15-440 Distributed Systems
Alex Cappiello (acappiel)
Michael Ryan (mer1)

Lab 3: Distributed File System and MapReduce

The object of this lab was twofold. First, the design and implementation of a distributed file system (DFS). Second, the development of a MapReduce (MR) framework to run parallel compuations. Major objectives include:

1. The distributed file system supports replication of files across multiple data nodes.
2. Maximizing efficiency by scheduling work to keep map and reduce nodes as busy as possible and assigning work that is location aware, so that IO to the DFS is minimized.
3. Maximizing throughput by taking advantage of multiple cores on a single node.
4. Handle failures on map/reduce nodes by restarting failed jobs.
5. Define an API for MapReduce tasks.
6. Implement management tools for administrative tasks.

I. Definition of System Components

DFS: Name node

Holds all of the meta-data about the files and data nodes currently on the DFS. Any file operation will first (internally) contact the name node and subsequently communicate with the relevant data node(s). The name node initially waits for all data nodes to connect before serving client requests. The name node is responsible for both looking up existing files and determining where to locate new files. Unless the client specifies where to put a file, the load is randomly distributed between data nodes.
For usage infomration, run bin/nameserver -h. The name node must start before any data nodes.

DFS: Data node

It is responsible for executing the actual file IO for the subset of files that it contains. On disk, each node maintains a separate directory for its contents. This is also used to provide files on startup, if any are already present in this directory, (the limitation of this feature is that the replication factor must already be enforced). Although not required, a directory structure may be used in filenames. Performance considerations were made to maximize read throughput, at the expense of write throughput, since read frequency dominates write frequency. The consistency policy is read one, write all. Also,

recently accessed files are cached in memory to decrease disk IO. However, in order to make writes durable, they are write-through to disk.
For usage information, run bin/dataserver -h.

DFS: Client Interface

The client communicates with the DFS using the DistributedIO java interface. This is implemented in the Connection class, which establishes a connection to the name node initially and is then used to perform file operations. A full listing of the available operations to the client is available in javadoc form at http://alexcappiello.com/projects/15440/Project3/.

DFS: Admin Tools

The administrative tools for the DFS allow querying the current state of the filesystem. The executable is bin/query and the usage is

```
$ bin/query -h
usage: query [-h] [-n <arg>] [-p <arg>] <FILES | NODES>
 -h,--help              Display help.
 -n,--hostname <arg>    Hostname to manage. Default: localhost.
 -p,--port <arg>        Port to connect to. Default: 9000.
```

For example:

```
$ bin/query -n localhost -p 9000 FILES
Filename      Location
------------------------------------------------------------------------------
foo.txt         [alex2-Desktop:8000, alex2-Desktop:8001]
abc.txt                 [alex2-Desktop:8000, alex2-Desktop:8001]
```

The name node can be queried to get all currently tracked files and all connected data nodes. The data nodes can be queried to get all of their files.

MR: Master

The JobTracker listens for both client requests and user responses. It is a single point of failure for the MikeReduce framework. Once a client makes a request, it spins up

MR: Mapper

Mappers might be better called workers – these nodes run all map and reduce work for

the framework. Map tasks are given a filename for input and output as well a mapper class. They then read input from the distributed filesystem and partition output according to the number of reducers that the task will use. This speeds up reduce time, as it can pull from several sources simultaneously. Reducers are given a set of filenames from which to read and output to a file marked with an integer less than the number of reduce nodes. If output is "out.txt" and 3 reduce nodes are used, output files will be "out.txt0", "out.txt1", and "out.txt2".

MR: Client Interface

The main client points of extension are the Mapper class and Reducer class. These each define and InputFormat and an OutputFormat, which parse and format inputs and outputs respectively. The actual map and reduce functions each take inputs and, after performing operations, should commit the line on their respective MapContext or ReduceContext (See code for examples).

The client interface is accessed through the runClient.sh script in the /bin directory. Users can request a list of currently running tasks (-l) or submit a new task as specified in the config file, which uses the .ini format. See test1Config.ini and test2Config.ini for examples. Required command line arguments include -a <hostname> and -p <port> to specify the hostname and port of the JobTracker.

MR: Admin Tools

The MapReduce framework can list all jobs currently running via the runClient script and the -l flag.

II. Points of Extension

DFS:
Currently, the robustness of the DFS is minimal. The replication protocol is read one, write all. So, the system can continue serving read requests in the event of failure, but not write requests. If writes occur while a node is down, then the file system may no longer be consistent. However, if the data node is restarted before any writes occur, the DFS can continue running. Another extension might be some degree of client-side caching, although the MapReduce framework avoids making redundant calls in the first place.

MR:
The robustness of the MR is also questionable. If a node fails, it should simply note the work being run on the node at that time and run it again when another node becomes available. The

JobTracker is a single point of failure. Additional scheduling would also benefit the system.

III. Discussion of Specification

We feel that we have met all of the goals outlined in I.

1. The DFS supports a user-specified degree of replication. This replicated data is used to load-balance accesses to a particular file across all replicas.
2. The MR framework keeps track of how many cores are available on each node, but does not use this information for much scheduling. The same is true for file locaitons on the DFS.
   The DFS exposes some information about the location of files, so that location-aware client code can be written. The first is that the client can request that a file is created on a particular node (while still replicated on others). Also, for read operations, the client can specify a node to give priority to. An attempt is first made to read from this node, only moving on to others if it is not available. Finally, the client can get a list of where the file is located to augment its own scheduling decisions.
3. Many tasks can run on the same node if multiple jobs are submitted at once.
4. Upon failing to send its scheduled heartbeat, all of a node's tasks will be rescheduled to run on other nodes.
5. The main client points of extension are the Mapper class and Reducer class. These each define and InputFormat and an OutputFormat, which parse and format inputs and outputs respectively. The actual map and reduce functions each take inputs and, after performing operations, should commit the line on their respective MapContext or ReduceContext (See code and Javadoc for examples).
6. The management tools are fairly simplistic, but allow a queries that are informative to help ensure that the current state of the system is as expected.

IV. Build Information

We built this project using the gradle build system. From the top-level dir, run

```
$ gradle assemble
```

to compile the source. Gradle should handle all dependencies. Since gradle isn't natively available on unix.andrew.cmu.edu, build with

```
$ ./build.sh
```

If this script doesn't find gradle in the local path, it will use an install from a publicly accessible directory on afs. You may wish to clear out $HOME/.gradle afterwards.

Additional usage information is given in section VI.

V. External Libraries Used

Gradle (If on afs, just use build.sh. Otherwise, you're on your own.)

Java Libraries (automatically acquired by Gradle)
Apache Commons CLI
Apache Commons IO
Apache Commons Lang
ini4j

VI. Example Usage and Tests

Scripts have been provided to start and stop the JobTracker+NameNode and several worker/data nodes. Usage is:
$ cd bin
$ ./start.sh
Usage: ./start.sh <hostfile>
$ ./stop.sh
Usage: ./stop.sh <hostfile>

Two hostfile configs have been provided. One starts two worker/data nodes on localhost. The other starts four worker/data nodes on various unix.andrew machines. The JobTracker+NameNode is always started on localhost. For example:
$ ./start.sh unixhosts.txt
$ ./stop.sh unixhosts.txt

STDOUT is redirected to the logs folder (though logging is minimal) and STDERR is not redirected.

After starting the MapReduce and DFS infrastructure, run a test:

test1Config.ini: Identity map and simple reduce on very simple test files (each line is 2 characters separated by a space)
test2Confg.ini: Inverts indices of Magic: The Gathering card names and ID numbers.

Run instructions:
./runClient.sh -c ../test1Config.ini

We hope you enjoy running the MikeReduce framework on top of the Alex Filesystem.