***Assignment1***

**Distributed System Programming Cloud Computing**



**By:**

Amir Tsur 203795828

Kobi Eini 201553245

**Main System Components**

**System Block Diargram**

**\*all the flows happen in parallel.**

**Local App:**

**Manager:**

**Worker**

**Detailed explanation of the system**

1) Local-App

Our implantation starts to work with the Local-App.

Every local app creates an AWS EC2, SQS, S3 instances to manage the AWS services it uses.

When a local App begins, it uses it's ec2 to check if a manager is running and starts one if needed. The local App also checks for all the queues and buckets in the system and builds them on demand. If the local App builds a new bucket for the jar files, it uploads the jar files as well.

After raising all the resources, the local app uploads its input files to the input bucket, sends "new task" massages, for each input file it uploads, to the APP\_MANAGER\_QUEUE and awaits for the "done task" massages to arrive to the MANAGER\_APP\_DONE\_QUEUE.

Periodically checks whether the manager is alive and if it has fallen reinitiates one – and resends it the message for the input file.

When receives "done task" massage, the local app downloads the summery file relevant to the input file from the TASK\_DONE\_BUCKET and creates an html file for that summery file.

After creating all html files, if it has been asked to do so through the receive function arguments the local app will send the manager a "terminate" message.

Arguments to initiate a proper local application:

1. args[0] – Full path to input file directory.
2. args[1] - Full path to JAR files directory.
3. args[2] - Full path to desired output files directory.
4. args[3] – Natural number that resembles the number of tasks that each worker will handle (total number of workers is limited to 19 due to Amazon free use terms).
5. args[4] – String – terminate for terminating manager and instances after job is done, else does nothing.

2) Manager

An examination of the system shows that the manager is the bottleneck of the operation.

Therefore, we used a fixed size Thread Pool Executor with 10 threads, for operation to be performed quicker.

The main thread creates the thread pool and the AWS classes and afterwards waits for a message to be received in the APP\_MANAGER\_TASK\_QUEUE.

While it waiting it checks that no worker has fallen and if one or more workers has fallen reinitiates them - total number of workers is limited to 19 due to Amazon free use terms.

When a message goes into the queue the main thread puts it on the THREAD\_TASK\_QUEUE and goes back to waiting.

The main thread will go on until a terminate message received from a locale up and them terminates the thread pool and closes all the EC2 instances created.

Each thread in the thread pool receives the following runnable task: parse the input file into different messages and send them to the MANAGER\_WORKER\_TASK\_QUEUE while counting how many messages were sent.

Afterwards, waits until the same amount of "done PDF task" or "error message", relating to that input file, arrives on the MANAGER\_WORKER\_DONE\_QUEUE.

Every time such message arrives, the thread appends it to the summery text file, which it uploads to the MANAGER\_APP\_DONE\_QUEUE at the end (when the workers returned all the needed tasks in the input file as “done PDF task” or “Error Message”).

The thread will then also send a "done task" message to the MANAGER\_APP\_DONE\_QUEUE, signaling the relevant local App the input file sent was fully dealt with.

3) Worker

The worker code runs on each worker instance opened by the manager.

At the beginning each worker creates an AWS SQS and S3 to use the required AWS services.

Afterwards, the worker starts a loop in which it pulls a message from the MANAGER\_WORKER\_TASK\_QUEUE.

If it fails, the worker goes to sleep for 5 milliseconds and tries again.

If the worker succeeds in pulling a massage it performs the following:

1. parses the message as the following:
   1. parsedMessage[0] = localAppID.
   2. arsedMessage[1] = terminate - true/false.
   3. parsedMessage[2] = n - number of workers.
   4. parsedMessage[3] = uploadedFileURL - input file URL in S3.
   5. parsedMessage[4] = numOfTask - how many input files the localApp uploaded – unused (we presume we will have only 1 input file per application, if not it may support several input files but will not distinguish between each file).
2. Saves massage's receipt handler.
3. Set a static Boolean named DONE to false.
4. Saves the massage's next attributes: command to execute (html/text/png), the URL of the PDF to convert and the locale ID of the locale App witch uploaded the input file.
5. Uses a function called processMSG which does the follows:
6. Download the PDF file to the instance
7. According to " command to execute", uses the PDF tools to convert the file to the requested type.
8. Deletes the original file.
9. Uploads the converted file to MANAGER\_WORKER\_DONE\_TASK\_QUEUE.
10. Returns the URL of the converted file in the bucket.
11. Sends a "done PDF task" to the MANAGER\_WORKER\_DONE\_QUEUE, including the command to execute, the original URL of the PDF to convert and the converted file URL.
12. Changes DONE to true
13. Deletes the massage from the MANAGER\_WORKER\_TASK\_QUEUE.
14. If ever an error occurred during one of the above steps ( you can tell using the DONE Boolean) the worker does as follows:
15. the worker catches the except
16. Send an error massage, containing the original URL, the command to execute and the error explanation to the MANAGER\_WORKER\_DONE\_QUEUE.
17. Deletes the massage from the MANAGER\_WORKER\_TASK\_QUEUE.

The worker will continue this way until it is terminated by the manager.

**Appendix**

**Security:**

We designed the system to keep credentials only in local app – which means that the local app is the only one that creates the resources using the “.withCredentials(credentialsProvider)” statement.

The manager and the Workers have no knowledge over the credentials, instead they are provided with pre-defined names for queues and storages.

Furthermore, the workers haven’t been granted an administrator privileges, only the manager has full access to SQS, S3 and EC2.

**Scalability:**

We can support several clients however due to system limitations (number of workers allowed, number of cores allowed to be used for each computer and synchronization issues) we recognize a bottleneck in some places:

1. Manger – due to synchronization issues only the main thread opens new workers.
2. Manger – we have 10 threads to deal with all the incoming workers messages – it takes a lot of time to deal with large input files because we have more workers than managers threads – possible solution is to increase the number of threads, or having 2 managers – we require further coding for synchronization issues.
3. Workers – we are limited to 19 workers only (20 instances in total and one of them is the manager instance) – takes a lot of time to deal with many tasks – optional solution – create thread pool in workers too (in our computers we have only 1 core, so this will not increase performance).