15 - 640 Project 3 Report

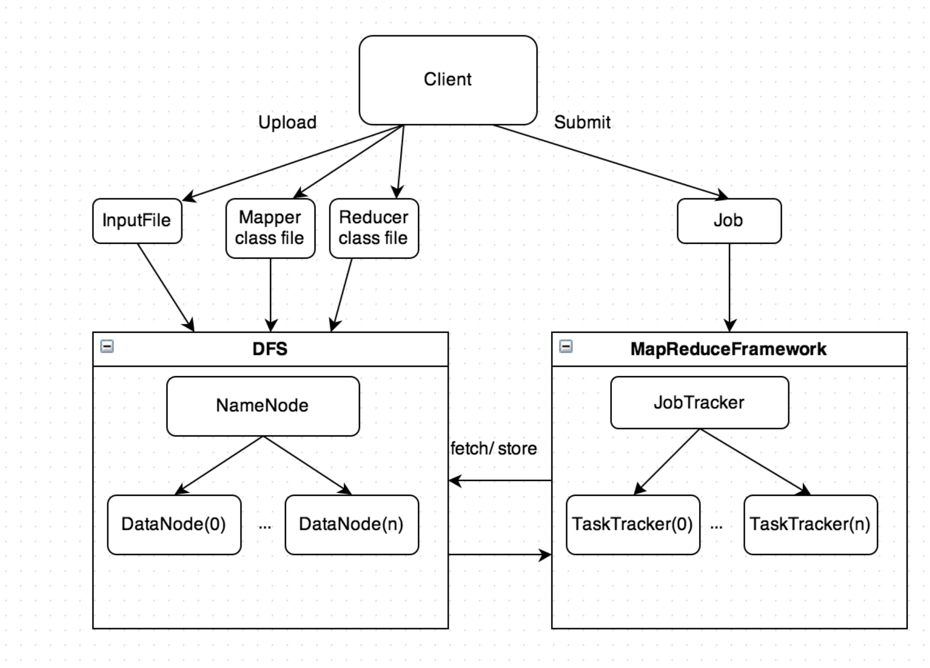
MapReduce Framework and DFS

Erdong Li (erdongl)   Yanan Jian (yjian)

1 Design Overview

The system is consisted of two parts: MapReduce Framework (MapReduceFM) and Distributed File System (DFS). Through API provided by DFS, Client can upload input files, mapper and reducer class files. With API provided by MapReduceFM, Client can extend Mapper and Reducer classes, override mapper and reducer functions, submit mapreduce Job to this system, monitor Job status and terminate Job.

MapReduceFM is running on DFS, which made the system robust and



extensible.

Fig.1: System overview.

2 Implementation Details

The project contains three packages:

- bin - scripts to run example jobs;
- conf - config file, clients can customize params in config.json;

- src - source code for DFS and MapReduceFM;

Within src package:

- dfs - implementation of distributed file system;
- mr - implementation of mapreduce framework;
- testmr: example programs (WordCount, WikiMediaFilter);

Each DataNode is identified by a machine number, the number starts from 0 and increased by 1 each time a new DataNode joins.

TaskTracker is running on the same machine as DataNode, so that TaskTrackers have the same identification numbers as DataNodes.

2.1 DFS

The interactions between DFS facilities are as shown bellow:

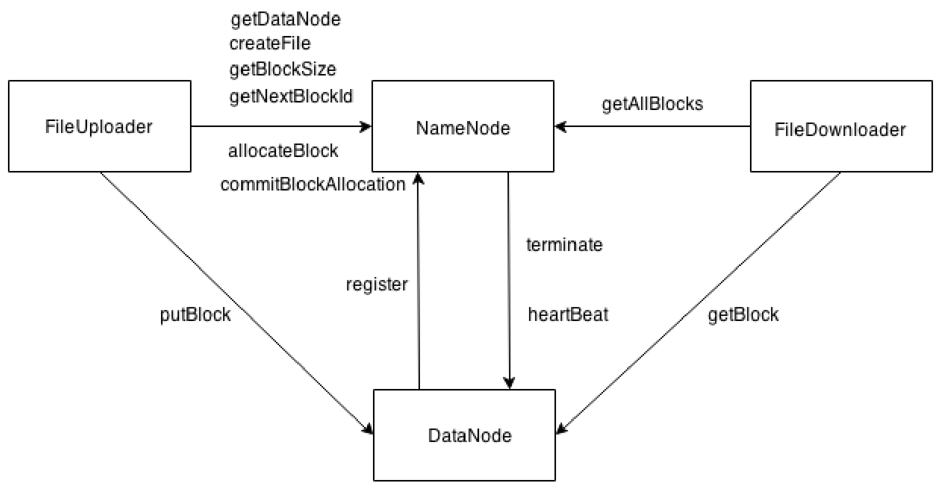


Fig 2. Interaction between NameNode and DataNode

2.1.1 NameNode and DataNodes

DFS uses a master-slave architecture. A DFS cluster consists of a single NameNode that manages the file system metadata and multiple DataNodes that manage storage attached to them. The metadata stored in the NameNode includes:

- File information that maps a file's filename to its replication factor and block ids;
- Block information that maps a block's id to the corresponding filename and a list of DataNode ids that contains this block;

- DataNode information that maps a DataNode's id to a DataNode's stub, a list of block ids that it has, and its status indicating if it is alive;

The DataNodes provide block read/write services to the file system's clients. The DataNodes also perform block creation and replication upon instruction from the NameNode. The system is designed in such a way that actual data never flows through the NameNode, the NameNode is only responsible for metadata management. Java's RMI is used for communication among NameNode, DataNodes, and clients. The NameNode will only initiate RMI for health check and termination request, and will respond to RMI requests issued by DataNodes or clients.

2.1.2 Bootstrap Phase

Upon bootstraping, the NameNode will create a registry on the specified port, and rebind itself to the registry. It then reads the fsImage, an image of the entire file system namespace and file-block map in memory, from local file system, creates metadata entries and inserts them into corresponding data structures. DataNodes export themselves to the local runtime, and send the returned stubs to the NameNode for future uses.

2.1.3 File Upload

Clients can upload a file to DFS using FileUploader. The FileUploader first contacts the NameNode to insert metadata including file name and replication factor into the FileInfo table, and retrieve the predefined block size from the NameNode. Then the FileUploader begins caching the file data into local memory until the size exceeds the block size limitation. At this point, the FileUploader first retrieves the next block id from the NameNode, and then ietartively retrieves an identifier to the destination DataNode, flushes the data from local memory to the assigned DataNode, and commits if the block allocation operation succeeds until replication factor is enforced. If the FileUploader fails, an incomplete file will be created, but the FileUploader guarantees that the content of each relica is consistent, as incomplete data block won't get committed.

2.1.4 File Download

A FileDownloader is provided to download a file from DFS to local file system. The FileDownloader first contacts the NameNode to retrieve identifiers for all data blocks and their corresponding DataNodes. Then the FileDownloader tries to get the data block from each of the DataNodes, flushes the data to the local file if succeeds, and repeats for the next data block. If the FileDownloader fails to retrieve a data block from all the corresponding DataNodes, it throws an Exception to notify the client, and an incomplete local file will be created.

2.1.5 Data Replicatioin & Robustness

2.1.5.1 DataNode Selection for Replication

Since the DFS is to be deployed in the Andrew cluster, we mainly focus on workload balancing instead of locality. When the NameNode receives a request to allocate a block, it responds with an identifier to the DataNode with least data blocks. Therefore the numbers of data blocks on each DataNode are relatively the same, which benefits the upper layers, for example, the job sheduling of MapReduce facilities.

2.1.5.2 DataNode Failure, Heartbeats, and Re-Replication

The NameNode periodically performs health check to all the alive DataNodes. If a DataNode failed to respond, the NameNode would mark it as dead and stop forwarding any incoming block allocation assignment to it, as any data that was registered to a dead DataNode is not available to DFS any more. To enforce replication factor, the NameNode replicates all data blocks originally stored in the dead DataNode, and put them in an alive DataNode. Again, the DataNode retrieves the replica directly from another DataNode. Actual data never flows through the NameNode to reduce its workload.

2.1.5.3 DataNode Restart

When a DataNode reboots, it first exports itself to the local runtime, and then register to the NameNode to start receiving requests. As the DFS has re-replicated all its data blocks at the time it failed, all the data blocks this DataNode has can be considered as extra copies, which violates the replication factor. However, the DFS won't explicitly delete these extra copies for several reasons:

- Disk is cheap, and having more replicas won't affect the functionality of the DFS;
- Deletion may affect the MapReduce facility if, for example, a Mapper is working on that data block;
- Deletion may affect the MapReduce facility if, for example, a Mapper is working on that data block;

2.1.6 Termination

Clients can terminate the whole DFS using DFSTerminator. The DFSTerminator simply notifies the NameNode to initiate the termination process, and catches any exception thrown by the NameNode. The NameNode first flushes out all metadata into the on-disk fsImage, and then notifies each alive DataNode to terminate. The DataNodes unexport themselves from the local runtime, and terminates. The NameNode then unexport itself and the registry from the local runtime, and the termination finishes. During termination, the NameNode stops accepting any incoming requests from clients, and stops performing health check to DataNodes.

2.2 MapReduceFM Interaction

MapReduceFM uses RMI for communication between JobTracker and TaskTracker. The job control flow is as shown in Fig 3.

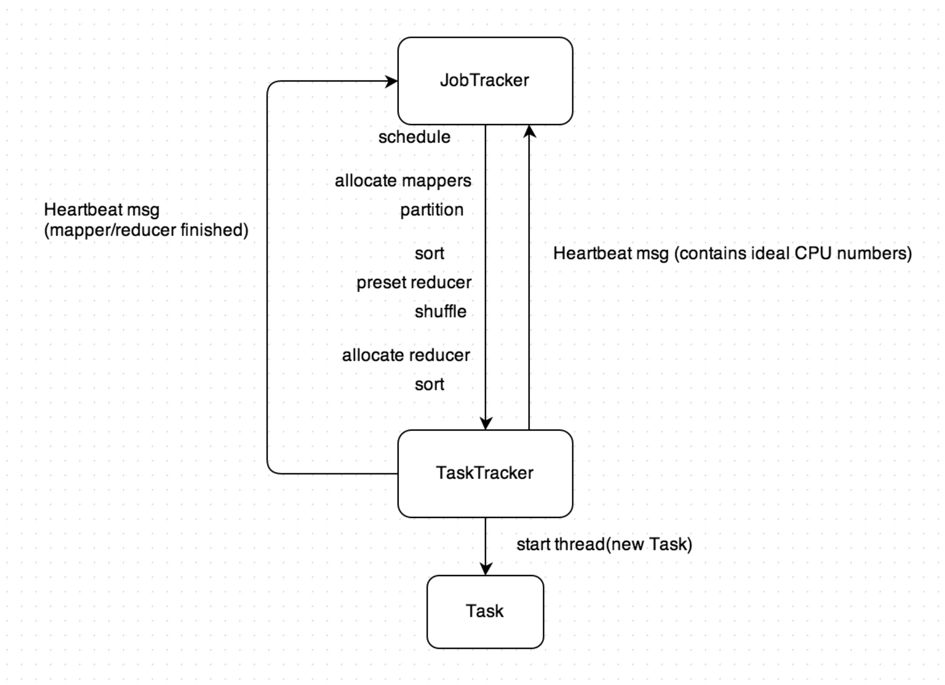


Fig 3. Interaction between JobTracker and TaskTracker

After Client submitted a Job, the first step is for JobTracker to schedule mapper tasks.

2.2.1 Schedule

The input files uploaded to DFS have been split into blocks, JobTracker assign mapper tasks to TaskTrackers based on the principle of one mapper per block. TaskTracker then wrap the task into Task instance, start a new thread to execute Task and periodically check if the thread has finished. TaskTracker reports to JobTracker once any thread is finished through heartbeat message. For each Job, when JobTracker received all 'mapper finished message' from TaskTrackers, it will go on to the 'preset reducer' phase.

2.2.2 Partition

The partition phase splits mapped result into partitions based on number of reducers. In order to save memory, mapper writes each line into a fixed size TreeMap, when the fixed size buffer is filled up, dump the buffer to tmp file, after all the mappers have finished, enter partition phase and hash keys to different partitioned files using external n-way merge. The partitions were put under the directory '/tmp/[jobID]/[machineID]/', and they are named as '[mapperID] _machineID@[hash[mapperID]%[reducer_counts]]'.

2.2.3 Sort

Sort partitioned files using merge sort.

2.2.4 Preset Reducer

After sort phase finished, all sorted partitions have been saved onto local disks. Preset reducer allocates reducer tasks based on available CPUs and locality principle, that is according to the partitions each machine has, allocate reducer tasks to minimize file transfer among machines. This phase pre-allocates reducer tasks, not actually starts them.

2.2.5 Shuffle

Based on the result of preset reducer phase, move partitions among different machines to ensure that each reducer can deal with the whole set of keys, that is to ensure the same key can not be processed by two reducers.

2.2.6 Allocate Reducer

After shuffle phase, partitions has been grouped, based on the result from preset reducer, JobTracker calls TaskTrackers to start reducer threads. TaskTrackers then start reducer threads and periodically check if the reducers have finished.

2.2.7 Sort

Each reducer sort the partitions using external n-way merge before starting reduce phase.

2.2.8 Robustness

TaskTracker periodically send heartbeat message to JobTracker, if a TaskTracker is down, JobTracker will notice and will reschedule jobs that were running by the failed TaskTracker. The rescheduling of jobs will cause the jobs run from mapper phase again. The reason of choosing this solution is that mappers are running in parallel, the cost of time of starting one mapper compared with starting all mappers is of no difference.

2.2.9 Job Control

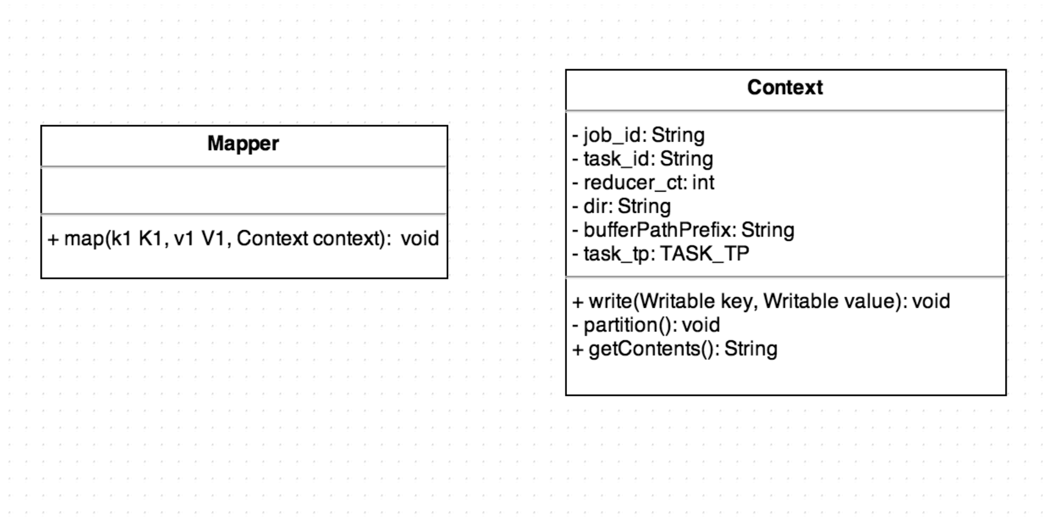
Client can check job status by using MRDescriber, client can check all job status, the specified job status. Job status is stored in Job Class per job, the HashMap<String, Job> jobID_job records all jobs running in this system. Client can also kill all jobs or a specific job by using jobID as param. When killing job, TaskTracker will cancel the running tasks within the job, report to JobTracker of the termination of tasks.

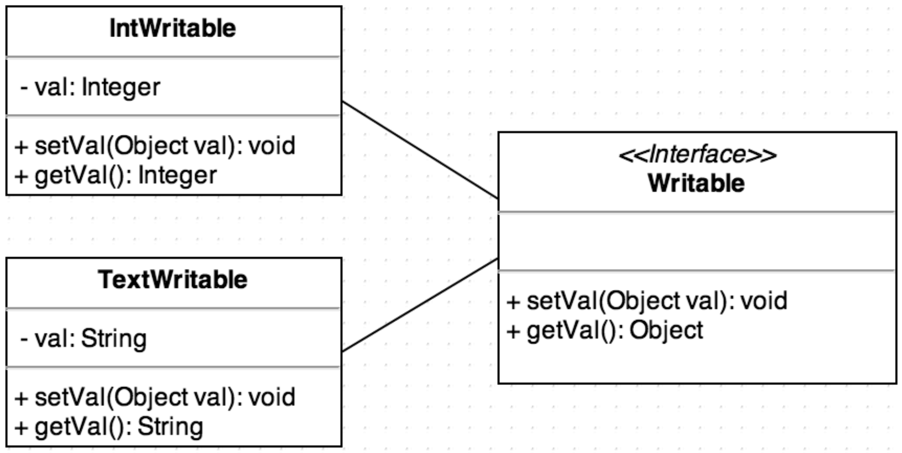
2.2.10 Termination

Client can terminate the MapReduceFM by using MRTerminator, the MRTerminator notifies JobTracker to terminate all running jobs and then send 'terminate' message to all TaskTrackers, after that, JobTracker export itself. TaskTrackers export themselves from local runtime, during termination, TaskTrackers ignore all incoming requests, and stop sending heartbeat messages to JobTracker.

3 API design

3.1 Mapper:

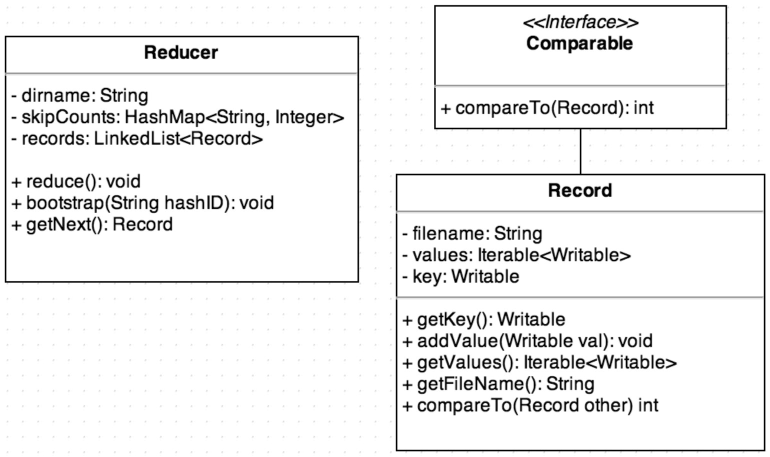




K1, V1 can be any kind of IntWritable or TextWritable.

Client can write Mapper class which extends Mapper class provided by mr.io, override mapper function, call context.write within the mapper function.

3.2 Reducer


Reducer
- dirname: String
- skipCounts: HashMap<String, Integer>
- records: LinkedList<Record>
+ reduce(): void
+ bootstrap(String hashID): void
+ getNext(): Record
<<Interface>>
Comparable
+ compareTo(Record): int
Record
- filename: String
- values: Iterable<Writable>
- key: Writable
+ getKey(): Writable
+ addValue(Writable val): void
+ getValues(): Iterable<Writable>
+ getFileName(): String
+ compareTo(Record other) int

Client can write Reducer class which extends Reducer class provided by mr.io, override reducer function, call context.write within the reducer function.

3.3 Java Documentations

Go to $PROJECT_ROOT, enter

```
ant doc
```

And find the documentations under $PROJECT_ROOT/doc/index.html.

4 Configuration

All configuration parameters are stored in $PROJECT_ROOT/conf/config.json including:

Name	Description
block size	Block size in bytes.
replication	Default number of replicas.
reducer	Number of reducers per MR job.
healthcheck interval	Determines DataNode heartbeat interval in seconds.
fsImage dir	Where fsImage is stored.
dfs_registry	Host name and port number of DFS's registry.
mr_registry	Host name and port number of MapReduceFM's registry.
namenode port	Port number of the NameNode service.
jobtracker port	Port number of the JobTracker service.
datanode	Specifies the port numbers and directories of each DataNode.
tasktracker	Specifies the port numbers and directories of each TaskTracker.

5 Usage, Demo and Test

5.1 Build the Project

Go to $PROJECT_ROOT, enter the following command to build the project:

```
ant
```

5.2 Start DFS and MapReduceFM

Go to $PROJECT_ROOT/conf and modify the configuration file. Notice that dfs_registry, mr_registry, NameNode and JobTracker should be running on the same host.

5.2.1 Start NameNode and JobTracker

Log on an Andrew machine, go to $PROJECT_ROOT/bin and enter the following commands:

```
python start_namenode.py

python start_jobtracker.py
```

5.2.2 Start DataNodes and JobTrackers

For each Andrew machine that the DFS and MapReduceFM is to be deployed on, go to $PROJECT_ROOT/bin and enter the following commands:

```
python start_datanode.py <datanode id>
python start_tasktracker.py <tasktracker id>
```

5.3 Upload/Download A File

5.3.1 Upload A Text File

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python dfs.py put_text <local path> <dfs filename> <replication factor>
```

5.3.2 Upload A Class File

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python dfs.py put_class <local path> <dfs filename> <replication factor>
```

5.3.3 Download A Text File

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python dfs.py get_text <dfs filename> <local path>
```

5.3.4 Download A Class File

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python dfs.py get_class <dfs filename> <local path>
```

5.4 Check DFS/MapReduceFM Status

5.4.1 Check DFS Status

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python dfs.py describe
```

5.4.2 Check MapReduceFM Status

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python mr.py describe [job id]
```

5.5 Terminate DFS and MapReduceFM

Go to $PROJECT_ROOT/bin, and enter the following commands:

```
python dfs.py terminate

python mr.py terminate
```

5.6 Kill Job

Go to $PROJECT_ROOT/bin, and enter the following command:

```
python mr.py kill [job id]
```

5.7 WordCount

We've provided a sample input called WordCount.in under $PROJECT_ROOT/bin. To run WordCount, we suggest you to set block size to be 512 for small input file. Then enter the following commands under $PROJECT_ROOT/bin:

```
sh testWordCount.sh <mr_registry.host> <mr_registry.port>
```

mr_registry.host and mr_registry.port can be found in $PROJECT_ROOT/conf/config.json.

The output would be WordCountOutput/<reducer_id> on DFS. You can use "python dfs.py describe" to check the file names, and download it to the local file system using

"python dfs.py get_text <dfs filename> <local path>".

5.7 WikiMediaFilter

This test program will explore WikiMedia's page views log, do a little regular expression analysis, and filter out all unqualified page view records.

Input files for WikiMediaFilter can be downloaded from

http://s3.amazonaws.com/wikitraffic/201306-gz/pagecounts-20130601-000000.gz

Please put the downloaded file in $PROJECT_ROOT/bin and unarchive it. We strongly suggest you to set the block size to be 64000000 so that DFS won't generate a huge amount of blocks. Then go to $PROJECT_ROOT/bin and run the following command:

```
sh testWiki.sh <mr_registry.host> <mr_registry.port>
```

Please be patient. The execution may take a relatively long time.

The output would be PageCountsOutput/<reducer_id> on DFS. You can use "python dfs.py describe" to check the file names, and download it to the local file system using

"python dfs.py get_text <dfs filename> <local path>".

We also privided a small test set called "pagecounts_small.in" under $PROJECT_ROOT/bin. As it is a small input file, we suggest you to set the block size to be 512, and run the following command to execute the test program:

```
sh testWikiSmall.sh <mr_registry.host> <mr_registry.port>
```

The output would be pagecounts_small_output/<reducer_id> on DFS. You can use "python dfs.py describe" to check the file names, and download it to the local file system using

"python dfs.py get_text <dfs filename> <local path>".

5.8 Clean the Project

Go to $PROJECT_ROOT, enter the following command:

```
ant clean
```

6 Limitations and Known Issues

- There are many disk accesses in our approach, and therefore the disk access time may become the bottleneck of the process speed;
- Since we are not doing logging/checkpointing, and NameNode is a single point of failure that won't be able to recover from failures.
- Deletion is not supported for DFS;

7 Assumptions

- (#total DataNodes) - (#failed DataNodes) > (#replicas) at any given time;
- Data blocks remain unchanged during system failure/reboot;
- Input files are text files and Mappers will process input line-by-line;