Distributed Systems - Assignment 1

NAME: Maor Rocky **ID:** 203246079
NAME: Yoan Shimoni **ID:** 317756609

Instructions on how to run our project and how the program works:

We used the instance: ami-076515f20540e6e0b , type: T2_SMALL

- First, the Credentials stored in the ./aws directory in the Local machine
- Next, we assume that the Local App receives 3 or 4 arguments:
  - Absolute path of the input file
  - Output file name to create
  - Number of operations per worker (which will be divided by the actual number of Workers).
- Terminate Manager after the job done (Optional)
- Then the Local App:
  - Creates a new ec2 instance if the Manager instance isn't exist yet (clarified by the unique tag.value - "Manager")
  - Creates a new bucket with a unique name, and upload the input file to this bucket.
  - Creates 2 new queues "L2M" and "Manager2Summary", if they're not exist yet, to communicate with the Manager - Send "New Tasks" and receives "Done Task".
  - Makes a new Task that represents the job and includes all the details needed by the Manager to manage it
  - Sends the Task values to the Manager through the queue with the help of json
  - Waits for a message from the Manager informs that the job was finished and download the output file from the bucket.
- The Manager runs immediately by the bootstrap script and do:
  - Creates 2 new queues, to Workers and from Workers.
  - Builds 5 thread pools for all the tasks: Download, Distribution, NodesCreation, ConvertPDF, Files.

  This way we support the Scalability principle, and the system works efficiently with multi-threading and designed to handle a growing amount of work. By dividing the work that has to be done to different batches of threads that takecare of the same mission, the system can concurrently and efficiently serves a lot of different Local App and one Manager instance. If we wish to handle a very large number of clients, we will have to choose better resources - a stronger type of instances, and large number of threads, but the system would handle that.
- Every Worker starts immediately to operate and convert the specific number of pdf files that the Manager told him, informs the Manager when the job is done and upload the result file to it's unique bucket.

We have tested our project with different attributes to check that everything works correctly with a few different ways: large and small input file, different number of Workers, concurrently with multi-threading and different number of threads per different missions. Despite many obstacles and bug fixes, the application run well. Nevertheless, due to Amazon's limitations on Educate users, the number of instance built at the same time is blocked and more then 10 is not allowed, and the number of threads running by an instance is limited too and throw an exception. We conclude that concurrently running works fine and more efficiently at least for small number of clients and operations, but we're disabled to check it on practice for very large number of clients and based on the theory.

Specific Questions

- how much time it took your program to finish working on the input files, and what was the n you used

We assign n to be 50 and the input file include 2500 operations, but due to the limitation of Amazon Educate we allow to run no more than 10 instances concurrently. The program took **4 minutes** to complete since the Local App has started to run.

- What about persistence? What if a node dies? What if a node stalls for a while? Have you taken care of all possible outcomes in the system? Think of more possible issues that might arise from failures. What did you do to solve it? What about broken communications? Be sure to handle all fail-cases!

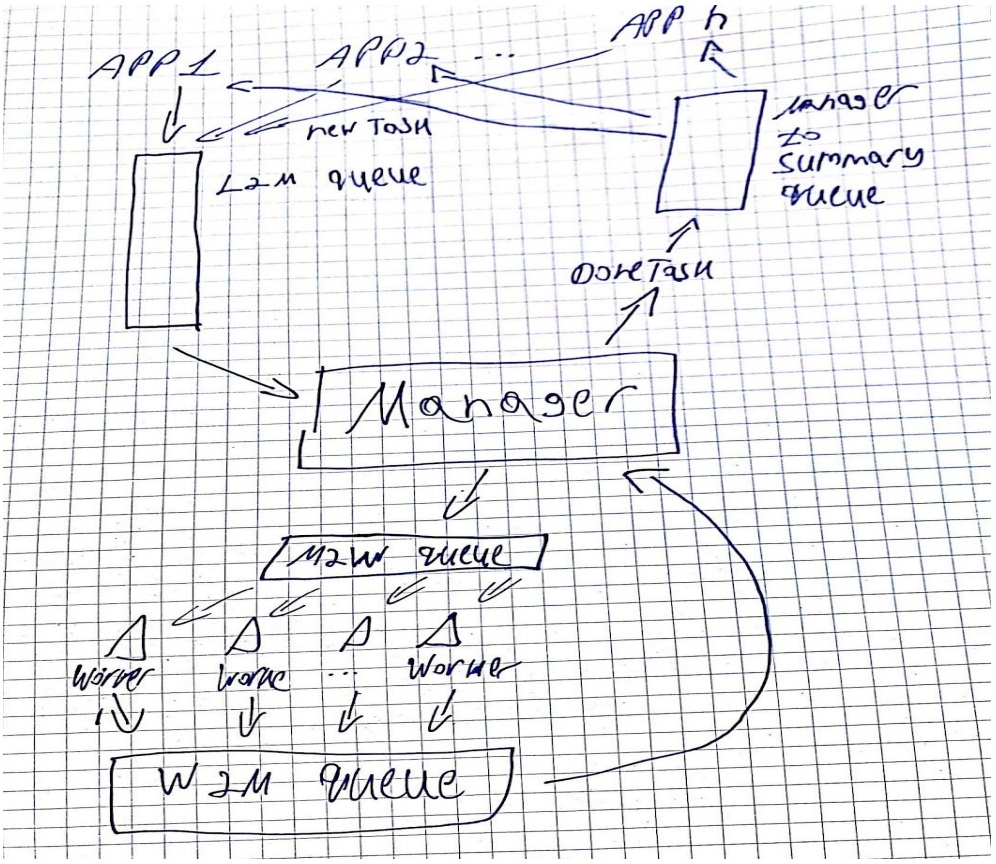
After receiving a message from the queue, a Worker will start working on the job, and only when it complete an Acknowledge message will be sent (method belongs to the SQS API). This is part of the Message Visibility as mentioned in the class. The Task-Message stays in the queue and right after a Worker took the Message, no other Worker can take it for 30 second, and then if it's completed other Worker can take it from the queue. so, if a node stalls or dies, no Acknowledge message was send and other Worker can take the Message and complete the task.

- Threads in your application, when is it a good idea? When is it bad? Invest time to think about threads in your application!

As we learned, Threads are useful and makes the application to run efficiently and faster, and serves the Scalability property. We used Multi-Threads for almost every part of the program, excluded the connection to a new EC2Client.

- Do you understand how the system works? Do a full run using pen and paper, draw the different parts and the communication that happens between them.

*Attached file:



- Did you manage the termination process? Be sure all is closed once requested!

The Termination process depend on the Local App arguments. If the Local App runs given 4 arguments, then the Manager will terminate all the Workers and Terminates himself at the end. If only 3 arguments were given, the Manager terminates all the Workers, but continue to run.

- Are all your workers working hard? Or some are slacking? Why?

In our application, we use “Queue Connection”, so all the messages are asynchronous, and every Worker can handle all each operation needed. This way all the work is divided between all the workers and no Worker is slacking.

- Is your manager doing more work than he's supposed to? Have you made sure each part of your system has properly defined tasks? Did you mix their tasks? Don't!

The Manager makes the minimum job needed - Send new tasks to Workers and send the answers received back to the Local App.

- Lastly, are you sure you understand what distributed means? Is there anything in your system awaiting another?

The Manager isn’t blocked in any part of the process. While the Workers work on the tasks, the Manager can still get messages from different Local Apps used by different queues. This way we support the distributive property of the application.

Questions about visibility and queues:

- What is in-flight-mode?

When a consumer receives a message from an SQS queue the message remains in the queue until the consumer deletes it from the queue. Because Amazon SQS is a distributed system there is no guarantee that the consumer actually received the message hence the message remains in the queue until the consumer deletes it. To prevent other consumers from processing that message again Amazon SQS sets a visibility timeout, when a message is consumed by consumer and he didn’t delete it yet, the message is in flight mode.

You have 2 apps connecting to the manager, both sent task requests and are waiting for an answer. The manager uses a single queue to write answers to apps.

The manager posted 2 answers in the queue. How would you play with visibility time to make your program as efficient as possible?

For that case, we decreased the visibility timeout to 5 seconds for this purpose.

- What happens if you delete a message immediately after you take from the queue? Two scenarios: 1) worker takes a message deletes it from the queue processes it and returns an answer. 2) Local app takes a message from the answer queue deletes it and downloads the file.

In the first scenario, if a Worker deletes a message before completing it, if the Worker will crush no other Worker will complete the task, this scenario is problematic. The second scenario may cause problems if there are more than one Local App. For example, if 2 Local Apps are running- A and B, and A deletes a messages which was subjected to B, then B won't be able to download the file which was supposed to be his.

Questions about memory:

- Lets say the manager saves the result of each message in its memory , is such a solution scalable? How would you solve it? (write to files buffers of reviews once it reaches a certain size)

This solution isn't scalable since the Manager's memory is finite, and if the program sends more messages than the amount of memory the Manager has, then the program will not run properly. Instead of writing only to the memory, the Manager can writes both to the memory and to the disk, using file buffers, and swap between memory and file buffers, by a certain criteria, for example: the size of the result until now.

- Lets say the manager opens a thread for each local applications, advantages and disadvantages? (advantages: faster, disadvantages: not scalable) (possible improvement)

The main advantage is application that run faster if there are more than one Local App. The disadvantage is that if the Manager instance is not strong enough, it could be crush will running.