HW4 – Distributed Election

Excepted Effort: 20 hours

Objectives

- Master at least one distributed election algorithm
- Become more familiar with the concept of resources managers
- Become more familiar tools for provisioning collections of resources that are needed for a distributed system

Overview

In this assignment, you will implement a distributed system with dozens of processes that share common data files and cooperate to compute edit distances for pairs of strings. Your distributed system will run on AWS EC2 instances and use S3 storage.

System Description

Each process will be able to play one of two possible roles: resource manager or worker. At any point in time only one of processes may be a resource manager. As a resource manager, a process must be able handle the following kinds of requests from other processes:

- Get a pair of strings for which the edit distance needs to be computed
- Save the edit distance for a specific pair of strings

A work process is responsible for computing the edit distance of string pairs, one pair at a time. It should know of the current resource manager's end point and make the above requests to the resource manager to get a pair and to save the results. The request should be reliable therefore it needs to follow at least a Request-Reply communication pattern. You can use web-services, which has this built in or you can do your UDP- or TCP-based communications.

The string-pair data are the shared resources in this system. To manage the sharing and to track which pairs have been processed, the resource manager will need to save its state information in some way on S3. That way, if the current resource manager goes down, another can pick up where it left off and guarantee that no pair gets skipped. Depending on how you design your resource manager, some work may have to be redone and that's okay. But, you don't want your system to have to re-do too much work when recovery from a failure and you can't allow any pair to get missed.

The resource manage should also be able to detect if a worker is no longer alive and re-assign the pair that it was given to compute the edit distance for.

Each process should also able to detect the failure of the resource manager, start a distributed election, and participate in elections started by other processes.

The input data files will be ASCII text files where each line contains a pair of strings separated by a comma. The strings will only contain letters and numbers. The strings may be very long. Your system should take as a command-line parameter an input directory and then consider every file in that directory as an input file.

Your system should take as another command-line parameter an output directory, into which it will save the results of the workers' computations. Each line in an output file should contain the original pair, followed by a comma and their edit distance. After your system finishes running the total number of lines in your output files should be the same as the total number of lines in the input files.

Instructions

Step 1 – Reuse / Adapt your Name Service Component from HW2

The processes in this system, like the disease-tracking system, will need to get the communication end-points of other processes in the system. You should be able to adapt your name service from HW2 for this purpose.

Step 2 – Design, Build, and Test Process Id Assignment Functionality

Figure out how you want to assign process id's, and then implement and test that functionality. Some ideas including a) building that into your application launching mechanism, b) extending your name service to provide an id upon registration, or c) creating a separate process for that purpose.

Step 3 – Design, Build, and Test your Resource Manager Functionality

Design your resource manager and the communications protocol for worker-to-manager interactions. Think about how it will save state information, including:

- what input file it is currently being processed
- The current read position in that file
- Current assignments being computed by workers

Then, implement and test your resource manager independent of any workers or running in a distributed environment.

Step 4 – Design, Build, and Test a Client API for the Resource Manager

Design and build an API that workers can use to communicate with the current resource manager. Test it locally.

Step 5 – Design, Build, and Test Worker Functionality

Look for an implementation of an edit-distance algorithm that you can adapt. There are some out there for Javascript and Node.js. Design and implement your worker functionality. Test it independent of any communications with the resource manager.

Then, build the logic that communicates with the resource manager and does work until none is available. Include in the implementation of the communication logic the raising of a "Resource Manager Failed" event if the resource manager is none responsive. That event could trigger the starting of an election. Test all the worker functionality locally, before testing in the distributed environment.

Step 6 – Design, Build, and Test the Distributed Election Functionality

Design and build all aspects of the election functionality, including the ability to start and participate in elections. Test locally before testing in the distributed environment.

Step 7 – Final System Testing

Run in the distributed environment with anywhere between 2 – 99 workers running on 1 – 100 instances. Randomly stop processes, including the current resource manager and verify that the system correctly completes all the work. Assume that your name service remains functional and accessible through any whole run.

Zip up your root directory and submit to Canvas.

Grading Criteria

	Points
	=====
Success complete of Step 1	20
Success complete of Step 2	20
Success complete of Step 3	20
Success complete of Step 4	20
Success complete of Step 5	20
Success complete of Step 6	30
Success complete of Step 7	20
Total	150