CSC 536 Spring 2025

Due June 11th 2025 11:59pm

Class Project

1. A general, scalable, distributed Map-Reduce framework

You will use the material from Lecture 3 and build on Homework 3 to develop a distributed and elastic MapReduce app that uses Akka Cluster and that can be used to solve a range of problems rather than just one specific problem. Your app should take a map function and a reduce function, whether dynamically or statically, and should also take a set of input key/value pairs as input and produce the set of output key/value pairs by running a MapReduce job using the map and reduce functions. Your framework should be able to scale out and use a number of Akka Cluster nodes running mappers and reducers.

Because many MapReduce type jobs use external resources (files, databases, resources on the Internet) you should make sure that faults are reasonably handled so the whole MapReduce job doesn't crash.

You should demonstrate the generality of your framework by running three different jobs:

1. counting the number of occurrences of words in a set of text files (the example used in Lecture 3)
2. computing the number of incoming hyperlinks for each html file in a set of html files (the first step in computing its PageRank)

If you need more info on MapReduce than provided in the textbook or Lecture 3, see the original paper on MapReduce available at http://research.google.com/archive/mapreduce.html. The last chapter of my *Introduction to Computing using Python* textbook also has a simple implementation, in Python, of a general Map-Reduce framework.

If you are really fast and ambitious, you can port your application to the cloud by 1) using Docker so every node in your cluster runs in its own Docker container and 2) implementing a bootstrapping and service (i.e. seed node) discovery using etcd or similar. You may use alternative tools to achieve the desired result.

2. An(other) Akka implementation of Raft

The subject line says it all. Your goal will be to implement the core parts of the Raft Consensus algorithm.

3. A better GroupService

You will take your GroupService with multicasting implementation from Homework 5 and improve it by making it fully-distributed across multiple machines/JVMs and adding one or both of the following features:

- Refactor the code to use Akka Cluster
- Provide some type of reasonable message ordering guarantees and ensure well defined group membership (i.e., joining and leaving) protocols so actors do not receive messages meant for groups they are no longer in.

4. A high-performance, distributed ticket sale application

You will develop a system for selling tickets for events. Each event has a date and venue and each venue will have a limited number of seats. You may assume that all seats are the same. Each ticket will be associated with a date and a venue. Checking that there are seats still available is the obvious bottleneck if the app is to handle many clients concurrently. In order to get around the bottleneck, you will create a set of "kiosk" actors that will be given small chunks of tickets for each event (date) from a master actor. Clients would connect to a kiosk to purchase tickets for a particular event.

When a kiosk runs out of tickets, it should request another chunk from the master. If the master has no more chunks for a particular event and not other kiosk does either then the event is sold out and the clients requesting tickets should be told so. If the master has no more chunks for a particular event but another kiosk does, then we need a way to transfer tickets from one kiosk to another. To do this, the master would have to keep track of ticket sales at each kiosk which makes the master a bottleneck in the application (i.e. we're back to the original problem). To solve this problem, you will avoid having the master communicate directly with all kiosk actors and organize the communication

architecture for the master and the kiosk actors into a message-passing ring. The master will send an event-related message to its clockwise neighbor (the first kiosk) who will handle the message and send (typically a different) message to the second kiosk, and so on until the master receives the event-related message from the last kiosk.

The initial message for a particular event should contain a number of tickets that is equal to the chunk size times the number of kiosks. When a kiosk receives this message, it will take its chunk of tickets and forward a message to the next kiosk in which the number of tickets has been decremented by chunk size. After receiving back the message (and possibly sleeping for a few seconds) the master actor will send another message with the next range of tickets. Kiosks will take a chunk only if they have run out of tickets. (To avoid the complexity of handling order for multiple tickets, you may assume a client is buying only one seat at a time.)

Once the master has run out of tickets, it will continue sending messages along the ring for a particular event until the event is sold out, but it will not be adding any new tickets. These messages will be used by kiosks to exchange tickets. There are multiple ways for the kiosks to do this; I will let you choose an approach and you will need to argue its benefits in your project writeup (see below).

Since kiosk are doing I/O, they are prone to failure and you should try to implement some fault-tolerance mechanism that insures no ticket is lost.

5. Implement transactions with 2-Phase Commit and 2-Phase Locking

You will extend the KVStore and KVClient discussed in Lecture 5 to implement serializable ACID transactions using two-phase locking (2PL) and two-phase commit (2PC). Each transaction executes within a single application actor that is a client of KVStore (e.g., within a GenericServer). So transactions are serial within each client actor, but multiple client actors may execute transactions concurrently.

We continue to assume that KVStore actors do not fail. Given that assumption, the **D** property is easy: no changes to the storage scheme are required to make transactions "durable". Also, this assumption makes 2PC easier, because it is unnecessary to deal with failure of a transaction participant.

Your transaction implementation should use the client-side caches implemented in KVClient, and it must keep the caches consistent.

Here are a few points to consider as you design your implementation.

- You will want to add a begin/commit/abort API to KVClient, and also add unique transaction IDs and commit/abort messages to the KVStore messaging interface.
- Given these extensions, it will be simplest to handle locking and conflict detection at the granularity of individual keys within the KVStore servers. If a transaction T1 requests a get/put for a key K, and then another transaction T2 requests a conflicting get or put for K before T1 completes (commits or aborts), your KVStore actors can and should detect this case and handle it.
- If you handle this conflicting access scenario by blocking T2 until T1 completes, then you must address the problem of deadlock. For example, if T1 requests K1 and T2 requests K2, and then T1 requests K2 and T2 requests K1, neither T1 nor T2 can make progress. Note: it is not required or expected that you will implement deadlock detection, although it is not so difficult. It will be sufficient to set a per-transaction timer on the client, and abort if a transaction is stalled for "too long".
- You must address the problem of a failed or partitioned client. If a client loses contact with a server for longer than some fixed timeout, its pending transaction (if there is one) must be forced to abort. Of course, there may be a transaction that has committed, but not all participants have been notified. You may assume that a failed or partitioned client (coordinator) does recover with its state intact: when that happens your client should restart the protocol to complete any actions still in progress.

Implement some tests to demonstrate that your transaction service is correct.

Note: You may find useful to use Akka schedulers to manage sessions (http://doc.akka.io/docs/akka/current/scheduler.html). The scheduler allows you to schedule a message to be sent to an actor at a specified time or after a specified delay.

6. A silly but compute intensive problem!

You will develop a scalable application using Akka Cluster to solve the following problem. You are given a list of 42 students. Associated with every student is the student's *number*, the student's *name*, the SHA-256 *hash* of the student's password, and the student's RNA gene *sequence*.

Your application will need to solve the following sequence of subproblems:

1. Crack the passwords: Knowing that each password consists of exactly six digits 0-9, find the six digit password for each hash using brute force.
2. Compare the gene sequences: For each student i, find among the other students the student (partner) who shares the longest common gene subsequence with student i by comparing RNA sequences pairwise. (Possible algorithms for finding the longest common subsequence include: dynamic programming, apriori-gen, suffix tree, etc.)
3. Solve a linear Diophantine equation: Given the 42 integer passwords p_1, p_2, p_3, ... p_{42} you found in step 1, find a_1, a_2, a_3, ... a_{42} such that $a_1*p_1 + a_2*p_2 + a_3*p_3 + ... + a_{42} * p_{42} = 0$ where each $a_i = 1$ or -1.
4. Mine the hash: For each student i, repeatedly hash the student's partner number (modified each time by adding a nonce, which is a random integer) until you obtain a hash that starts with 11111 if a_i is 1 or with 00000 if a_i is -1.

Your application should have one master and support an arbitrary number of worker nodes (ActorSystems) using Akka Cluster. After starting all the workers nodes, the master should be given the problem input contained in this CSV file: inputs.csv. The output should go to the console and include the execution total running time. After that your application should terminate cleanly.

7. You have a cool idea

Run it by me so I make sure it fits the scope of the course and the assignment.

What you will submit

When you have finished the assignment as described above, you will submit:

1. Your code.
2. A file (or files) containing sample output generated by running your program. Readable screenshots with short explanations are great.
3. A separate document describing 1) how to run the project, 2) the overall system design, 3) the benefits of various design decisions you have made as well as the tradeoffs, and 4) the synchronization, replication, and fault tolerance aspects---if applicable---of your app.
4. The directories containing all the executable files and other test directories you used to test your code (so I can easily reproduce your executions)

Archive and compress all the above using zip and submit the archive using the assignment submission tool on d2l. I hope you enjoy your project.