Names & Ids:

Bar Hadad - 318182987.

Elad Man – 304858186.

**Run program instructions:**

*java -jar LocalApp.jar [input\_file\_path] [output\_file\_path] N [terminate?]*

input file path – path to input file (including extension)

output file path – path to output file will be written (without extension)

N – number of messages to be processed per worker

Terminate – true or false whether shut-down manager and workers instances and delete all queues

Important: must have aws credentials under ~/.aws/credentials to run

**Types of instance we used:**

**Image-Id:**

ami-076515f20540e6e0b

**Instance Type:**

T2\_MICRO

Running examples:

2500 pdfs processed in 8 min where N was ~357 (i.e 7 workers)

Java – jar LocalApp.jar input-sample-1.txt output-sample-1 357 true

10,800 pdfs processed in 20 min where N was 1350 (i.e 8 workers)

Java – jar LocalApp.jar input-sample-2.txt output-sample-2 1350 true

2 local apps simultaneously – each with 100 pdfs - worked

Note: Input files can be found in the zip folder we submitted

* Security:

We used Amazon Ami-Role in order to give our soon to be launch instances the security credentials in a safe way.

Our bucket is private and cannot be accessed from outside

* Scalability:

We used thread pool in our manager implementation to make it scalable. Our program does not depend on some final number of clients, there is no reasonable reason for our code to crush on 1 million / 1 billion clients (it is a good idea to move to a cluster of several managers to be even more scaled)

* Persistence:

Each time we call aws service we are handling all possible outcomes, i.e. all relevant exceptions that might raise.

In case of communication issue - for example worker reading from a queue which is not up yet it will retry to recover indefinitely (with 1 sec sleep).

In case worker instance shut-down unexpectedly the message it took will be returned to the tasksQ.

In case of delay of some worker we have configured the Q visibility timeout to be longer

* Threads:

Bad idea: when number of clients (local apps), exceeds some number (about several thousands) we should give more power to the manager in the form of Thread pool or even several managers

Good idea: We don’t make new local apps requests wait

We synchronized the activation of new EC2 instances to ensure no more than 10 instances will be running, regardless the number of concurrently running local apps.

* Full run:

***Local app:***

For every local app we generate some unique id.

1. Local app gets an input file and upload it to a private s3 bucket to folder with the name which equal to the id of the local app (The file key will be “inputFile” + localAppId)
2. Local app checks if a manager is active and if not, starts it.
3. Local app sends a message to the manager in the “Local\_Manager\_Queue”
4. Local app waits for its result from its unique Q with the name “Manager\_Local\_Queue” + local app id.

***Manager:***

1. The manager reads messages from Local\_Manager\_Queue until it gets a “terminate” message.
2. ***(loops endlessly loops endlessly to receive messages from Local\_Manager\_Queue)***
   1. If it gets a task message
      1. extracts the numOfPdfForWorker, local app id from the message
      2. asks the thread pool to spawn a new thread run to process the local app message.
   2. (terminate case detailed later)

Each managerRunner thread create the worker tasks, TasksQ and the tasksResultQ (with the name “TaskResultQ” + localAppid). Then launch number of workers as needed, delegate messages to them (in the TasksQueue), wait for all the tasks results to finish and make and upload a summary file to the Local app in the (Manager\_Local\_Q + id) Queue.

***Worker: (loops endlessly to receive messages from TasksQ)***

Each worker read messages from the TasksQueue until the manager shut them down.

* For each message, the worker download the pdf, performed the requested operation, upload the resulting output to s3 and sent a message in the (“TasksResultsQ + localAppId”). In a case of an error, the final message contains the error description.
* Termination Process:

When the manager gets its terminate message, it deletes all the queues and terminate all running workers and itself at last.

* System Limitation:

The manager has some limit on the number of threads it can run.

Our educate amazon account is limited in the number of allowed buckets and EC2 instances so we are have at most 10 EC2 instances (validated in our code)

* Workers Job:

All the workers sharing the same queue (Named “TasksQueue”), so there is no reasonable reason for some worker to work harder than others.

* Project structure:

We split the code into 4 modules: LocalApp, Manager, Worker and utils.

Each 1of the first 3 we mentioned contains a main class running only its relevant logic.

In addition, each module has its own specific dependencies defined in its own pom.xml file.

Any shared functionality like using S3, EC2 and SQS is taken place in the utils module so could can efficiently be reused cross modules.

We also added slf4j with simple logger implementation as our log provider.

* Distribution**:** the system is indeed distributed as we divide big amounts of work into multiple machines which work independently and asynchronously**.**

There are several threads that wait for another process to finish. E.g. local app actively waits for the summary file message and manger waits for all workers to complete their tasks.