

Assignment 1: Text Analysis in the Cloud

Due Date: 30/4

Submit your questions to [Moshe](#)

Abstract

In this assignment you will code a real-world application to distributively process a list of text files, analyze them in various levels, upload the analyzed texts to S3, and generate an output HTML file.

More Details

The application is composed of a local application and instances running on the Amazon cloud. The application will get as an input a text file containing a list of URLs of text files with the required type of the text analysis. Then, instances will be launched in AWS (workers). Each worker will download text files, perform the requested analysis, and upload the analyzed text to S3, and generate an output HTML file.

The use-case is as follows:

1. User starts the application and supplies as input a file with URLs of PDF files together with the required text analysis, an integer n stating how many files a single worker can analyze, and an optional argument terminate, if received the local application sends a terminate message to the Manager.
2. User gets back an output html file.

Input Files

Each line in the input file will contain the required type of analysis by a tab ("\t") and a URL of a text file. Three types of analysis are defined:

POS – part-of-speech tagging.

CONSTITUENCY - constituency parsing (context-free representation).

DEPENDENCY - dependency parsing.

In the assignment's folder there is an example of an input file.

Output File

The output is an [HTML file](#) containing a line for each input line. The format of each line is as follows: <analysis type>: <input file> <output file>, where:

- **Analysis type:** POS | CONSTITUENCY | DEPENDENCY
- **Input file** is a link to the input text file.
- **Output file** is a link to the output analyzed file.

If an exception occurs while analyzing a text file, or the text file is not available, then output line for this file will be: <analysis type>: <input file> <a short description of the exception>.

System Architecture

The system is composed of 3 elements:

- Local application

- Manager
- Workers

The elements communicate with each other using queues (SQS) and storage (S3). It is up to you to decide how many queues to use and how to split the jobs among the workers, but, and you will be graded accordingly, your system should strive to work in **parallel**. It should be as **efficient** as possible in terms of time and money, and **scalable**.

Local Application

The application resides on a local (non-cloud) machine. Once started, it reads the input file from the user, and:

- Checks if a Manager node is active on the EC2 cloud. If it is not, the application will start the manager node.
- Uploads the file to S3.
- Sends a message to an SQS queue, stating the location of the file on S3
- Checks an SQS queue for a message indicating the process is done and the response (the summary file) is available on S3.
- Creates an html output file representing the results.
- In case of *terminate* mode (as defined by the command-line argument), sends a termination message to the Manager.

IMPORTANT: There might be more than one than one local application running at the same time.

The Manager

The manager process resides on an EC2 node. It checks a special SQS queue for messages from local applications. Once it receives a message it:

- If the message is that of a new task it:
 - Downloads the input file from S3.
 - Creates an SQS message for each URL in the input file together with the operation that should be performed on it.
 - Checks the SQS message count and starts Worker processes (nodes) accordingly.
 - The manager should create a worker for every n messages, if there are no running workers.
 - If there are k active workers, and the new job requires m workers, then the manager should create $m-k$ new workers, if possible.
 - **For any case don't run more than 19 instances – AWS uses to block students who try this.**
 - Note that while the manager creates a node for each n messages, it does not delegate messages to specific nodes. All worker nodes take their messages from the same SQS queue; it might be the case that with $2n$ messages, hence two worker nodes, one node processed $n+(n/2)$ messages, while the other processed only $n/2$.
- If the message is a termination message, then the manager:
 - Does not accept any more input files from local applications.
 - Waits for all the workers to finish their job, and then terminates them.

- Creates response messages for the jobs, if needed.
- Terminates.

IMPORTANT: the manager must process requests from local applications simultaneously; meaning, it must not handle each request at a time, but rather work on all requests in parallel.

The Workers

A worker process resides on an EC2 node. Its life cycle is as follows:

Repeatedly:

- Gets a message from an SQS queue.
- Downloads the text file indicated in the message.
- Performs the requested analysis on the file.
- Uploads the resulting analysis file to S3.
- Puts a message in an SQS queue indicating the original URL of the input file, the S3 url of the analyzed file, and the type of the performed analysis.
- Removes the processed message from the SQS queue.

IMPORTANT:

- If an exception occurs, the worker should recover from it, send a message to the manager of the input message that caused the exception together with a short description of the exception, and continue working on the next message.
- If a worker stops working unexpectedly before finishing its work on a message, some other worker should be able to handle that message.

The Queues and Messages

As described above, SQS queues are used for:

- communication between the local application and the manager.
- communication between the manager and the workers.

Specifically, we will have the following messages:

- new task message from the application to the manager (location of an input file).
- new analysis task message from the manager to the workers (URL of a specific text file and the required analysis).
- done message from a worker to the manager (S3 location of the analysis file, the type of analysis, and the URL of the input file).
- done message from the manager to the application (S3 location of the analysis summary file).

It is up to you to decide how many queues you want (you can have different queues for different tasks, or one queue, whatever you find most convenient). Be ready to explain your choices.

Running the Application

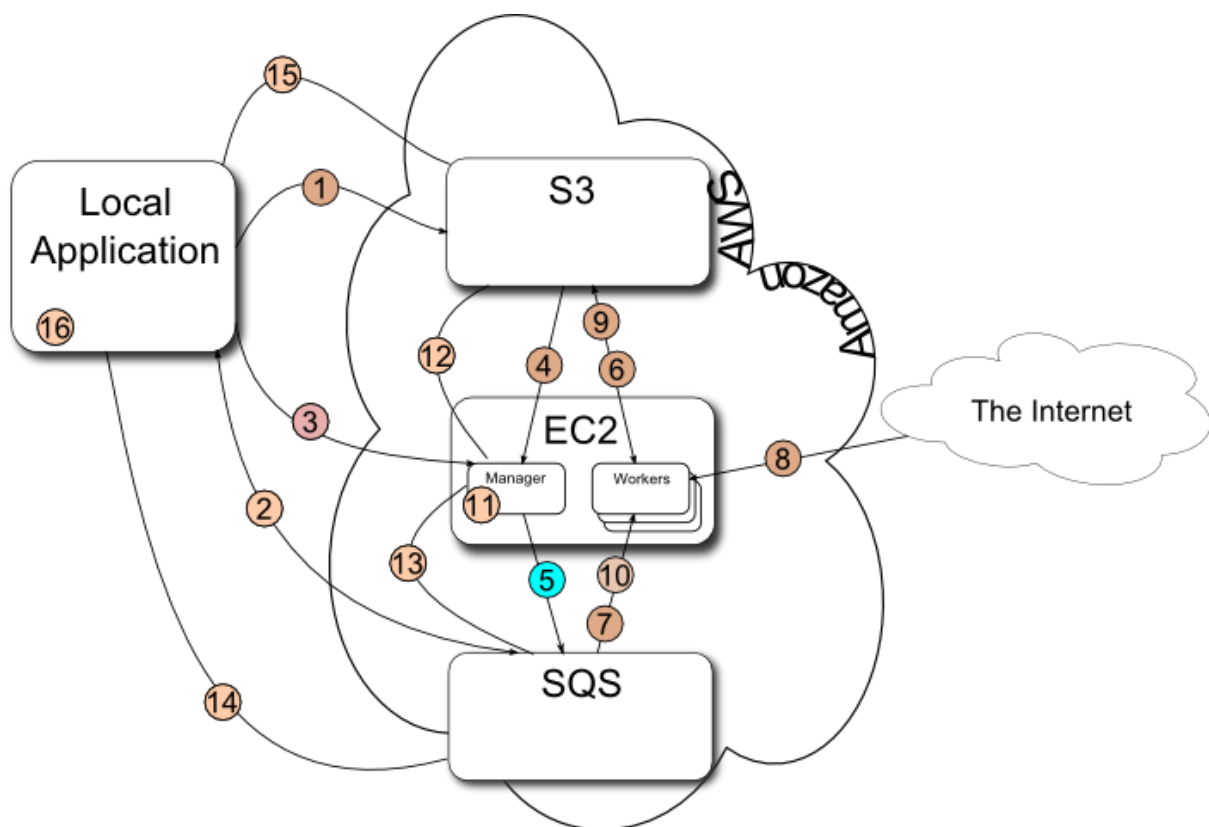
The application should be run as follows:

```
>java -jar yourjar.jar inputFileNames outputFileNames n [terminate]
```

where:

- *yourjar.jar* is the name of the jar file containing your code (do not include the libraries in it when you create it).
- *inputFileName* is the name of the input file.
- *outputFileName* is the name of the output html file.
- *n* is the workers' files ratio (max files per worker).
- *terminate* indicates that the application should terminate the manager at the end.

System Summary



12. Manager uploads the summary file to S3.
13. Manager posts an SQS message about the summary file.
14. Local Application reads SQS message.
15. Local Application downloads the summary file from S3.
16. Local Application creates html output file.
17. Local application send a terminate message to the manager if it received *terminate* as one of its arguments.

Getting Started

- Read the assignment description.
- Read the reading the material, and make sure you understand it.
- Create a [maven](#) project.
- Write the code that analyzes a given text file (based on the Stanford code, as follows), run it on your computer and make sure it works.
- Write the local application code and make sure it works.
- Write the manager code, run it on your computer and make sure it works.
- Run the manager, the local application, together with a worker on your computer and make sure they work.
- Run the local application on your computer, and let it start and run the manager on EC2, and let the manager start and run the workers on EC2.

Technical Stuff

- Getting started with AWS:
 - Read the guide in the AWS academy lab - all you need to know is presented there, and is updated to the latest features and libraries of AWS.
 - Make sure that you place the .pem (credentials) file in the right location, and where your AWS CLI expects it to be -- this is up to you to make sure you do it correctly. Otherwise, you won't be able to communicate with the AWS services.
 - Be aware of AWS regions - make sure that the time of the region you're using (the region of the instances) is consistent with your system (the region of your aws console, the region of the AWS dashboard). Since the session key of AWS academy is only valid for several hours, large time differences can mean that while your access keys are indeed 'fresh', it is not recognized by AWS.
This [link](#) may be useful.

- AMI Image

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine. In our case: EC2 instance. An image of a virtual machine is (in simple words) a copy of the VM, which may contain an OS, data files, and applications (just like your personal computer) - in our case: Amazon Machine image (AMI).

You need an AMI composed of any data needed for running the Manager and the Workers (basically, Linux, Java JDK 1.8, Amazon AWS cli, the jar file of the Manager and the Worker classes, and the Stanford-Parser jar).

You can create a new AMI from the [Amazon EC2 Instances view](#), or from the console of an EC2 instance:

Downloading from EC2 console: In Linux, the command `wget` is usually installed. You can use it to download web files from the shell.

Example:

`wget https://www.gutenberg.org/files/1659/1659-0.txt -O a.txt`

will download the content at <https://www.gutenberg.org/files/1659/1659-0.txt> and save it to a file named `a.txt`. [wget man](#)

Installing from EC2 console: In Ubuntu (or Debian) Linux, you can use the `apt-get` command (assuming you have root access to the machine). Example: `apt-get install wget` will install the `wget` command if it is not installed. You can use it to install Java, or other packages. [apt-get man](#).

In case AWS disable the option for creating new AMI, which seems to be the case right now, use the user-data script for downloading the required jars each time the node is initialized.

- The AWS SDK

The assignment will be written in Java, you'll need the [SDK for Java](#) for it. We advise you to read the [Getting Started guide](#) for getting comfortable with the SDK.

AWS Maven dependencies:

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>ec2</artifactId>
  <version>2.17.155</version>
</dependency>

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>s3</artifactId>
  <version>2.17.155</version>
</dependency>

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>sqs</artifactId>
  <version>2.17.155</version>
</dependency>
```

- SQS Visibility Time-out

Read the following regarding SQS timeout, understand it, and use it in your implementation:

[Visibility Timeout](#)

- Bootstrapping

When you create and boot up an EC2 node, it turns on with a specified image, and that's it. You need to load it with software and run the software in order for it to do something useful. We refer to this as "bootstrapping" the machine.

As taught in class, the bootstrapping process of a new node is based on [user data scripts](#), and

another [guide](#). User-data allows you to pass a shell-script to the machine, to be run when it starts up. Notice that the script you're passing should be encoded to base64. Here's a [code example](#) of how to do that.

Your user-data scripts can be written in any language you want (e.g. Python, Perl, tsch, bash). bash is a very common choice. Your scripts are going to be very simple. Nonetheless, you might find these [bash tutorials](#) useful.

- Checking if a Manager Node is Active

You can check if the manager is running by listing all your running instances, and checking if one of them is a manager. Use the "tags" feature of Amazon EC2 API to mark a specific instance as one running a Manager: [using tags](#), [CreateTagsRequest API](#)

- Third-party libraries

To analyze the given texts, we will use the [Stanford Parser](#) tool. Add the following dependencies to the Worker pom.xml file:

```
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-parser</artifactId>
  <version>3.6.0</version>
</dependency>
```

In case the Stanford parser find it hard to analyze the whole text file, feel free to parse the file line-by-line (writing the output of each line to the output file one by one)

Grading

- The assignment will be graded in a frontal setting.
- All information mentioned in the assignment description, or learnt in class is mandatory for the assignment.
- You will be reduced points for not reading the relevant reading material, not implementing the recommendations mentioned there, and not understanding them.
- Students belonging to the same group will not necessarily receive the same grade.
- All the requirements in the assignment will be checked, any missing functionality will cause a point reduction. Any additional functionality will compensate for lost points. You have the "freedom" to choose how to implement things that are not precisely defined in the assignment.

Notes

Cloud services are cheap but not free. Even if they were free, waste is bad. Therefore, please keep in mind that:

- It should be possible for you to easily remove all the things you put on S3. You can do that by putting them in a specified bucket under a folder, which you could delete later.
- While it is the Manager's job to turn off all the Worker nodes, do verify yourself that all the nodes really did shut down, and turn of the manger manually if it is still running.
- You won't be able to download the files unless you make them public.
- You may assume there will not be any race conditions; conditions were 2 local applications are trying to start a manger at the same time etc.

A Very Important Note about Security

As part of your application, you will have a machine on the cloud contacting an amazon service (SQS and S3). For the communication to be successful, the machine will have to supply the service with a security credentials (password) to authenticate. Security credentials is sensitive data – if someone gets it, they can use it to use amazon services for free (from your budget). You need to take good care to store the credentials in a secure manner. One way of doing that is by compressing the jar files with a password.

Submission

Submit a zip file, contains:

- all sources and class files, without the libraries that you're supposed to download;
- the output of running your system on the sample file.
- a text file called README with instructions on how to run your project, and an explanation of how your program works. It will help you remember your implementation. Your README must also contain what type of instance you used (ami + type:micro/small/large...), how much time it took your program to finish working on the input files, and what was the n you used.

Mandatory Requirements

- Be sure to submit a README file. Does it contain all the requested information? If you miss any part, you will lose points. Don't forget your names and ids...
- Did you think for more than 2 minutes about security? Do not send your credentials in plain text!
- Did you think about scalability? Will your program work properly when 1 million clients connected at the same time? How about 2 million? 1 billion? Scalability is very important aspect of the system, be sure it is scalable!
- 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!
- Threads in your application, when is it a good idea? When is it bad? Invest time to think about threads in your application!
- Did you run more than one client at the same time? Be sure they work properly, and finish properly, and your results are correct.
- 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.
- Did you manage the termination process? Be sure all is closed once requested!
- Did you take in mind the system limitations that we are using? Be sure to use it to its fullest!
- Are all your workers working hard? Or some are slacking? Why?
- 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!
- Lastly, are you sure you understand what distributed means? Is there anything in your system awaiting another?

All of this need to be explained properly and added to your README file. In addition to the requirements above.