery workers from 1 to 8. Workload of each worker is fixed at 12 raw json files. Because the size of json files vary much, to ensure the reliability of the test, 12 random files will be selected for each machine and 5 tests are done for each case. To test the time consuming of the entire work, we access the service remotely, and time command of linux is used: which is able to present time consuming of each access in real life best.
The results are plotted and shown as following.



As we can see from the figure, with the increase of number of workers, the running time increases from 7s to 11s, and the efficiency of each worker decreases from 100% to 60% correspondingly. This may cause by:

1. communication time cost between workers and brokers: communication time cost increases with the increase of number of workers
2. scheduling time cost of celery: with more overall workload and more workers, celery costs more time on scheduling tasks as well
3. the work is not 100% parallel handled: the summing up step of the procedure is done on only one machine: with more and more workload, this may become the bottleneck of the performance.