COEN 317 Project

Build a MapReduce Framework

Yuan Song 00001286630 ysong3@scu.edu
Jing Jin 00001285847 jjin@scu.edu

Contents

I. Goals

Our goal is to implement a MapReduce framework and provide APIs for clients to submit and run MapReduce jobs. Basically, clients will submit the jobs to Master node. Then the master node will dispatch the jobs across multiple worker nodes. Our implementation also supports execution multiple jobs concurrently and handle failure of worker nodes.

We will discuss implementation including the detailed design decisions and tradeoffs we made in later parts, and also provide examples and instruction of using the framework.

II. Description of the Distributed System

Our system consists of one master node and several worker nodes. Clients can submit the jobs to be executed through the APIs provided by the master node. Once the master nodes get the job, it will select the proper worker nodes and partition the whole jobs into multiple tasks to these worker nodes.

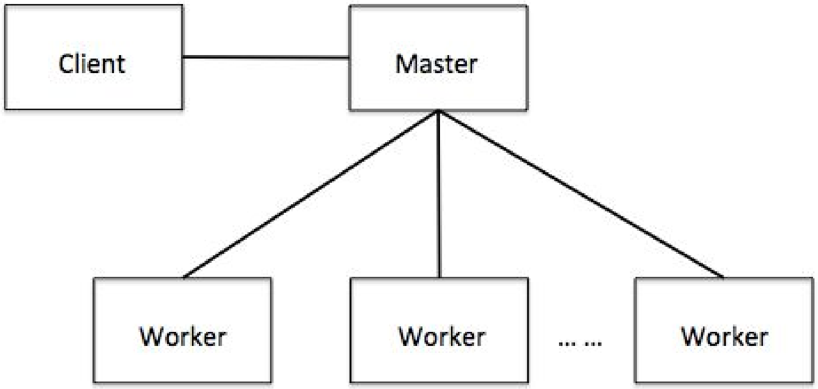


Figure 1. Overview of the distributed system

III. Related Work

MapReduce is a programming model and an associated implementation for processing and generating large data sets.[1] It is designed to handle large input data size in a parallel and distributed environment of a large cluster.

The MapReduce program consists of two steps, one is the Map step, the other is the Reduce step. First, the runtime system partition the large input into small parts and schedules different machines available in the cluster to process each parts. The Mapper reads the corresponding input data and parses the input data into key-value pair and compute its logic produce intermediate result of key-value pairs. Then the reducer sorts the intermediate results by the intermediate key and passes each key and its set of intermediate values to the reduce function. Then the final result is combined from different reducers.

And we reference the original MapReduce idea publish by Google and the open source Hadoop framework in our implementation.

IV. Design Decisions

In order to implement a basic, reliable and scalable MapReduce framework, as well as to balance the complexity of our work, we made some assumptions and tradeoffs and we will discuss the major ones.

1. Distributed File System

Generally, the MapReduce in running over a distributed file system. For time limit, we did not implement a distributed file system in this project and assumed that the input data of user's job has multiple duplications, each of which has been stored in any of the participating nodes, including the master and all worker nodes. So master and worker node can access this input file data in its local file system. In the future work, we can implement a distributed file system or implement a adapter for an existent distributed file system, such as HDFS, to let this framework can run in distributed file system environment.

2. Failure Handling

We will only handle worker failures, rather than master failure. It is more critical to handle worker failure. Once a worker fails, it is necessary to select another worker node to restart the tasks which ever running on this failed node. However, if the master node fails, we just assume to reboot the master node and restart all the jobs.

3. Communication among Nodes

All the communication among different master and worker nodes is realized through Java RMI. This mainly includes submitting jobs from client to master node, registering worker node, allocating tasks from master to different worker nodes, as well as the heartbeat of worker nodes transmitted to master.

4. Allocating tasks

In practical, the master allocates map or reduce tasks based on overall considerations of various metrics of the resources the worker has. Also the network condition should be considered. For simplifying the whole work, we only make the allocation decision depending on the available number of map and reduce slot on the worker node.

5. Thread Pools

In our implementation, we use Java Thread Pools to managing the collection of threads for performing individual tasks. It is more efficient to thread pools instead of multiple threads that thread pools can avoid the expensive overhead of invocation and terminating new threads.

6. Event-based coordination mechanism

Scheduling resources and tasks among master and worker nodes is driven by Event in our implementation. For that, we use heartbeat to report the task events which happened in worker node to master. In this way, we only transmitting specific events instead of whole tasks status data so that the workload of master node to check is far less. And we assume that there is no heartbeat transmission failure.

V. Implementation

Our system mainly consists of three parts, client, master node and worker node. Figure 1. Below shows the the architecture of our system. We will mainly discuss the master and worker parts in details.

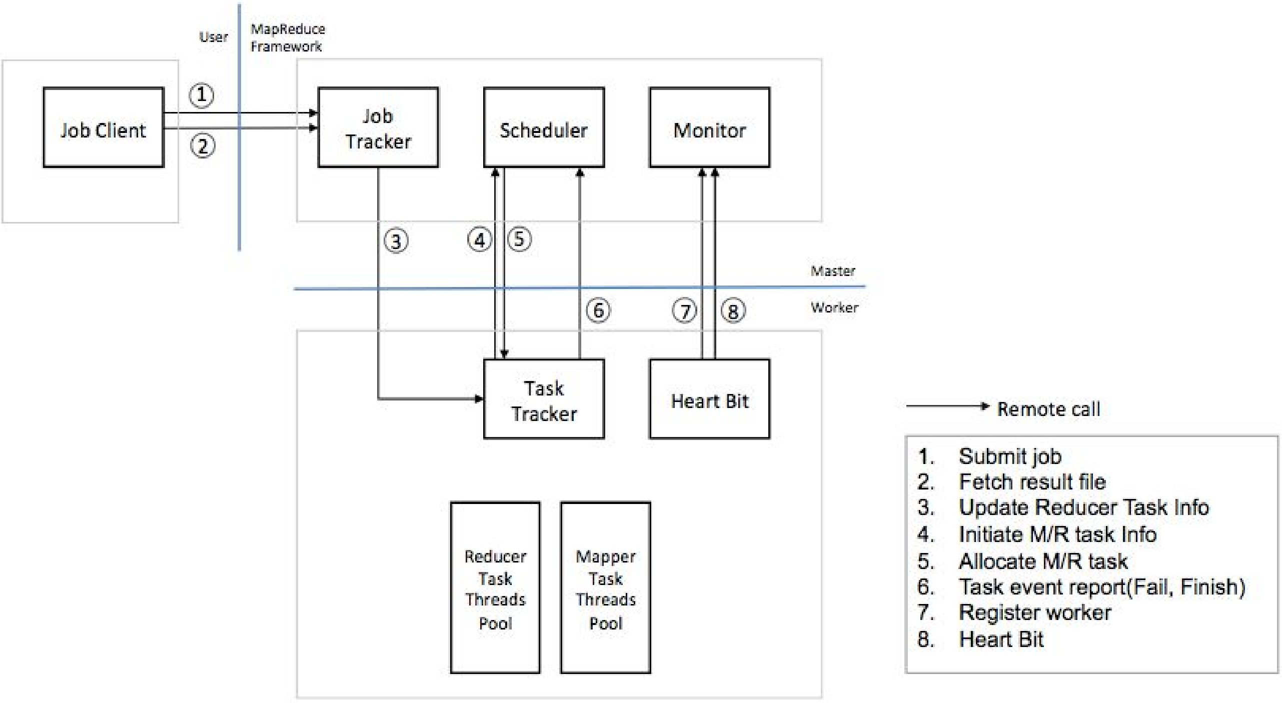


Figure 2. Overview of MapReduce Framework

1. Master

- JobTracker

The JobTracker is used to initiate the master node and managing the jobs submitted by the clients. Once the JobTracker is initiated, it starts the Monitor to coordinate all the worker nodes.

- Monitor

The monitor class is responsible for monitoring all the worker nodes in the system and also provides the interface for communication between master node and worker node. Monitor is realized as a Java RMI object, and worker nodes that aware of the RMI registry server can communicate with the monitor by calling the remote method.

When the worker gets online, it first register itself with its RMI registry information as well as the resource information. After that, the Monitor is also responsible for checking the heartbeat that each worker node sends at a constant rate (which is every second in our implementation). If the Monitor cannot track the heartbeat of a specific worker for 6 times, it assumes that this worker is currently down and sends a Worker Fails event to the scheduler in order to restart the tasks currently running on that worker node. Also for the events that are incorporated in heartbeat, the monitor will notify the Scheduler about the event, then the Scheduler will take corresponding execution.

- Scheduler

Scheduler also runs as a thread on the master server and it is responsible for scheduling the allocation of tasks of different jobs to different worker nodes and also keeping track of the status of all worker nodes. Once a job is submitted by the client, the Scheduler would first partition the whole data file into multiple parts. Then select the proper worker to run the tasks. For simplicity, the scheduler would select the worker node with most available slots.

Scheduler keeps a queue of worker node in order to allocating tasks to the proper worker. Once a worker node fails or added in the system, it would update the worker queue and take corresponding actions. And the scheduler also keeps a queue of event.

2. Worker

- TaskTracker

A TaskTracker instance is running on each worker node and is responsible for managing the map and reduce tasks, also for communicating with master. We assume that the TaskTracker is aware of the RMI registry service of Monitor. Each TaskTracker running on worker nodes is also a Java RMI object. After the TaskTracker instance is initiated, it will register itself to the master node with its RMI registry service and its available resources including number of mapper and reducer slots. After registration done, the Monitor on master node can also communicate with each worker through the RMI methods.

- Heartbeat

We implement the communication among master and worker nodes as event based. Each worker node sends heartbeat to master every 1 second, indicating that this worker is alive. The heartbeat also incorporates the event that the master need to be aware of. Once the mapper task or reducer task completes or fails, a TaskEvent object is constructed and sent to the master so that the master node will take corresponding execution.

- Map/Reduce Task

For implementation simplicity, every map or reduce task in running in a thread. And we maintain a separate map thread pool, as well as a reducer thread pool. When allocated a new map/reduce task, the TaskTracker would start a new thread to run the specific task.

For map task, as Fig. shows, once a new map task is started, it need to first process the input file into the key value pairs, then it can be process by the specific logic defined by the client into intermediate results. Then the mapper thread would partition these intermediate result based on their hash codes of the key, into multiple parts according to the number of reducer they have. In this way, the reducer tasks can easily fetch their own partition according to its reducerID provided.

The actual reduce work should start as all the partitions are fetched. However, in order to significantly reduce the IO overhead of reading and writing the intermediate results from mapper, the reducer task get the intermediate partition once any of the mapper completes. Every time the map task completes, the master node will notify all the reducer tasks to fetch the corresponding results. The reducer task will keep track of how many partitions they have and once they get all the results, it will automatically start the reducer work.

VI. Demonstration of example

1. Deploy the System

Before starting running the whole system, the administrator should first provide the configuration file, which consists of configuration file for job client, master node and worker node.

The job.conf file is for client to set up and it consists of the master's service IP, master's service port, the file path of mapper method, reducer method, input file and final result, partition size, job client's IP and interval of job status checking.

The mapreduce.conf file is for master node and consists of service IP address of master, service port of JobTracker module, which responsible for track all jobs and communicate with the job client, service port of Monitor module, which responsible for monitoring the framework's status through heartbeat, the interval of heartbeat.

The slave.conf file should be set on all the worker nodes. It consists of master's service IP, master's service port, master's service name, the file path of intermediate file, partition file and result file of reduce task, the resource of worker, including the number of mapper slot and reducer slot, interval of heartbeat and service name of worker.

After setting the proper configuration file, start a Java process for each participating node. First, start the master node by execute the JobTracker class:

```
java JobTracker
```

Then for each worker node, execute the TaskTracker class on each worker:

```
java TaskTracker
```

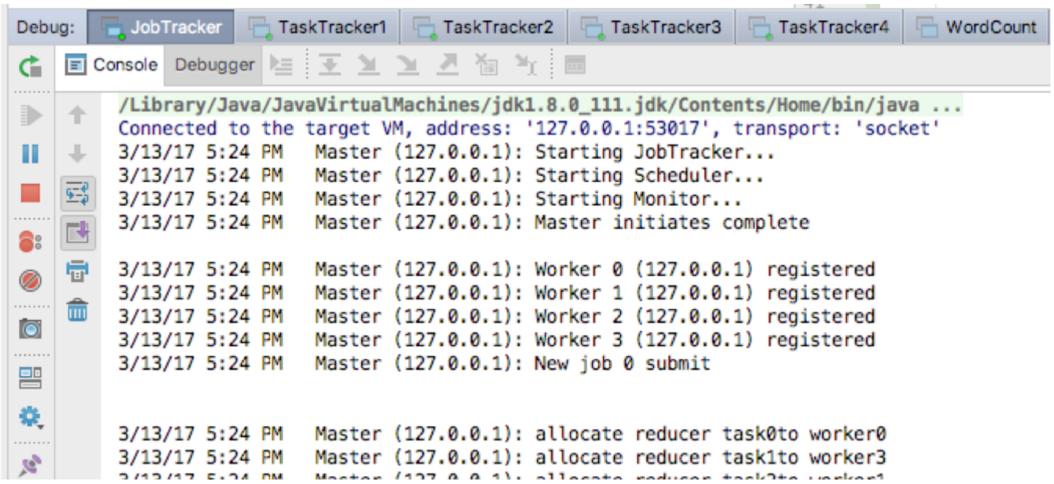
At last, start the job client class.

2. Word Count

After setting up the environment and corresponding file directory paths, an example of executing a Word Count task is shown here. And in this example,there are one master node and 4 worker nodes in the system, the Java RMI registry information of different nodes is set as below:

```
master 127.0.0.1 JobTracker 9000, Monitor 8000
worker 0 127.0.0.1 5000
worker 1 127.0.0.1 5001
worker 2 127.0.0.1 5002
worker 3 127.0.0.1 5003
```

For each node, we start a Java process and run the JobTracker class, TaskThread class. After the master node is initiated, all the worker will first register itself to master. Then run the WordCount class in job client to submit the job. Figure 3 shows that master node is initiated successfully and all 4 worker nodes also registered to master. Then the job client submitted job 0 to master and master started to allocating tasks to multiple worker nodes.



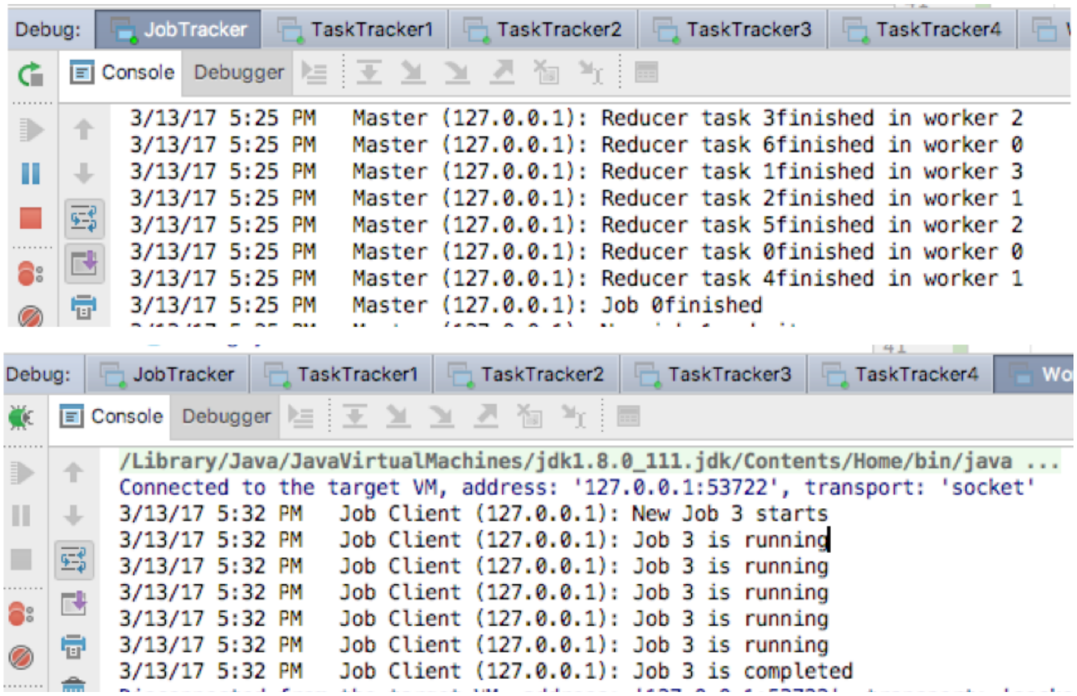
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:53017', transport: 'socket'
3/13/17 5:24 PM   Master (127.0.0.1): Starting JobTracker...
3/13/17 5:24 PM   Master (127.0.0.1): Starting Scheduler...
3/13/17 5:24 PM   Master (127.0.0.1): Starting Monitor...
3/13/17 5:24 PM   Master (127.0.0.1): Master initiates complete

3/13/17 5:24 PM   Master (127.0.0.1): Worker 0 (127.0.0.1) registered
3/13/17 5:24 PM   Master (127.0.0.1): Worker 1 (127.0.0.1) registered
3/13/17 5:24 PM   Master (127.0.0.1): Worker 2 (127.0.0.1) registered
3/13/17 5:24 PM   Master (127.0.0.1): Worker 3 (127.0.0.1) registered
3/13/17 5:24 PM   Master (127.0.0.1): New job 0 submit


3/13/17 5:24 PM   Master (127.0.0.1): allocate reducer task0to worker0
3/13/17 5:24 PM   Master (127.0.0.1): allocate reducer task1to worker3
3/13/17 5:24 PM   Master (127.0.0.1): allocate reducer task2to worker1
```

Figure 3. Node initiation and job submitted

As the task get executed, master would get notification about the completion of individual tasks from worker nodes. And after the whole job finished, master node would update the status that this job finished. Form the job client side, the client would also get notified that the job finished.



```
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 3finished in worker 2
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 6finished in worker 0
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 1finished in worker 3
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 2finished in worker 1
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 5finished in worker 2
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 0finished in worker 0
3/13/17 5:25 PM   Master (127.0.0.1): Reducer task 4finished in worker 1
3/13/17 5:25 PM   Master (127.0.0.1): Job 0finished
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:53722', transport: 'socket'
3/13/17 5:32 PM   Job Client (127.0.0.1): New Job 3 starts
3/13/17 5:32 PM   Job Client (127.0.0.1): Job 3 is running
3/13/17 5:32 PM   Job Client (127.0.0.1): Job 3 is running
3/13/17 5:32 PM   Job Client (127.0.0.1): Job 3 is running
3/13/17 5:32 PM   Job Client (127.0.0.1): Job 3 is running
3/13/17 5:32 PM   Job Client (127.0.0.1): Job 3 is running
3/13/17 5:32 PM   Job Client (127.0.0.1): Job 3 is completed
```

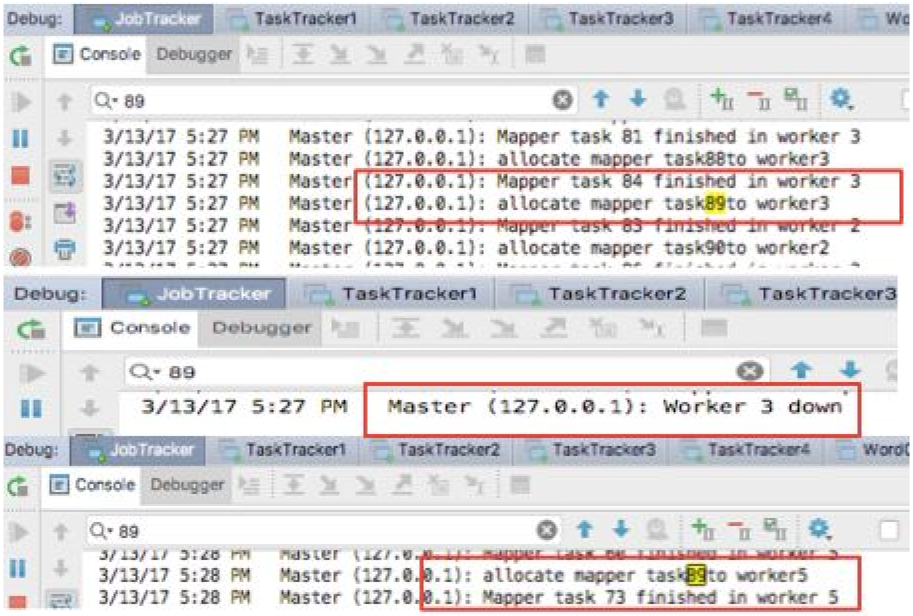
Figure 4. Job completed

After the execution of the text file, the intermediate file for reducer tasks is stored in the path and the final result is stored in "./result/finalResult/" as set up in the configuration file.

VII. Performance Testing

1. Testing of Worker Failure

Once the Monitor cannot receive the heartbeat from a specific worker for consecutive several times, the monitor will consider this worker as failed, and stops allocating new tasks to this node and also reallocate the tasks still running on this worker. From the testing result, we can see that we first terminate the worker 3 node. After the master node detected that the worker 3 fails, it would check the tasks that still running on worker 3 and reallocate the tasks to other available worker nodes. In this case, the master reallocate the task 89 of worker 3 to worker 5.



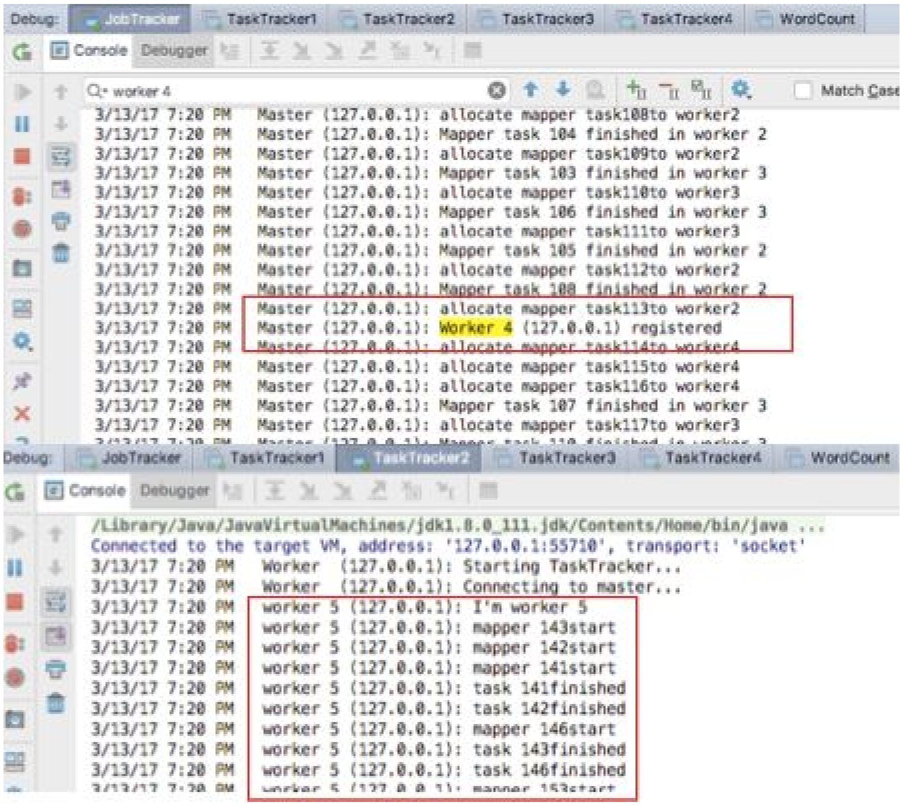
```
3/13/17 5:27 PM   Master (127.0.0.1): Mapper task 81 finished in worker 3
3/13/17 5:27 PM   Master (127.0.0.1): allocate mapper task88to worker3
3/13/17 5:27 PM   Master (127.0.0.1): Mapper task 84 finished in worker 3
3/13/17 5:27 PM   Master (127.0.0.1): allocate mapper task89to worker3
3/13/17 5:27 PM   Master (127.0.0.1): Mapper task 83 finished in worker 2
3/13/17 5:27 PM   Master (127.0.0.1): allocate mapper task90to worker2
```

```
3/13/17 5:27 PM   Master (127.0.0.1): Worker 3 down
```

```
3/13/17 5:28 PM   Master (127.0.0.1): allocate mapper task89to worker5
3/13/17 5:28 PM   Master (127.0.0.1): Mapper task 73 finished in worker 5
```

Figure 5. Worker Failed

As we restart the worker , the master would consider this restarted worker node as a newly added worker and then allocating jobs to the new worker node. As Figure 6 showed, the two restarted worker now registered as worker 4 and worker 5.



```
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task108to worker2
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 104 finished in worker 2
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task109to worker2
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 103 finished in worker 3
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task110to worker3
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 106 finished in worker 3
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task111to worker3
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 105 finished in worker 2
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task112to worker2
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 108 finished in worker 2
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task113to worker2
3/13/17 7:20 PM   Master (127.0.0.1): Worker 4 (127.0.0.1) registered
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task114to worker4
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task115to worker4
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task116to worker4
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 107 finished in worker 3
3/13/17 7:20 PM   Master (127.0.0.1): allocate mapper task117to worker3
3/13/17 7:20 PM   Master (127.0.0.1): Mapper task 110 finished in worker 3
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:55710', transport: 'socket'
3/13/17 7:20 PM   Worker  (127.0.0.1): Starting TaskTracker...
3/13/17 7:20 PM   Worker  (127.0.0.1): Connecting to master...
3/13/17 7:20 PM   worker 5 (127.0.0.1): I'm worker 5
3/13/17 7:20 PM   worker 5 (127.0.0.1): mapper 143start
3/13/17 7:20 PM   worker 5 (127.0.0.1): mapper 142start
3/13/17 7:20 PM   worker 5 (127.0.0.1): mapper 141start
3/13/17 7:20 PM   worker 5 (127.0.0.1): task 141finished
3/13/17 7:20 PM   worker 5 (127.0.0.1): task 142finished
3/13/17 7:20 PM   worker 5 (127.0.0.1): mapper 146start
3/13/17 7:20 PM   worker 5 (127.0.0.1): task 143finished
3/13/17 7:20 PM   worker 5 (127.0.0.1): task 146finished
3/13/17 7:20 PM   worker 5 (127.0.0.1): mapper 153start
```

Figure 6. Worker restarted

2. Execution time of different number of worker nodes

Theoretically, it is obvious that the execution time of a specific job should decrease as more worker nodes in this whole system. We also testing the execution time of same wordcount job based on different number of worker nodes in the system. From figure 7, we can see that the execution time decreases roughly the same speed as we increase the number of worker nodes to execute the job. Since our work count example is of small workload, the execution time becomes quite stable as the number of worker nodes increased to 8.

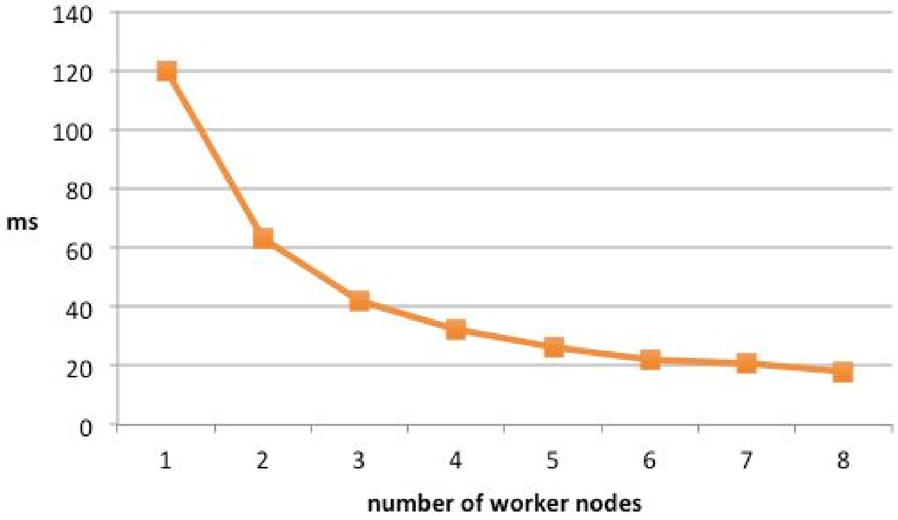


Figure 7. Execution time of different number of worker

VIII. Summary and Future Work

In this project, we build a MapReduce framework by Java from scratch. We introduced the implementation and major design decisions about how we build this MapReduce framework. For simplicity, we have made some assumptions and tradeoffs in our implementation. Our system provided APIs for clients to submit the files and methods to be executed and also supports executing multiple job concurrently. All the communication among participating nodes is realized by Java RMI. Then we give the instruction about how to deploy the system and an example of running Word Count using this framework. Then we also provided the testing results of worker failure and recovery.

Still, there are some aspects that we could improve in our future research and work.

1. Handle more robust fault tolerance mechanism

In this project, we consider the worker node has only two states: run or crash. So we only check the worker status by heartbeat, which can only find worker crash or network connection failure.

Actually, there are several other possibilities in practice, such as disk crash, which can cause task failure with normal heartbeat. In future work, we should improve the worker status check mechanism. Worker should report more kinds of error of its own, like disk failure, to master, and master should decide how to address these failures according to the exact level of failures, such as kick off the worker which can not work normally because of IO failure.

2. Managing jobs by client

Currently, the client only interact with the master node by submitting the jobs. We could provide more APIs for clients to manage all the jobs they submitted. For example, the client could stop or terminate specific job by specific command. Also, our framework can provide information about the running status of a job so that the client can visualize the percentage of completion of the current job.

3. Improve load balancing

In our implementation, the master allocate the jobs to different worker nodes by selecting the worker with most available slots. However, other information such as network condition, RAM usage percentage, and where is the file is stored should also be considered in a practical MapReduce framework in order to achieve better performance. In the future, we could take more metrics into consideration to improve the load balancing. Also, we could take some scheduling algorithms here among jobs to avoid that some jobs may be encounter longer response time.

4. Master failure

This project only implement worker node failover. Actually, we can implement some mechanism to support master failover. The paper, "MapReduce.", mentions that it can partially support master failover through checkpoint mechanism. Master can store checkpoint into disk periodically. After restoring from crashing, master can restore the MapReduce process through the checkpoint data stored in disk.

Reference

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce." *Communications of the ACM* 51.1 (2008): 107.
[2] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce." *Communications of the ACM* 53.1 (2010): 72.