Brian Bauman

CSC 536

Final Project Description

**How to run the project**

1. This program has been set up to use 3 remote servers to host the mappers and reducers. To add more servers, you must add more addresses in the master.scala file. The servers must be running and accepting traffic at that port in order to function.
2. The number of mappers and reducers running in total can be changed in the relevant application.config files.
3. The client code currently supports three behaviors, as defined in the assignment. To select which, simply uncomment out and recommend the appropriate sections of the client.scala code. Alternatively, you can do multiple in a row.
4. Once those decisions have been made, running the system is trivial.

(1) First, start up each server by navigating to the “server#” directory included (or additional that were created) and perform “sbt run” from a terminal or command line.

(2) Once all servers are running, start up the client program by navigating to the “client” directory and performing “sbt run”

(3) The client code will generate the requested data and forward it to mappers and reducers that will then pass the responses back to the client machine and print out the appropriate results.

**The overall design**

The goal of this design was to create a system that could configure a standard MapReduce infrastructure with the ability to quickly change out the behavior of the mapping and reducing functions. To do this, the Mapper and Reducer actors have been generalized and only expect to speak in terms of generic parameters. As a part of their initial configuration, they are informed of their expected behavior and load in the appropriate map, hashMap, reduce, and aggregate functions to use as data arrives.

When they are finished processing, they pass the results on to the next actor in the chain (or back to the client), still in a generic form.

The general structure of the design is as follows:

1. The client.scala program determines which work is to be done and creates a MasterActor (master.scala) that will be responsible for setting up the appropriate mapper and reducer actor pools and routers. This class is generic and will be created to match the expected input and provided map()/reduce()/aggregate()/etc. functions.
2. MapperInput messages that match to the work that will be performed are also created and sent to the Master actor, who passes them on (generically) to the MapperRouter and ReducerRouter.
3. The client program waits five seconds and alerts the Master actor that it is finished sending input.
4. The Master actor sets up the Mapper and Reducer servers and sends them a configuration message. This message (passed initially by the client) informs them of their jobs. In response, they load up the appropriate functions to use on the incoming data.
5. Each MapperInput message arrives at a GenericMapper and is parsed according to the loaded function. The transformed results are sent via consistent hashing to the GenericReducer
6. The master actor forwards the Finished message to the Mappers who broadcast it to the Reducers after finishing their mappings.
7. The GenericReducer aggregates the data as it comes in. When it hears that all Mappers are Finished, it takes the aggregated data and performs the previously loaded reduce() function on this data, generating a final result.
8. This final result is returned generically to the Master actor. This actor uses a loaded printResults() function specific to the application to display the results as necessary.
9. The program ends on the client’s side when the Master has received and displayed all results from the Reducer actors

**Various design decisions**

If another function was desired, the only changes that would need to be made are to define the map, reduce, aggregate, and hashMapping functions in a static object file (such as hyperlinks.scala provided), and direct the mappers and reducers to these files when they receive the associated activity request in their initial configuration. As can be seen, the benefit of this design is that it’s extremely easy to scale to new functionality without having to rewrite the majority of the framework. The common code that remains unchanged by differing requested function can be modified easily to add additional servers to the pool or to increase the number of mapper and reducer threads that are active.

A tradeoff is the fact that you must edit GenericMapper and GenericReducer code, but only barely. Definitely saves the majority of the effort — creating the infrastructure.

In future iterations, the ability to specify more of the inputs int he config file would be nice. Also, the ability to create a user front end and interactively select the options would make the entire experience more straightforward, though this is definitely out of the scope of the current project.

**Distributed Systems considerations**

Synchronization

One interesting aspect about this (and most) MapReduce apps is that the individual Mappers and Reducers do not need to synchronize at all. They are designed to be working in parallel and have no need to communicate with each other. None of their inbound messages must be processed in any strict, global order, either, which renders most synchronization topics unimportant.

Replication

Similarly, there is not much replication that needs to be done most scenarios. In other iterations, you could include ability to send mapping and reducing inputs to a cluster of multiple nodes. Each of these nodes would process the input identically and, assuming no rogue nodes or network problems, should agree on an output. This would then be forwarded to the next layer, etc. In my current system, a single rogue node will be able to propagate any data that it chooses to.

Fault Tolerance

The aspect of fault tolerance is certainly where this project is most vulnerable. Because the data is being evenly distributed to each of the active mappers, the reducers must wait on all of them to finish processing the data and successfully forward a Finished message before they can begin their reduction. Were any of those mappers to go down, the reducers would never start processing and the application would timeout. As discussed in the Replication section, additional mappers and reducers in a cluster could prevent this, or reduce its effect. Similarly, if a reducer were to go down, its data would never be returned to the client. Depending on the implementation, this may mean that no data (or partial data) is returned.

**Overall**

This was a very interesting project. I thought that it was nice to see that the majority of my time was spent determining the best way to generalize the code that worked in Homework 3. After that was working, adding the suggested use cases was very simple and require little testing. The few things that did need to be changed were bugs with the introduced functionality, not anything with the pipeline of the Mapper/Reducer. I suppose that fact is what makes this a useful and interesting project.

Also, I’d never used generics in Scala before, so that required a bit of Googling. I think there are possibly better ways to go about this that can decouple the GenericMapper and GenericReducer slightly more from the “plugins”, but this more simple implementation does get the job done.