15-640: Distributed Systems

Assignment #3: Map-Reduce Engine

Ankit Agarwal: *ankitaga@andrew.cmu.edu*
Iosef Kaver Oreamuno: *ikaveror@andrew.cmu.edu*

11/22/2014

0. Architecture

There are 8 main components in our Map Reduce (MR) system:

1. MR-Master

This is the main component of our Map Reduce system. The MR master is responsible for listening for create-job requests and making sure that all of the mappers and reducers are successfully completed for the job submitted by the user. It works closely with the Scheduler, DFS master and the Node managers to make sure that everything runs smoothly. Whenever a create-job request comes, the MR-Master (or job manager) asks the scheduler to schedule the mappers and reducers of the job. Also, it keeps track of the state of the mappers and reducers currently working on each job. In case a mapper or reducer fails, it asks the scheduler to schedule a new one, or terminates the job if the mappers and reducers have failed too many times.

2. DFS-Master

The DFS master is the main component of the distributed file system that works along the map reduce system. Currently, the DFS lives alongside the MR-Master, but the current design allows the DFS to be easily decoupled from the MR system. The DFS master is responsible for listening for write requests. Whenever a client requests the DFS master to write a file, the DFS saves the file in several data nodes (the number of nodes is given by the replication factor parameter) and keeps track of which node has each file. Whenever a data node goes down, the DFS master replicates all of the files that were on the node to the other data nodes currently on the system. Additionally, the DFS master also responds for "name" requests, basically, it tells the clients on which node can they find the file that they're looking for.

3. MR-Node Manager

MR-Node Manager manages a single node of the MR-cluster. The responsibilities of the MR-Node manager are:

- Management of mapper JVMs: Each map reduce can run require several mapper tasks. One or more mapper tasks can be assigned to a node manager. A node manager, on request from mr master starts up a new mapper JVM, assigns it relevant work, polls for the current state of the work and terminates the jvm when the map task finishes or fails.

- Management of reducer JVMs: Similarly, a map reduce can involve multiple reduce tasks. One or more reduce tasks can be assigned to a node manager. A node manager, on request from mr master startups up a new reducer JVM, assigns it relevant work, polls for the current state and terminates it when the reducer JVM has successfully written it's output to the distributed file system.

- Management and streaming of locally generated data: Data generated by a map task is sorted and stored on the local file system. This data needs to be streamed to the reducer. A reducer instance can talk directly to the relevant node manager (which has the data generated by a map task) for obtaining the data.

- Provide scheduling information/status updates: The node manager, on request by the master provides status of a mapper/reducer task which was assigned to it at an earlier point in time. A scheduler can talk to a node manager for obtaining a picture of how "heavily" loaded a node is before making scheduling decisions.

4. DFS-Data Node

The data node is a component of the DFS module of the system. The data node simply stores files that the DFS master requests him to store. Additionally, it listens for read requests of other nodes. Whenever a read request comes, the data node streams data to the node that made the request.

5. Mapper

A mapper represents an attempt at running a map task on an input chunk. Each mapper is run as a separate JVM and is managed by a node manager. It is booted with enough information so that it can communicate with the distributed file system. A node manager communicates with a mapper instance via RMI.

A mapper instance maintains a state variable whose value can be either RUNNING or FAILED or FINISHED. The input to the mapper task is a partition of the input file on which it has to act upon. The meta data required for obtaining partition is supplied by the node manager and the list of nodes where the actual partition exists is obtained by communicating with the DFS-Master. The mapper then talks to the closest node which has the required input chunk and downloads it. It then loops through input record of the chunk obtained and passes it along to user supplied map function. The output of a mapper task is a single file on the local file system. Since both node manager and mapper run on a same instance, they share the same local file system. A mapper task is considered to be successful, when the node manager managing it obtains (file) path to it’s output. A mapper task can be terminated at any point of time by the node manager.

6. Reducer

A reducer represents an attempt at running a reduce task. Each reducer is run as a separate JVM and is managed by a node manager. It is booted with enough information so that it can communicate with the distributed file system. Similar to mapper, a node manager communicates with a reducer instance via RMI.

A mapper instance maintains a state variable whose value can be either RUNNING or FAILED or FINISHED. The input to a reducer task is a set of node managers from which it needs to pull it’s data. The reducer talks to each of the node managers (*from the above set*) and attempts to pull the key, value pairs (generated by various mapper tasks) on which it needs to act upon. The key, values pairs thus obtained are then merged and supplied to the user defined reduce function. The output is collected and is written to the DFS. Once the output is successfully written to the DFS, the reducer is said to be successful (i.e. the state variable is updated to FINISHED).

At any point of time, if a reducer encounters an exception (either while pulling data from the node managers/or while running user defined reduce function etc.), the reducer is said to have failed. In such a scenario, the reducer will be terminated by the node manager and will be rescheduled (*possibly on another node*) by the mr-master (after talking to the scheduler).

7. Scheduler

The scheduler listens for requests of the MR Master to schedule mapper or reducer jobs. Whenever a new request comes, the scheduler selects the node manager more appropriate to run the job and sends over the job request to him. The scheduler tries to assign the job to node managers that already have the data locally, and that don't have too many jobs running already.

8. MR Client

The client project is simply allows and end user to access the MR framework. For instance, it allows the user to upload files to the DFS, submit jobs to the MR master, get the current list of jobs running and get the list of jobs that have finished.

1. Design Choices:

- Master-Slave Architecture:

The inherent architecture of the system is master-slave. Hence, the machine running mr-master is the single point of failure of the system. Even though the machines are unreliable and a master-slave architecture may seem like a bad choice, in real world, we can invest additional hardware resources to make sure that mr master is more reliable. Also, master slave architecture reduces system (and code) complexity and is easier to maintain.

- Jobs can be launched from anywhere:
  An executable jar for submitting job from any machine has been provided. The machine need not be a part of the cluster. The IP address of master along with the port needs to provided as command line arguments to the this jar.

- Indirect communication between mr-master and mapper/reducer instances:
  In our current implementation, any communication between master and mapper/reducer instance goes via node manager. We chose to go with this approach since master no longer needs to do any additional bookkeeping to keep the the map/reduce remote object around. Also, it delegates responsibilities of handling communication failures etc. with the node manager. Finally, the communication overhead of indirect communication is very low since the nodemanager and mappers/reducers it manages, run on the same machine.

- Push vs pull mode of communication between different entities
  Specifically, we chose to use polling to for various tasks (like fetching the current state of the mapper task, fetching the current state of reducer task etc.) over a push based mechanism. One of the primary reason for taking this approach was that the entities which were supposed to make the "*push*" could have died. A hybrid approach probably works better but it was decided that investing time in introducing this additional complexity wasn't worth the gain.

- Each Map/Reduce instance runs in a separate JVM:
  An easier choice would have been to run each of the mapper/reduce instances in a separate thread. However, since a user supplied code could be buggy, malicious(what if it makes a *System.exit(1)* call), it was decided that introducing this additional level of complexity is essential.

- DFS datanode is built along with the MR node manager
  In our current implementation, a dfs data node is built along with the mr node manager binary. However, the current design allows for easy decoupling of mr node manager and dfs data node services. Similarly, dfs master is built along with MR master node.

- Communication over RMI rather than Sockets:
  While websockets are good for dealing with low level communications, RMI provide a abstraction for dealing with complex interactions among machines, since they allow different entities to interact with each other as objects. If we were to use websockets, a significant coding time would have to be spent designing a reliable (and error free) protocol for handling communications between machines.

- Framework is written in Java:

For starters, when you are dealing with a medium scale architectural complex project, there aren't many option to start with. Between C++ and Java, the choice was Java since both the authors are more familiar with it. Personally, we would have loved to give the project a shot in Golang but considering time constraints, we didn't follow through.

---

2. Project structure

For this assignment, we decided to create 7 different Java projects.

- Shared library project:
    - Located in **PROJECT_ROOT/src/common**
    - This library contains code that is shared between all of the projects. It also contains the example Mappers and reducers we have implemented for testing the project.
- MR-master project:
    - Located in **PROJECT_ROOT/src/mrmaster**
    - Contains all the code related to the Map Reduce master node. Includes the job manager (responsible of creating jobs and terminating them), job tracker and the scheduler.
- MR-client project:
    - Located in **PROJECT_ROOT/src/mrclient**
    - Contains all the code related to clients that want to use the MR system. For example, it allows clients to see which jobs are currently running and which jobs have finished. It also allows them to upload and download files to the DFS and to create new jobs for the MR system.
- MR-nodemanager project:
    - Located in **PROJECT_ROOT/src/mrnodemanager**
    - Contains all the code related to the node managers. These are responsible for managing mapper and reducer instances local to a node. They are able to launch and terminate these instances, as well as query them for their state.
- MR-dfs project:
    - Located in **PROJECT_ROOT/src/mrdfs**
    - Contains all the code related to the distributed file system.
- MR-map project:
    - Located in **PROJECT_ROOT/src/mrmap**
    - Contains all the code related to running a "map" task attempt. Specifically, this project contains code for running a new jvm for each new mapper task.
- MR-reduce project:
    - Located in **PROJECT_ROOT/src/mrreduce**
    - Contains all the code related to running a "reduce" task attempt. Specifically, this project contains code for running a new jvm for each new reducer task.

All of the projects have the same structure: A src subdirectory and a target subdirectory. The src subdirectory contains all of the source files and the target subdirectory contains the generated jars, .class files and maven metadata files.

For more details about how to deploy the system, see the **How to run, deploy and build the project** section.

---

3. System administrator guide

System requirements

- Python 2.6.6+ if you want to use the automated deploy script.
- Java 6.
- UNIX system.

How to build the project

If you wish to compile the code on a GHC machine, you can simply run the **ghcbuildall.sh** script in the project directory:

sh ghcbuildall.sh

This script runs some *maven* commands to download dependencies and build the project. maven is already installed on the ghc machines and all dependencies are downloaded and packaged automatically. It also builds the python paramiko module which is required by the setup script. If you do not want to download the dependencies, you can use the ready to run jars that are located in the src/<module>/target directory.

The ghcbuildall.sh script compiles the code and generates the following executable jars which could be found by in the following directories (relative to the project root):

- **MR Master:** mrmaster/target/mrmaster-1.0-SNAPSHOT-jar-with-dependencies.jar
- **MR Node manager:** mrnodemanager/target/mrnodemanager-1.0-SNAPSHOT-jar-with-dependencies.jar
- **MR Mapper:** mrmap/target/mrmap-1.0-SNAPSHOT-jar-with-dependencies.jar
- **MR Reducer:** mrreduce/target/mrreduce-1.0-SNAPSHOT-jar-with-dependencies.jar
- **MR Client:** mrclient/target/mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar

Each of these jars correspond to one of the architecture components described above. The DFS and the common binaries are included as libraries to these projects.

How to deploy the project

First, you must create a system configuration file. We provide you with an example one in config/system-config.txt . It is a JSON file with a few parameters. Here's an example:

```
{
    "config": {
        "master-host": "ghc48.ghc.andrew.cmu.edu",
        "master-port": 4917,
        "participants": [
            {
                "ip": "ghc33.ghc.andrew.cmu.edu",
                "port": 8932
            },
            {
                "ip": "ghc32.ghc.andrew.cmu.edu",
                "port": 4321
            },
            {
                "ip": "ghc31.ghc.andrew.cmu.edu",
                "port": 7854
            },
            {
                "ip": "ghc30.ghc.andrew.cmu.edu",
                "port": 4372
            },
            {
                "ip": "ghc29.ghc.andrew.cmu.edu",
                "port": 8432
            }
        ],
        "max-retries-before-job-failure": 20,
        "time-to-check-for-job-state": 3,
        "time-to-check-for-nodes-state": 4,
        "time-to-check-for-data-nodes-state": 5,
        "workers-per-node": 6,
        "replication-factor": 3,
        "chunk-size-in-MB": 32,
        "max-dfs-read-retries": 5
    }
}
```

Here's an explanation of all the parameters:

- master-host: The hostname of the master node.
- master-port: The port number in which the master node will listen for requests.
- participants: An array of participant object. Each participant is described by an ip and a port. There are the node managers / data nodes of the system.

- max-retries-before-job-failure: The amount of times the MR master will allow any mapper or reducer to fail before it terminates the job.
- time-to-check-for-job-state: The MR master will query the node managers for the state of each job every (time-to-check-for-job-state) seconds.
- time-to-check-for-nodes-state: The scheduler will query the node managers for their busyness state every (time-to-check-for-nodes-state) seconds.
- time-to-check-for-data-nodes-state: The DFS master will query the data nodes for their state every (time-to-check-for-data-nodes-state) seconds.
- workers-per-node: The amount of mappers and reducers that can be working simultaneously at any node.
- replication-factor: The DFS master will replicate each file of the file system on (replication-factor) data nodes.
- max-dfs-read-retries: The DFS will try to read a file (max-dfs-read-retries) before it returns with an error.

IMPORTANT: Please note that max-retries-before-job-failure is the amount of failures that the MR system will tolerate before terminating a job. This is the total amount of failures (if mapper 1 fails once, mapper 2 fails once, mapper 3 fails once, and the max-retries-before-job-failure parameter was 3, then the job will be terminated after mapper 3 fails). Please make sure of setting this parameter to a high enough value if you want to test that the system still runs properly when nodes are terminated (For instance, if you intend to terminate 5 mappers, then make sure this parameter is at least 6, else the system will terminate the job when the fifth mapper crashes). If you put down entire node manager instances, these may be responsible for

A python script **json_vallidator.py** (under src/scripts/) has been provided to assist with basic sanity checking of the configuration. To invoke the checker, run the command:

```
python json_validator.py --json_file=<path to configuration file>
```

The script either prints a success or failure message along with details. It is important to note that the the configuration checker is nothing more than a basic syntax validator. It doesn't do an exhaustive check for semantics. For instance, if a replication factor which is greater than the total number of participants in the system has been specified then the system may fail to function correctly, while the validation script may pass it.

The final step in the process is to copy the relevant binaries across the cluster and run mr master and node manager services.

A python script **setup.py** (under src/scripts/) has been provided for assisting with the setup of the mr cluster. This script will copy relevant binaries to the mrmaster and participant (node manager) machines. It will also startup each of the instances with the proper arguments, according to the configuration file given. Lastly, it also creates the required directories for the

MR system to work properly (/tmp/ankitaga-ikaveror-mr directory, /tmp/ankitaga-ikaveror-mr-dfs directory and /tmp/ankitaga-ikaveror-mr-local directory). Please use the **setup.py** script to get the system running, else it is too much work to copy the binaries and create the required directories on every machine. You can run the script from any machine.

Here's an exhaustive list of parameters which must be specified to the setup script:

- "master-binary-path": Path where the mr-master binary is located.
- "slave_binary_path": Path where the mrnodemanager binary is located.
- "mr_map_binary_path": Path where the mrmap binary is located.
- "mr_reduce_binary_path": Path where the mrreduce binary is located.
- "config": Path to mr cluster configuration file. This file can be validated using json_validator and is required to have a structure defined above.
- "dfs_base_path": Path on the cluster machines where files on dfs are to be stored.
- "localfs_base_path: Path where any local intermediate files which are generated by the map reduce are to be stored.
- "rwd": rwd is an abbreviation for remote working directory. This is the remote directory where any relevant binaries along with configuration files will be copied to.

Additionally, a few more optional parameters may be specified:-

- "skip_copy": Skips the copying phase of binaries in the cluster.
- "cleanup": Cleans up any files which were created by a previous run of the setup script.
- "kill_all": Kills all instances of mappers, reducers, mrnodemanagers and mrmaster.

A sample invocation of setup script is included in the **README** under src/scripts. Here's an example (you can simply copy this command on the src/scripts directory and the system will be setup, and ready to go, assuming that the system config file system-config.txt exists on the src/scripts directory):

```
python -W ignore setup.py
--master_binary_path=../mrmaster/target/mrmaster-1.0-SNAPSHOT-jar-with-dependencies.jar
--slave_binary_path=../mrnodemanager/target/mrnodemanager-1.0-SNAPSHOT-jar-with-dep
endencies.jar
--mr_map_binary_path=../mrmap/target/mrmap-1.0-SNAPSHOT-jar-with-dependencies.jar
--rwd=/tmp/ankitaga-ikaveror-mr/
--mr_reduce_binary_path=../mrreduce/target/mrreduce-1.0-SNAPSHOT-jar-with-dependencie
s.jar --config=system-config.txt
```

The only thing you need to change to this command is the **--config** parameter. We have copied a system-config.txt on the scripts directory that you can use, in case you simply want to run the command and get everything working.

<u>Note:</u> While running the setup script, andrew id and password are required. The credentials are used to ssh into different machines of the cluster and copy relevant binaries. The credentials are NOT stored in any form.

<u>**4. Application programmer:**</u>

Using the MR system is quite simple once its up and running. First make sure that your system administrator has already set up the MR system properly and that it is up and running. This is a general guide on how to use the MR system, we provide specific examples on the next section.

<u>**Mapper:**</u>

Each user defined mapper should implement IMapper interface and therefore needs to implement the following method:-

public void map(String record, ICollector collector);

For the sake of simplicity, each *record* is fixed length record of characters.

ICollector exposes the following interface:-

public void collect(String key, String value);

To emit a key, value pair from a mapper, call collector.collect(key, value). Multiple invocations of collect can be made for each run of map. For example,

```
public class WordCountMapper implements IMapper {
        public void map(String record, ICollector collector) {
                String cleanRecord = record.trim();
                collector.collect(cleanRecord, "1");
        }
}
```

<u>**Reducer:**</u>

Each user defined reducer should implement IReducer interface and therefore needs to implement the following method:-

public void reduce(ICollector collector, String key, List<String> values);

ICollector exposes the following interface:-

public void collect(String key, String value);

The output to a reducer, similar to mapper is a key, value pair. The key, value pair in the output are separated by a “,” (comma). Multiple invocations of collect can be made for each run of reduce. For example:

```
public class WordCountReducer implements IReducer {
        public void reduce(ICollector collector, String key, List<String> values) {
                collector.collect(key, Integer.toString(values.size()));
        }
}
```

Note: User defined mapper and reducer class should provide a zero argument constructor for constructing a map/reduce instance.

Now, we must create a job configuration file. It is simply a JSON file with a few parameters that we must complete. Here is an example job configuration file:

```
{
        "config" : {
                "job-name" : "MySuperIncredibleGrepJob",
                "bundle-path" : "/Users/YourUser/your/path/to/your/jar/superjar.jar",
                "map-class" : "com.ikaver.aagarwal.hw3.common.examples.GrepMapper",
                "reduce-class" : "com.ikaver.aagarwal.hw3.common.examples.IdentityReducer",
                "input-file-path" : "grep-in",
                "output-file-path" : "grep-out",
                "num-reducers" : 2,
                "record-size" : 21
        }
}
```

Here's an explanation of all the parameters:

"job-name" : The name you want to give to your job. Feel free to be creative.

"bundle-path" : The path to the jar that contains the mapper and reducer classes.

"map-class" : The complete class name of the mapper class that implements the IMapper interface.

"reduce-class" : The complete class name of the reducer class that implements the IReducer interface.

"input-file-path" : The input file path of your MR job. You will upload this file to the DFS before running your MR job. Please, make sure the path does not contain "/" characters.

"output-file-path" : The output file of your MR job. The reducers will save their output to (output-file-path)-REDUCER_NUMBER.out.

"num-reducers" : The amount of reducers you want for your MR job.

"record-size" : The record size in bytes. If your records are separated by new lines, the new line is counted as part as the record size.

Now that you completed the job configuration file, implemented mapper and reducer classes and have your input file, you're ready to submit everything to the MR System!

First, run the MR-client program. You can find the jar in src/mrclient/target/mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar. Now, execute the jar:

java -jar mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar -config <PATH_TO_YOUR_SYSTEM_CONFIG_FILE>

Where, <PATH_TO_YOUR_SYSTEM_CONFIG_FILE> is the path to the config file you used to initialize the system.

Here are all the commands that you can give to the client program:

- terminate <JOBID>
  - Terminates the job with the given job id
- download <REMOTE-FILE-PATH> <LOCAL-DESTINATION-PATH>
  - Downloads the file with path <REMOTE-FILE-PATH> from the DFS and saves it to the local file <LOCAL-DESTINATION-PATH>.
- finished
  - Returns a list of all the jobs that have finished.
- list
  - Returns a list of all the jobs that are currently running, with information of how many mappers and reducers have finished their task.
- create <FILEPATH>
  - Creates the job with job configuration <FILEPATH>.
- upload <INPUT-FILE-PATH> <DESTINATION-PATH> <INPUT-FILE-RECORD-SIZE>
  - Uploads the file <INPUT-FILE-PATH> to the path <DESTINATION-PATH> of the DFS. Each record of the file is of size <INPUT-FILE-RECORD-SIZE>. Please make sure that <DESTINATION-PATH> doesn't include "/" characters.
- shutdown
  - Shutsdown the entire MR system.
- end-session
  - Terminates this client session.
- info <JOBID>
  - Returns information of the job with job id <JOBID>.

To submit a job, simply upload your files with the upload command, and then run the create command with your job configuration. For a specific example see the next section.

5. Examples

We provide you with two examples, a word count and a grep example. We will give you a step by step guide of how to to run each of these examples.

Grep example

Our grep example returns all of the records that contain the string "-DS.FTW-". We will guide you through all of the steps: creating the input file, creating the job config file, uploading the file to the DFS and submitting your job to the MR master. We are assuming you already followed all of the steps in the "System administrator guide", therefore the system should be up and running at this point.

First, we'll create the input file:

We provide you with a python script that creates a random input file. First, go to the grep example directory:

cd examples/grep

Now, run the following command to create the file:

python grep_data_generator.py -p "-DS.FTW-" -r 20 -n 20000000 > grepin.txt

This will create a file with 20 million records of random strings, where approximately 10% of them will contain the string "-DS.FTW-", each separated by a new line (therefore, each record will be of 21 bytes). This takes a few minutes, please wait for a little bit.

We provide you with a config file for the job. You can find it in examples/grep/grep-job.txt. **<u>IMPORTANT</u>:** The only thing that you will have to change is the **bundle path** parameter. You must specify the complete path to the common jar of the project. This jar is located in:

PROJECT-ROOT/src/common/target/common-1.0-SNAPSHOT.jar

```
{
        "config" : {
                "job-name" : "GrepJob",
                "bundle-path" : "PROJECT-ROOT/src/common/target/common-1.0-SNAPSHOT.jar",
                "map-class" : "com.ikaver.aagarwal.hw3.common.examples.GrepMapper",
                "reduce-class" : "com.ikaver.aagarwal.hw3.common.examples.IdentityReducer",
                "input-file-path" : "grepin",
                "output-file-path" : "grepout",
                "num-reducers" : 2,
                "record-size" : 21
        }
}
```

Now that your job config is ready, let's go ahead and submit the job! Run the client program, if it isn't running already.

You can find the jar in src/mrclient/target/mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar. Now, execute the jar:

java -jar mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar -config <PATH_TO_YOUR_SYSTEM_CONFIG_FILE>

Where, <PATH_TO_YOUR_SYSTEM_CONFIG_FILE> is the path to the config file you used to initialize the system (system-config.txt).

Now, upload the file to the DFS, run this command on the client program:

upload <PATH-TO-THE-INPUT-FILE-YOU-CREATED> grepin 21

You should see an output similar to this one:

```
upload ../../../examples/grep/grepin.txt grepin 21
> Will start file upload process...
Uploading chunk 1 of 13...
Uploading chunk 2 of 13...
Uploading chunk 3 of 13...
Uploading chunk 4 of 13...
Uploading chunk 5 of 13...
Uploading chunk 6 of 13...
Uploading chunk 7 of 13...
Uploading chunk 8 of 13...
Uploading chunk 9 of 13...
Uploading chunk 10 of 13...
Uploading chunk 11 of 13...
Uploading chunk 12 of 13...
Uploading chunk 13 of 13...
File uploaded successfully!
```

Next, submit your job to the MR system

create <PATH-TO-THE-JOB-CONFIG-FILE-YOU-CREATED>

You should see an output similar to this one:

```
create ../../../examples/grep/grep-job.txt
> Submitting job to the MR master...
Created job: [(Job ID: 2), (Job name: GrepJob), (Mappers completed: 0), (Reducers completed: 0)]
```

Now you have to wait a little bit until the job finishes. You can issue list commands to see the state of your job:

```
list
> [(Job ID: 2), (Job name: GrepJob), (Mappers completed: 0), (Reducers completed: 0)]
list
> [(Job ID: 2), (Job name: GrepJob), (Mappers completed: 4), (Reducers completed: 0)]
```

```
list
> [(Job ID: 2), (Job name: GrepJob), (Mappers completed: 13), (Reducers completed: 1)]
```

Eventually, you will get this output after running the list command:

```
list
> No jobs currently running...
```

Great! This means that your job finished executing. Now use the "finished" command to see the status:

```
finished
> [ID: 1, Name: GrepJob, Success: true, Output: { grepout-0.out grepout-1.out }]
```

Now you can download the output of the reducers (grepout-0.out, grepout-1.out) by using the download command, for example:

```
download grepout-0.out ./grepout0.txt
> Starting download process, please wait...
File downloaded successfully!
download grepout-1.out  ./grepout1.txt
> Starting download process, please wait…
File downloaded successfully!
```

You should see several lines that contain the string "-DS.FTW-". For instance:

```
AA-DS.FTW-EMJEWIYSQH,
AA-DS.FTW-ETMXFPDVFN,
AA-DS.FTW-EXJKHZPZYM,
AA-DS.FTW-FBCHAYEIKN,
AA-DS.FTW-FCHPEHSVXC,
AA-DS.FTW-FICZSTRLQN,
AA-DS.FTW-FJTKCUIOOZ,
AA-DS.FTW-GFLJPVSVDP,
AA-DS.FTW-GVGIONSNWB,
AA-DS.FTW-GXVIIZJZUJ,
```

Word count example

Our word count example simply counts the times each word appears on the input file. You must follow exactly the same steps than the grep example, the only difference is the way to generate the data:

First, we'll create the input file:

We provide you with a python script that creates a random input file. First, go to the word count example directory:

cd examples/wordcount

Now, run the following command to create the file:

python wordcount_data_generator.py -r 20 -n 5000000 > wcin.txt

This will create a file with 5 million records of random alphanumeric strings, each separated by a new line (therefore, each record will be of 21 bytes).

The next steps are identical to the grep steps, using the word count input files, mapper and reducer classes and word count job config file.

We provide you with a config file for the job. You can find it in examples/wordcount/wc-job.txt. **IMPORTANT:** The only thing that you will have to change is the **bundle path** parameter. You must specify the complete path to the common jar of the project. This jar is located in:

PROJECT-ROOT/src/common/target/common-1.0-SNAPSHOT.jar

```
{
        "config" : {
                "job-name" : "WCJob",
                "bundle-path" : "PROJECT-ROOT/src/common/target/common-1.0-SNAPSHOT.jar",
                "map-class" : "com.ikaver.aagarwal.hw3.common.examples.WordCountMapper",
                "reduce-class" : "com.ikaver.aagarwal.hw3.common.examples.WordCountReducer",
                "input-file-path" : "wc",
                "output-file-path" : "wc",
                "num-reducers" : 2,
                "record-size" : 21
        }
}
```

Now that your job config is ready, let's go ahead and submit the job! Run the client program, if it isn't running already:

You can find the jar in src/mrclient/target/mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar. Now, execute the jar:

java -jar mrclient-1.0-SNAPSHOT-jar-with-dependencies.jar -config <PATH_TO_YOUR_SYSTEM_CONFIG_FILE>

Where, <PATH_TO_YOUR_SYSTEM_CONFIG_FILE> is the path to the config file you used to initialize the system.

Now, upload the file to the DFS, run this command on the client program:

upload <PATH-TO-THE-INPUT-FILE-YOU-CREATED> wc 21

You should see an output similar to this one:

```
upload ../../../examples/wordcount/wcin.txt wc 21
> Will start file upload process...
Uploading chunk 1 of 4...
Uploading chunk 2 of 4...
Uploading chunk 3 of 4...
Uploading chunk 4 of 4...
File uploaded successfully!
```

Next, submit your job to the MR system

create <PATH-TO-THE-JOB-CONFIG-FILE-YOU-CREATED>

You should see an output similar to this one:

```
create ../../../examples/wordcount/wc-job.txt
> Submitting job to the MR master...
list
Created job: [(Job ID: 1), (Job name: WCJob), (Mappers completed: 0), (Reducers completed: 0)]
```

Now you have to wait a little bit until the job finishes. You can issue list commands to see the state of your job:

```
list
> [(Job ID: 1), (Job name: WCJob), (Mappers completed: 0), (Reducers completed: 0)]
list
> [(Job ID: 1), (Job name: WCJob), (Mappers completed: 4), (Reducers completed: 0)]
list
> [(Job ID: 1), (Job name: WCJob), (Mappers completed: 4), (Reducers completed: 0)]
list
> [(Job ID: 1), (Job name: WCJob), (Mappers completed: 4), (Reducers completed: 0)]
list
> [(Job ID: 1), (Job name: WCJob), (Mappers completed: 4), (Reducers completed: 1)]
```

Eventually, you will get this output after running the list command:

```
list
> No jobs currently running...
```

Great! This means that your job finished executing. Now use the "finished" command to see the status:

```
finished
> [ID: 1, Name: WCJob, Success: true, Output: { wc-0.out wc-1.out }]
```

Now you can download the output of the reducers (wc-0.out, wc-1.out) by using the download command, for example:

```
download wc-0.out ./wc0.txt
> Starting download process, please wait…
File downloaded successfully!
download wc-1.out ./wc1.txt
/Users/Kaver/Desktop/wc1.txt
File downloaded successfully!
```

You should see an output similar to this one:

```
AAAAAAAAAAAAAAAAAAAA,5
AAAAAAAAAAAAAAAAAABB,6
AAAAAAAAAAAAAAAAABAB,3
AAAAAAAAAAAAAAAAABBA,6
AAAAAAAAAAAAAAAABAAB,3
AAAAAAAAAAAAAAAABABA,4
AAAAAAAAAAAAAAAABBAA,5
AAAAAAAAAAAAAAAABBBB,5
AAAAAAAAAAAAAAABAAAB,6
AAAAAAAAAAAAAAABAABA,4
AAAAAAAAAAAAAAABABAA,4
AAAAAAAAAAAAAAABABBB,4
AAAAAAAAAAAAAAABBAAA,4
AAAAAAAAAAAAAAABBABB,5
AAAAAAAAAAAAAAABBBAB,5
```

---

6. Correctness

All of the required features are implemented and the code runs as expected.

---

7. Technologies/Third party libraries

- Maven: For managing external dependencies the project. Fortunately, ghc machines have a maven installed binary under /usr/local/netbeans-7.2.1/java/maven/bin/mvn.

- JCommander: Command line argument parser for Java.

- Apache log4j: Logging utility. This dependency is downloaded and added to the executable jar automatically by maven.. This dependency is downloaded and added to the executable jar automatically by maven.

- Paramiko: Python implementation of the SSHv2 protocol, providing both client and server functionality. Used for system bootstrapping.

---

8. Future work

- Implement generics on mapper and reducer classes:

Our current implementation requires that both the key and values for a mapper or a reducer are strings. The framework should in general allow for any arbitrary key/value types, as long as they satisfy some basic constraints.

- MR Master and DFS master should run as separate services:

As mentioned previous in our document, in our current implementation, we run mr master and dfs master as a single binary. While it was felt that investing time in running them as separate service wasn't worth as far as this project was concerned, in order to productionize the system in real world, it's important that they run as separate services since a mr master failure shouldn't affect a dfs master failure and vice versa.

- Refactor functionalities from setup script:

Currently, our setup script is one big monolithic python script which has quite a few independent functionalities. It would be better for system administrators if we were separate each of the functionality into a separate script.

9. Learnings

- When dealing with a complex distributed systems, it's always a better idea to outline interactions between different entities and assign responsibilities to each component. It's also not a bad idea to check in interfaces and then start with implementation.

- Investing time in a setup script is worth the effort. Although it took us a while to get it working, but once we had it coded up, we could effortless build and deploy our system to multiple machines in few tens of seconds.

- Debugging a distributed system can be extremely painful if ample number of logging statements are not added to the code. Using a logging library along with setting the appropriate level of log messages (like DEBUG, INFO, WARN, ERROR or FATAL) can really help in sifting through the noise of data generated and figure out what exactly is going wrong with the system.