Report of MapReduce Facility

Project 3 of 15-640

Kailiang Chen(kailianc) & Yang Pan(yangpan)

# Design

In this project, our goal is to design and implement a **MapReduce Facility**, similar to Hadoop, but with certain design constraints aimed at enabling it to work more efficiently in our computing environment with smaller data sets. It is capable of dispatching parallel maps and reduces across multiple hosts, as well as recover from worker failure.

The whole MapReduce framework in Hadoop consists of four independent entities:

1. The Client, which submits the MapReduce job.
2. The Jobtracker, which coordinates the job run.
3. The Tasktrackers, which run the tasks that the job has been split into.
4. The Distributed Filesystem(HDFS), which is used for sharing job files between the other entities.

Hadoop MapReduce provides a clear interface for Application Programmer to customize specific Mapper and Reducer by overriding map and reduce functions to accomplish different jobs. When the job is submitted from application to the framework , it is passed from JobClient to the JobTracker, and then divided into tasks to pass to the TaskTracker. By RPC and HTTP transmission, each TaskTracker sends heartbeat to inform its status to the JobTracker. MapTask and ReduceTask are launched concurrently in a Child process to accomplish each small tasks, which consists of split, map, sort, partition, shuffle, and reduce. Figure1 and Figure2 describe the MapReduce workflow.

All of these work are based on a local/remote filesystem, which the latter one is called Hadoop Distributed File System(HDFS), which is made up of a NameNode and several DataNodes. Figure 3 and Figure 4 describe the HDFS read/write flow.

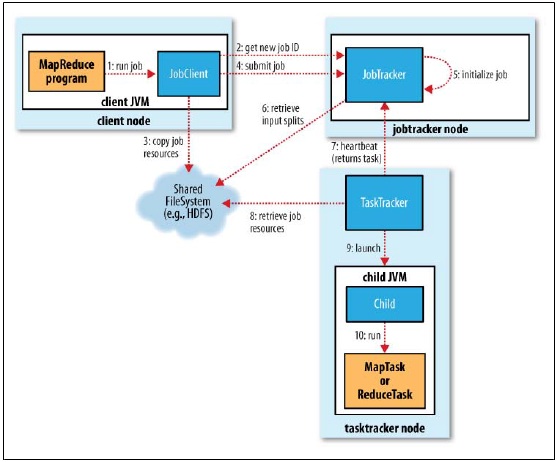
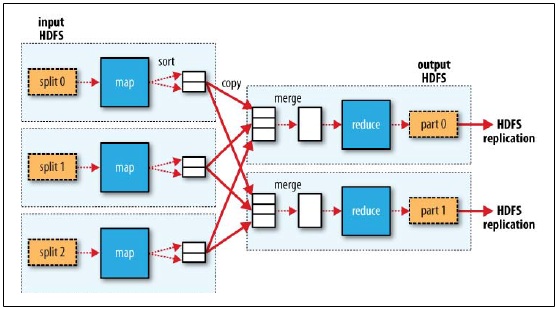
 

Figure 1 MapReduce Job Flow Figure 2 MapReduce Task Flow

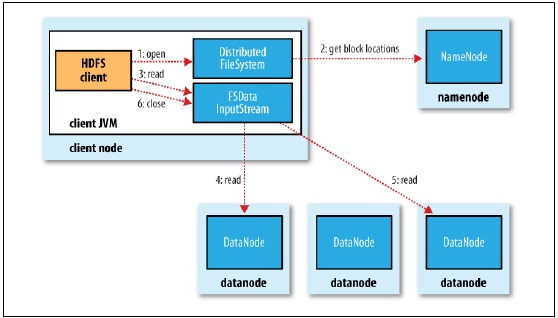
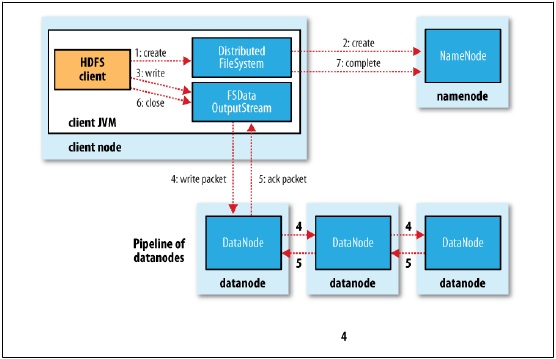
 

Figure 3 HDFS read flow Figure 4 HDFS write flow

In our design, we tried to extract and accomplish the core features of Hadoop as simple as possible to meet the requirements. Our facility is made up of four packages

1. HDFS : common file processing, transferring and sharing module designed and implemented based on Java RMI (avoid using AFS)
2. JobControl : jobs and tasks creating, dispatching and coordinating module designed and implemented based on Java Concurrency
3. MapReduce : map and reduce including file and record reading, writing, result sorting, partition module designed and implemented based on Java RMI, Concurrency.
4. Network : communication massage handling between node

According to the Writeup, we provide a Configuration File (config.txt) for customizing the following parameters

1. Master/Participant nodes HostName, Port Number
2. NameNode/DataNode HostName, Port Number
3. FileChunkSize
4. NumOfReducer
5. Maximum Number of Mapper, Reducer Tasks
6. SplitFile, MapperResultFile, ReducerResultFile Directory Name

We initiate the execution of the program from the MapReduce main function, which can be started on any participating node, only if configuring the Master HostName. E

We execute portions of the program other than maps and reduces on the participant nodes, such as splitting the input file, partitioning.

After the input file is split, the map and reduce tasks are dispatched to the participant nodes, which can be executed concurrently. We define a linkedlist data structure to store assigned tasks in participant nodes, which allows multithread processing of all the tasks if necessary. Accordingly, when failure happens, appropriate processing of the waiting tasks other than running ones should be considered. Here, we simplify the design to re-dispatch all the waiting tasks if failure happens.

The JobManager and JobDispater schedule and dispatch maps and reduces tasks to maximize the performance gain through parallelism. Similar to Hadoop’s design, we prepare a JobQueue in JobManager. If there is a job request submitted, it will be dispatched to a participant node.

We also provide the recovery tolerance. Before each map task, we set a specific number of replication. If a map task on a participant node failed, we look for a new participant node, and restart the map task. Similar to Hadoop, the participant failure is detected by heartbeat transmission, in other words, if in a period of time, no heartbeat is sent to the master to report a healthy status, this participant node is considered unhealthy and all of its tasks will be restarted. Similarly, when reduce task failure is detected, all of its tasks will also be restarted.

We provide a general-purpose I/O facility to support a line-by-line input format in map and reduce tasks. To facilitate the implementation of Mapper and Reducer for application programmer, we define a Pair (Key-Value) Class and PairContainer (a list of Pairs) Class for input/output. The Mapper input is a file name, output is a PairContainer. However, we have simplified the Hadoop Input/Output design here to assume all the types are String format.

Besides that, we also provide a console as a management tool which can start up and shut down all the facility, as well as start, stop and monitor the jobs.

# UML

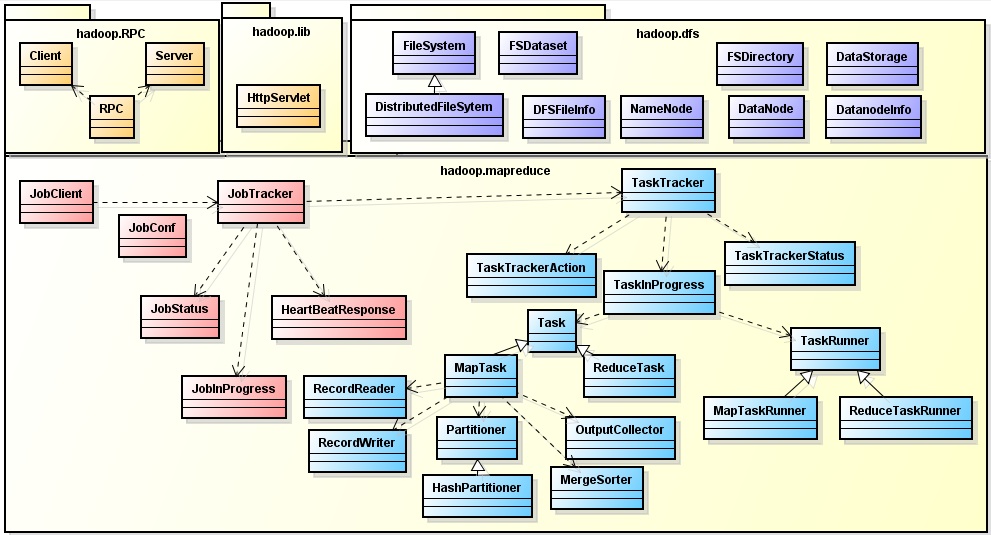


Figure 5 Hadoop MapReduce Class Diagram

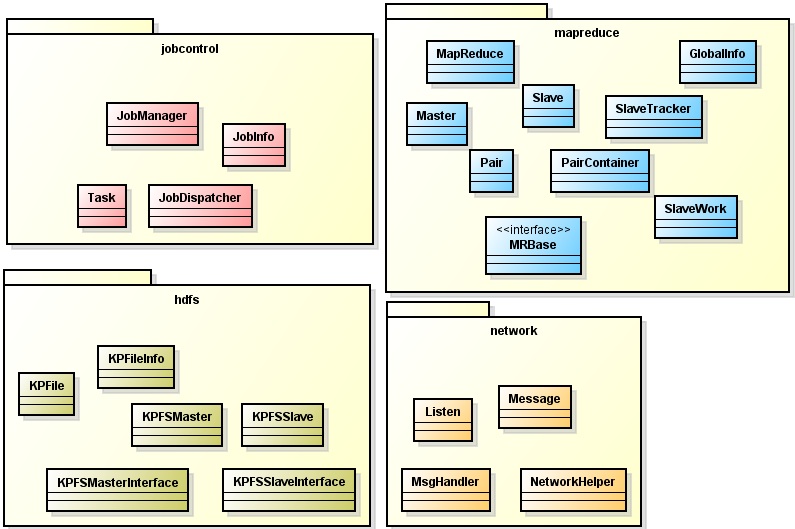


Figure 6 Our MapReduce Class Diagram

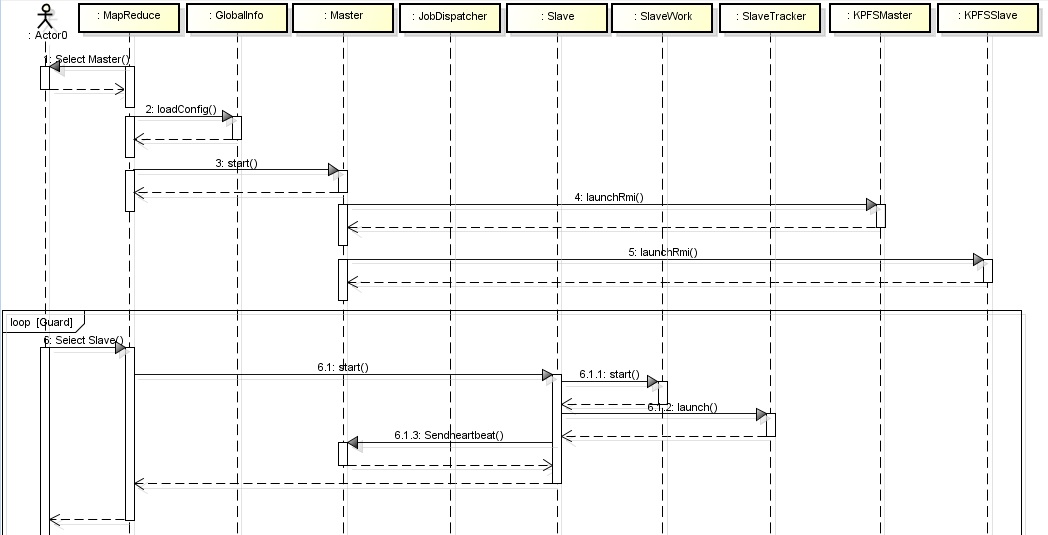


Figure 7 Initialization Sequence Diagram

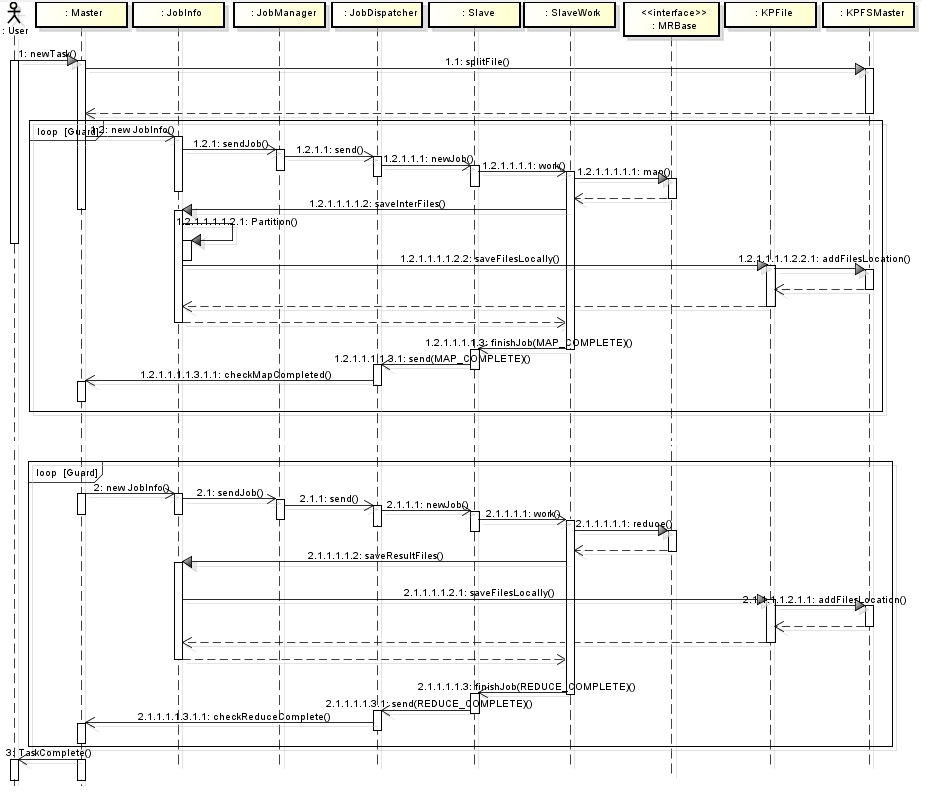


Figure 8 MapReduce Sequence Diagram

## Special features

1. Support MapReduce Master/Participant node, DFS Namenode/Datanode configuration
2. Support files transferring and read/write between MapReduce and DFS based on RMI
3. Support message communication, including task dispatch between Master and Participant node
4. Support replication and failure recovery during map and reduce tasks
5. Support easy-to-use Mapper/Reducer interface for Application Programmer
6. Support easy-to-extend and loosely coupled communication module

# Implementation

## Class Implementation

According to the UML class diagram, our system consists of:

1. RMIMessage: a concrete class defines the message transmitted between Server and Client, which is used during service lookup, method invocation and result return. Additional message type and content can be added if necessary.
2. RemoteObjectReference: a concrete class defines the service instance reference Server provides for Client, its information is managed by Server.
3. RMIService: an abstract class defines interface for services and serves as an element of Server’s information table. It supports pass by reference and pass by value depending on whether Client service extends it or not.
4. StubBase: an abstract class defines interface for services and wraps RemoteObjectReference for Server/Client communication.
5. RemoteException: a common exception class defines the specific exception for RMI facility, it can be extended by adding some message display or handling method in the future.
6. RMIServerRegistry/RMIClientRegistry: registry classes define the basic service lookup and bind functions for Server and Client. A common base class or interface can be added to improve the extensibility, however, we remain it unimplemented as it is not necessary in our project.
7. RMIServerNetworkMgr/ RMIClientNetworkMgr: communication classes define the basic message transmission functions between Server and Client. Similarly, we divide it into Server/Client module as we do for registry classes for readability.
8. RMISvrHandler: Server monitor thread for accepting messages from a client. It accepts message and passes it to RMIServerNetworkMgr to process.
9. RMIServer: Server main thread to bind a new service and launch monitor thread.
10. RMIClient: Client main thread to lookup, localize and invoke a new service.

## Development environment

This project is developed with Eclipse IDE for Java Developers, Luna Release (4.4.0), JDK 8u20. If you want to write a test class inheriting MigratableProcess, you should work in the same environment.

# Test with our examples

We provide two service classes to test the TransactionIO classes and other migration features. They are NonIOProcess and IOProcess.

NonIOProcess is a simple class with only a counter tick-tocking every 0.1s. After migration, the counter starts from the stage right before migration.

IOProcess mainly test the TransactionIO classes, TransactionalFileInputStream and TransactionalFileOutputStream. This class keeps reading in a series of shuffled characters from a file, sorting them and writing them out to a file.

## Test environment

A server at unix.andrew.cmu.edu, unix machines connected to AFS

## Deploy and run

1. Open three Terminals and connect them to a server connected to AFS using ssh. Make sure they have the same login user (saving the troubles brought by file permission).
2. Change directory to ./src/. (for convenience, we denote ProcessMigration/ as ./) in three Terminals.
3. Type “make” and then “make run” in the three Terminals.
4. In one Terminal, type “server” to become a server. And you will get an echo showing the IP and port (say, 192.168.0.1:6777). In the other two, type “client 192.168.0.1 6777) and they will connect to that server.
5. In Client A, type “create NonIOProcess” to create a migratable process without IO operations.
6. In Client B, type “create IOProcess input.txt output.txt” to create a migratable process with IO operations. Note: input.txt and output.txt should have absolute path.
7. In server, type “ps” to show all the running processes on all clients. In the two clients, type “ps” to show all the local running processes.
8. In server, type “migrate 1 0 0” to migrate the IO process on Client B (cid 1) to Client A (cid 0).
9. In server, type “migrate 0 0 1” to migrate the non-IO process on Client A to Client B.
10. In server, type “ps” to show all the running process on all clients after migration. In the two clients, type “ps” to show all the local running processes.
11. Wait for IOProcess to finish on Client A. When it shows “JOB COMPLETED”, the IOProcess finishes all the IO operations.
12. In server, type “exit” to exit server. All other clients connected to this server will exit automatically.

## Understand the test result

The result consist of three parts:

1. Migration status: In step 7, you can see the NonIOProcess running on Client A and IOProcess on Client B. In step 10, you can see NonIOProcess running on Client B and IOProcess on Client A. This means that the two processes have been migrated to each other client.
2. Process status: In step 9, we can see an echo from NonIOProcess saying its counter, like “NonIOProcess : suspend(), cnt = xxx” on Client A. After migration, we can see an echo like “NonIOProcess : run() begin, cnt = yyy”, where yyy = xxx + 1. This means the suspended work before migration is resumed after migration.
3. IO status: In IOProcess, we read a shuffled alphabet one character by one character from input.txt, sort them and write them to output.txt, and repeat this procedure 8 times. So, open output.txt, and you can see 8 set of ordered alphabets in it. This means the IO operations works fine during migration.

# Writing customized test class

By implementing the MigratableProcess class, you can write your own process class to test TransactionInputFileStream, TransactionOutputFileStream and other migration features.

## Developer environment

You can write the class anywhere you want, as long as you inherit from MigratableProcess and implement the three abstract methods. However, it’s strongly recommended to develop in the same environment as ours.

## Understand our interfaces and framework

To run your test class under our framework, you must inherit your test class from MigratableProcess. In MigratableProcess, there are three abstract classes that you have to override in your own class:

1. suspend(): This method will be called before the object is serialized. It affords an opportunity for the process to enter a known safe state.
2. resume():This method will be called after migration. Resume all the work that was suspended.
3. toString():This method is used for debugging. It can print the class name of the process as well as the original set of arguments with which it was called.

Besides, here is some other things should be noted: (assume your class file is GregProcess.java)

1. The constructor of your test class should have exactly one argument with type String[], no matter you need it or not.
2. You can design some test methods to be called after GregProcess has been instantiated. This method should have exactly one argument with type String[].
3. At the beginning of GregProcess.java, you should declare the package by typing “package edu.cmu.andrew.ds.ps;”.
4. In the suspend() method

## Deploy and run

Once you finish your work, copy your test class file (GregProcess.java) to the directory “./src/edu/cmu/andrew/ds/ps/”. And then, edit ./src/Makefile by adding “./edu/cmu/andrew/ds/ps/GregProcess.java” to “CLASSES”.

Now, change directory to ./src/ and type the following commands to run the whole program:

> make

> make run

In the client, after you connect to a server, you can type the following command to create a new instance of your process:

create GregProcess [ANY OPTIONAL ARGUMENT TO CONSTRUCTOR]

You can also call any method in GregProcess to test by typing:

call PID METHOD\_NAME [ANY OPTIONAL ARGUMENT]

where PID is the pid of that instance.

Use all these interfaces to test our framework.