Project 3: Map Reduce Facility

Michael Wang - mhw1
William Maynes - wmaynes@andrew.cmu.edu

Section 1 - Design

Introduction

We wanted to provide a light weight, easy to use Map Reduce framework. As a means to this end, we also built a light, but flexible Distributed File System (DFS) built around a custom but highly-applicable file type.

The Distributed File System (DFS)

The Distributed File System has two components, the Coordinator and the Participant. These two parts are supported by a custom Object, the MRFile, which I will discuss later. The Coordinator is responsible for distributing files across the network and keeping track of which participants have which distributed files. The Participants are responsible for writing files to their local space and retreiving files from both their local space and in the network.

Using just the above functionality, the DFS is capable of partitioning files into custom chunk sizes, replicating these chunks, and then spreading these chunks across the nodes in the system. By breaking files into chunks and replicating those chunks across the network, the system reduces queue times on resources and allows the Map Reduce Facility faster access to working data than single copies would allow.

The system is also capable of the seamless lookup of files in the system by name, whether or not the file is on the local host. If a Participant doesn't have a file that it wants, it contacts the coordinator to find out who does and requests the new file directly from one of them. It is important to note that the entire DFS has a flat namespace, so writing files to the same name or distributing files with the same name is undefined behavior.

An important piece of the System is the MRFile - short for Map-Reduce File. The MRFile is designed around the ArrayList Object for a number of reasons. First, ArrayLists are serializable and are easily sent through network communications and even through file communication. Second, they allow arbitrary accesses of data, meaning that users can access random locations in the File. Finally, although the DFS does enforce a fixed chunk size while distributing files, intermediate files and initial files using the MRFile format can have arbitrary sizes. This means that the file is easily partitioned into chunks and that workers can write local files of arbitrary size without having to worry about chunk violations.

The Map-Reduce Facility (MRF)

The Map-Reduce Facility has three components the Scheduler, the Master, and the Worker. The Master is incharge of managing and getting both jobs to do and new worker nodes, the Scheduler is in charge of taking the work that is given to the master and distributing it amongs the Workers, and the Workers are in charge of actually carrying out the work. It's important to note that work can be added at the workers or at the master, but the workers just simply pass on their inputs to the master to be handled.

The Master is designed to be able to add Workers to its workforce, add tasks for those workers to do, and to track progress on those tasks by name. This is primarily done through Remote Method Invocation, using Java's build in RMI Framework, and having workers update teh master on their tasks as they complete them.

With the information that the Master tracks, the Scheduler then takes work and schedules it across worker nodes. In order to do this, it uses a simple scheduleling mechanism to try to keep the number of tasks assigned to workers across the system roughly equal.

Although we allow work and tasks to be added from the workers, their primary role is to actually perform the tasks that they are assigned by the scheduler and then update the Master when they are done with their work. There are two parts to these tasks. The first is the Map Task and the second is the Reduce Task. The Map Task is the task which takes input data from the file system and does something with that data. The Reduce task is the task which combines the output of all workers that are currently collaborating on this task. Our framework is built around tasks that have large and intensive map processes, so that we can distribute the map work across many workers. The reduce task can then simply combine the results after the map portion of the work has been completed.

Combining the DFS and MRF

The MRF is the primary workhorse for our framework. As workers receive tasks, they pull the files relevant to their task from the DFS using the DFS Participant that lives with them. They then can both write the results of their computations locally to the machine and retreive those results later using the DFS as well. The only thing to note is that the files which the MRF uses must be MRFiles. This requirement is does restrict format, but it still allows quite a bit of power in the representation of the information, since MRFiles can represent arbitrary length collections of objects.

Section 2 - Administration and Configuration

The Configuration File

Our framework requires a configuration file be given to the DFS Coordinator. This file is passed as a path writen to Standard In. The Framework will then parse the necessary information out of the config file.

The config file contains all information necessary for the DFS to be initialized and run. It must maintain a simple $< KEY >=< VALUE >$ format. We require the following keys: factor, chunksize, initdata, dfsmaster, and registry. Also, for each DFS Participant that you want to have, the name of that participant (which is used for in the registry and tracking files) must map to the location of the participant, where the location has the format "host:port". The required "dfsmaster" and "registry" keys must also map to the locattions that the DFS Master and Registry live on, respectively. All location values follow the same format.

Here is a simple example config file:

```
factor=2
chunksize=10
initdata=data
dfsmaster=localhost:15500
registry=localhost:15650
part1=localhost:1554
part2=localhost:1555}
```

Starting the Workers and DFS Participants

Once you have created your config file, you must then start your initial set of workers and dfs participants. This means running WorkerServer on each machine that you want to start a worker and participant on. It will take two arguments from standard in. The first is the port for the DFS Participant to listen for the DFS Coordinator on (this should match the port given in the config file for each particular participant). The second is the name of the worker for the task manager.

Starting the Master, Scheduler, and DFS Coordinator

Once all of your workers and participants have been started, you need to start your management tools. This simply means running MasterServer and providing the path to the config file (which you hopefully wrote previously) on standard in when prompted. The framework will take care of the rest.

Watching Over an Active System

After the MasterServer has been started, you will have a small number of administrative options to actively watch over the system. You can poll your workers to see who are still alive and you can check what files are actively in your DFS at any given moment. To poll workers, simply type "poll" into the prompt of the MasterServer. Viewing files in the DFS can be done by typing "files" into the MasterServer prompt.

Adding Tasks to the Framework

To add a task for the Framework to perform you just need to follow the following steps:

1. Get a RemoteReferenceObject (RRO) for any Worker/Master
   (It may help to note that the Master is mapped to "taskmaster" in the Registry)
2. Create a MapReduceTask Object.
   This Object will embody all aspects of the task, including the map and the reduction after the map is complete.
3. Call addTask on the RRO with the MapReduceTask Object you created.

The task will then be dispatched and scheduled by the Scheduler.

Section 3 - Programming for the Framework

In order to program a task for the framework, you simply need to create a MapReduceTask which implements MRTask. To do this, you need to create a MapCallable which is the code for the Map portion of your task and a ReduceCallable, which is the reduce portion of your task.

MapCallable

Creating a MapCallable is done by extending the AbstractMapCallable, which takes an input Object type U and converts it to an Object of type T. All you need to do is return the transformed data, whatever the transformation is. It is important to note that the map function that you, as the programmer, implement will be called on each entry in the MRFile that the current worker has. Thus, if you only want to transform specific sets of data, you should set your types so that you receive your arguments accordingly.

ReduceCallable

Create an object that implements ReduceCallable. You will receive an ArrayList of the results of the MapCallables as an argument, in order. Thus, the first entry will be the result of the first map, the second result will be the result of the second map, and so on. You, as the programmer, simply need to combine the elements of this ArrayList as you see fit for your return type T.

Section 4 - Demonstrations

We have two demonstrations. One that is a simple demonstration of the basic features of the MRF, particularly reducing the results, partitioning results, and replicating them. The other is a demonstration of the powerful parallelism that can be achieved be our framework.

Math and Numbers

Demonstration one simply takes a file that has the numbers 0-199, adds one to each line, and then sums the result. Using small chunk sizes and large replication factors can easily demonstrate how

well the DFS distributes files across the Participants if easy access. Also, by checking the results of the computation you can easily verify that the framework reduces results correctly, since the result is easily mathematically verifiable (it should be 20100).

Image Manipulation

Demonstration two shows the massive power of parallelism by inverting the colors of an image. It will take an RGB image in the correct format and invert it by splitting the inversion process across the framework. Notice that manipulating large images can take quite a while, especially when manually manipulating individual pixels. The framework allows the picture to be broken apart into chunks and manually manipulate pixels across the workers in parallel. This demonstration is purely to show parallelism. (Note that network overhead actually makes the speed of this demonstration on reasonable image sizes less practical than we would like.)

Section 5 - Requirements Met, Missed, and Exceptional Mentions

---

Failures

The only requirement that the Framework lacks at the moment is a way for dropped nodes to rejoin the network. That being said, this could be easily fixed with the following changes: When a node receives a ConfigInfo object, it writes that object to the disk. Any time a node starts, it first checks to see if it has a valid ConfigInfo object on disk, and if it does it boots from that object and uses the information contained in it to contact the DFS Master. The DFS Master then needs to have a method which allows it to add a new DFS Participant, which is really just a ConfigInfo object with the new participant's name, host, and port. Also, we would have to change the WorkerServer object to allow for two kinds of startups, either joining a Framework in progress or a new Framework. Either way, these changes are not significant and could be done with a bit more time.

Alternatively, once the downed nodes have been restored and the current active task has been completed, the system can be brought down and reinitialized to continue work, which works just as well.

Requirements Met

Other than the miss mentioned above (which was noticed just too late), we met all of the requirements, as far as we are aware.

Special Mentions

Although it is not implemente as such, the DFS could very quickly be changed to a ring structure instead of relying upon a coordinator. This is because of the design of the request function and the fact that it is invokable remotely. Thus, if a worker needed a file, it could request the file from its local participant which could simply ask the next participant in the ring, which could ask the

participant after it, and so on and so forth until the file was found, at which point the methods would all simply return.

Also, the MapReduce facility allows for dynamic typing of Tasks even on a single instance of the framework by inheriting types and objects from the Map and Reduce Callables. That's pretty cool.