15-640 Distributed System - Project 3

Huanchen Zhang (huanchenz), Mengwei Ding (mengweid)

1. System Design

In this section, we introduce our whole system in detail in two perspectives: the Map/Reduce paradigm and the job and task managing.

a) Job and Task Management

The architecture of our whole system is shown in Figure 1, which is largely inspired by the design of Hadoop. We adopt many design decisions from Hadoop, but at the mean time, we try to make our own innovations.

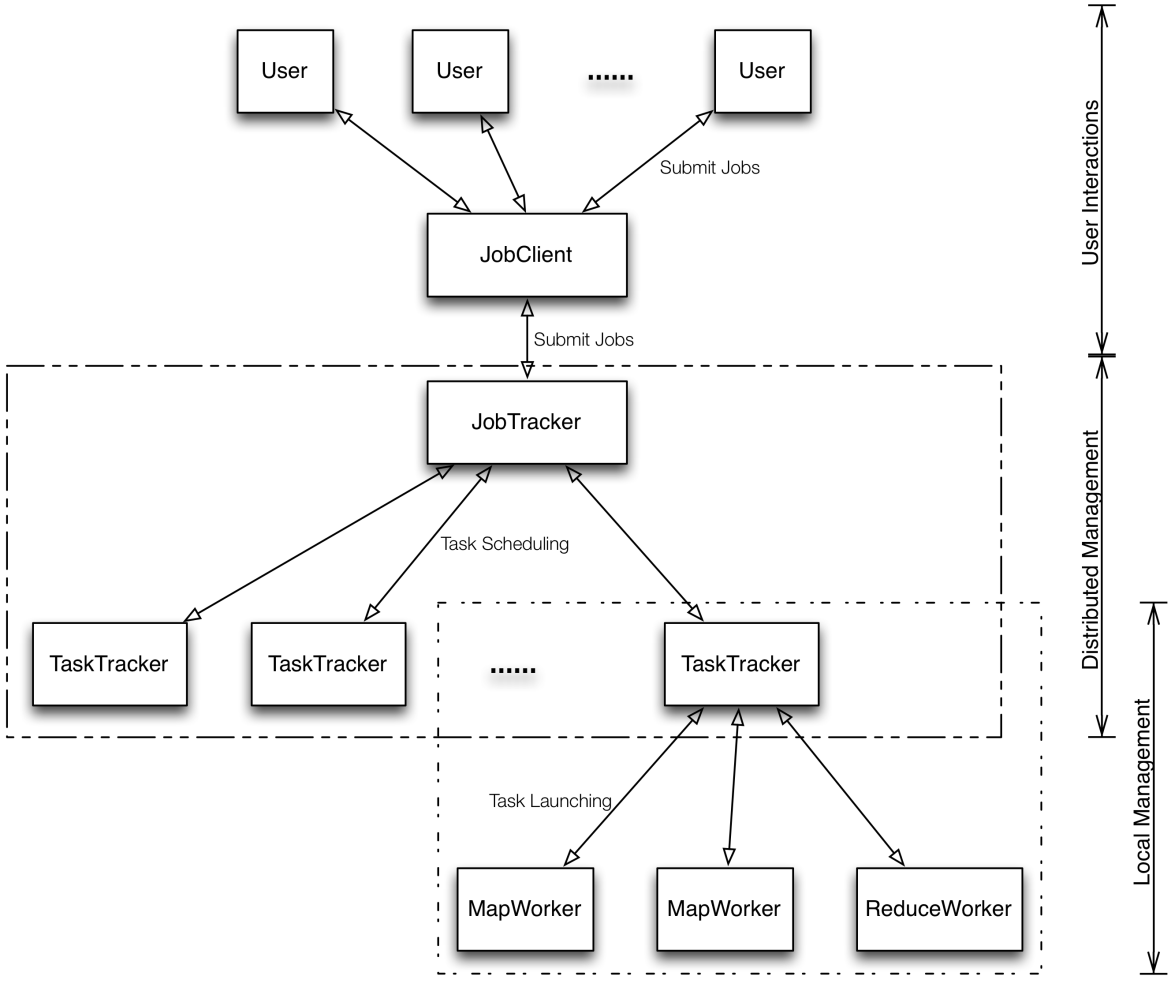

Figure 1 The architecture of our whole map-reduce system

As we could see from this figure, there are mainly three parts in this system: Distributed Management, Local Management and User Interaction.

Either the Distributed Management or Local Management parts could be regarded as a two-level Server-Client model. The TaskTracker could be treated as either Server or Client in different parts.

- In the User Interaction part, things there are quite simple. User could submit their own Map/Reduce job to JobClient after configuring and packing up all their code and data. Then the JobClient could unpack all the code and data to the right place, which is visible to our system. Next, JobClient informs the JobTracker that a new job comes and everything is well prepared for it so that JobTracker could execute that job instantly. To be more specific, JobClient could be regarded as an agent of JobTracker for submitting jobs, so it generally locates along with JobTracker. We also provide a tool (Job) that could automatically connect to the remote JobClient and submit the job. So whenever the user is, he could submit the job to the system through JobClient. The code of the user's job should be packed into a jar file. After the job being submitted, the system would extract all .class files from the jar archive as the way to prepare the code.

- In the Distributed Management part, JobTracker plays the role as the Sever, while TaskTracker as the Client. JobTracker knows the existences of all TaskTrackers and all their profiles, such as what's the name of this TaskTracker, which tasks are now running on it, what's the progress of these tasks and how many available working slots available on it, etc. To keep their profiles in JobTracker up-to-date, TaskTrackers need to periodically (2 seconds defaultly) send heartbeat message to JobTracker and make update to their profile. And if the JobTracker does not receive the heartbeat message from a TaskTracker for a very long time (8 seconds defaultly), it would assume that the TaskTracker dies, and would of course reassign the tasks running on that TaskTracker to another ones.

After a job has been submitted to the JobTracker, it would be split into many small Mapper and Reducer tasks. The number of Mappers is decided by the size of each input data split, which is specified by the user himself. And the number of Reducers is directly specified by the user himself in the job configuration. The JobTracker has 2 queues, which holds Mapper and Reducer tasks, respectively. Both the queues are priority queues, in which the lower the job id of the task, the higher the priority is, in which way the system tries its best to finish the job that comes the first as soon as possible. After the tasks of a job has been created and added to the system, the next thing is how to make arrangement and schedule for these tasks to run. Currently, the default task scheduler would go through all TaskTrackers and fill up a TaskTracker as many as possible, whenever there are available slots on that TaskTracker.

- In the Local Management part, TaskTracker plays the role as the Server, while each Map/Reduce Worker as the Client. Right after the TaskTracker receives the request

from the JobTracker to start a new Task, it would initiate a new Java process as the Worker with the entire context for a process, like the input split, output path and the mapper or reducer classes. To keep the Mapper and Reducer tasks in track, these tasks also need to send heartbeats to TaskTracker to report their status periodically. Also, like the JobTracker, the TaskTracker would also assume that a Worker fails if it does not receive a heartbeat message from this Worker for a fixed period of time.

This 2-level hierarchy is not always strictly organized. For a Reducer Worker, it needs to wait for all mappers of that job to finish until it moves forward to the sort and reduce phases. However, the TaskTracker does not know even a thing about a job, so the Reducer Worker has to acquire the job's status directly from the JobTracker.

- For a distributed system, fault tolerance should be considered for sure. The failure could happen to either a TaskTracker or a Mapper/Reducer Worker. Our system handles both these 2 situations. When the JobTracker detects that one TaskTracker is lost, it would immediately retrieve those tasks that running on that TaskTracker, and place them to the task queues again. This is easy for JobTracker, since it maintains every piece of information about a TaskTracker. When a Mapper/Reducer Worker dies, the TaskTracker would first know this, and then report to the JobTracker. The JobTracker would insert this task into the queue again, and in this way this task would be executed in the future in any possible TaskTracker. However, each task only has limited opportunities (4 by default). For each task, it has a counter called "Attempts", which means how many times this task has been tried to executed. After this counter reaches 4, the JobTracker would treat the task as failed. Then it would treat the whole job as failed and remove all unscheduled tasks away from the queue. By the way, the JobTracker could also crash down somehow, but this could hardly be handled except restarting the whole system.

b) Map/Reduce Workflow

Our system tries out best to provide the highest flexibility to allow programmers customizing their Map/Reduce workflow. Our framework could support all components that appear in Hadoop except the Combiner.

Basically, the programmers are required to provide a Mapper and a Reducer. They should at least implement the map() and reduce() function. Also, our framework provide a setup() and a cleanup() function which are call right after the task begins and right before the task ends, respectively, because in some cases, programmers might wish to do some preprocessing jobs or cleanup jobs.

While it is assumed that all the data has already store in the AFS, the Reducer could directly access all the temporary output files of Mapper. Therefore, there is no "copy" phase in our Reducer. Also, the Reducer combines the "sort" and "reduce" phases together that the reducer could fetch next key/value pair on-demand from the sort phase. Since the output files of Mapper have already been sorted in terms of Key, our system

uses a K-way merge algorithm to feed the reducer with key/value pair. The K-way merge algorithm would only move a step forward and read a key/value pair from a file only when the task requests the next key/value pair. In this way, we do not need to load all unsorted data into memory, which might be inappropriate when the data is huge.

The system also provides facilities and interface for I/O operations.

To read and structurize the input data, the programmer could also specify his own InputFormat, which enables the framework to recognize the input data format and parse the data into key/value pairs. Our framework provides a default InputFormat class, which treats each line of input data as a key/value pair.

Similarly, to make the output data organized, the programmer could create his own OutputFormat, which helps the framework format key/value pairs into raw data. Our framework also provides a default OutputFormat class, which output each key/value pair as a tab separated line into the file.

Our framework even provides the freedoms to customize the way to mapping output key/value pairs from Mapper to Reducer, by implementing the Partitioner abstract class.

c) System Monitor

Our system provides a tool to monitor the whole system at the JobTracker side. The user could list the status of current system, such as the TaskTracker status, the job list, and the task list of a specific job.

d) Network Communication

In this project, instead of using the traditional socket programming to build the network communication part, we use the Java Remote Method Invocation (RMI) to do message passing. The original motivation is applying what we have just learned in previous project into practice. And the good thing of using RMI is that it saves us a lot of time of spawning extra threads to do socket communication, open or close socket.

2. System Deployment

Almost all parameters can be configured in the configuration file (config/config file under the installation directory). Following is the table of parameters descriptions.

Parameter	**Description**
JOB_TRACKER_REGISTRY_HOST	RMI registry hostname of job tracker
CLIENT_HOST	RMI registry hostname of client

REGISTRY_PORT	Port number of all RMI registries
TASK_TRACKER_1_NAME	1st task tracker name
TASK_TRACKER_1_NUM_MAPPER	Num of mapper slots on 1st task tracker
TASK_TRACKER_1_NUM_REDUCER	Num of reducer slots on 1st task tracker
JOB_TRACKER_SERVICE_NAME	The RMI service name of job tracker
CLIENT_SERVICE_NAME	The RMI service name of client
HEART_BEAT_PERIOD	Time period between heart beat in seconds
MAPPER_STANDARD_OUT_REDIRECT	File path of standard output of mapper tasks (for user use)
MAPPER_STANDARD_ERR_REDIRECT	File path standard error of mapper tasks (for user use)
REDUCER_STANDARD_OUT_REDIRECT	File path of standard output of reducer tasks (for user use)
REDUCER_STANDARD_ERR_REDIRECT	File path standard error of reducer tasks (for user use)
ALIVE_CYCLE	Longest time period between heart beats before being detected down
REDUCER_CHECK_MAPPER_CYCLE	Time period between reducer's request for mapper progress information
THREAD_POOL_SIZE	Thread pool size for scheduled executor
SYSTEM_TEMP_DIR	Temporary file directory
USER_CLASS_PATH	Directory to un-jar user's jar file

Since our system will run on the AFS, the installation is quite simple. Just extract and copy the whole system's working directory to the place you want. Then to start the jobtracker, tasktracker or jobclient, just switch to different machine nodes, which could all access the system's working directory.

When deploying the system, the user can use the default parameter values in the default configuration file. For the user's convenience, a make file is also provided in the project root directory. The user should follow instructions as follows.

- Compile the source code first, by the following command.

```
make compile
```

- Archive all the code of our system into jar file, by the following command.

```
make jar
```

- For each machine in the cluster, first start one RMI registry using following command.

```
make rmi PORT=1234
```

The PORT should be consistent with the REGISTRY_PORT in the configuration file.

- Start job tracker using following command.

```
make jobtracker
```

- Start client using following command.

```
make client
```

One thing to remember is that the machine running the jobclient should be identical to the value of CLIENT_HOST in the configuration file.

- Start task tracker using following command.

```
make tasktracker SEQ=1
```

SEQ is the id (or sequence) of task tracker. For example, if there are three task tracker on three machines, type ‘make tasktracker SEQ=1’, ‘make tasktracker SEQ=2’, ‘make tasktracker SEQ=3’ on each machine. Also, add corresponding configure parameters in the configuration file, ‘TASK_TRACKER_1_NAME’, ‘TASK_TRACKER_1_NUM_MAPPER’, ‘TASK_TRACKER_1_NUM_REDUCER’ etc.

- To test the word count program, use following command

```
make wordcount
```

The input data is in ‘input’, and the output of reducers is in ‘output’.

- To test the degree count program, use following command

```
make degreecount
```

The input data is in ‘input’, and the output of reducers is in ‘output’.

3. Programming Manual

a) Map

The interface of the Map is shown in the following figure:

```
3  public abstract class Mapper {
4
5    /**
6     * the method to do some initialization work. The method
7     * would only be called once at the very beginning of this
8     * task
9     */
10   protected void setup() {
11
12   }
13
14   /**
15    * the method to do some cleaning job. The method would only
16    * be called once at the very end of this task
17    */
18   protected void cleanup() {
19
20   }
21
22   public abstract void map(String key, String value, Outputer out);
23 }
```

The map() function is required to be implemented. The outputer is used to collect the result of the map() function. And Map also provides 2 optional functions: setup() and cleanup(). The setup() function will be called at the very first beginning of the mapper task, designed to do some initialization work. The cleanup() function will be called at the very end of the mapper, designed to do some cleaning job. A typical implementation of Map is shown in the following figure:

```
6  public class WCMapper extends Mapper {
7
8    @Override
9    public void map(String key, String value, Outputer out) {
10     String line = value;
11     String[] words = line.split(" ");
12
13     for (String word : words) {
14       out.collect(word, Long.toString(1));
15     }
16   }
17
18 }
```

b) Reduce

The interface of the Reduce is shown in the following figure:

```
5  public abstract class Reducer {
6    /**
7     * the method to do some initialization work. The method
8     * would only be called once at the very beginning of this
9     * task
10    */
11   protected void setup() {
12
13   }
14
15   /**
16    * the method to do some cleaning job. The method would only
17    * be called once at the very end of this task
18    */
19   protected void cleanup() {
20
21   }
22
23   public abstract void reduce(String key, Iterator<String> values, Outputer out);
24 }
```

The reduce() function is required to be implemented. The outputer is used to collect the result of the reduce() function. Like Map, Reduce also provides 2 optional functions: setup() and cleanup(). The setup() function will be called at the very first beginning of the reducer task, designed to do some initialization work. The cleanup() function will be called at the very end of the reducer, designed to do some cleaning job. A typical implementation of Reduce is shown in the following figure:

```
8  public class WCReducer extends Reducer {
9
10   @Override
11   public void reduce(String key, Iterator<String> values, Outputer out) {
12     long sum = 0;
13
14    while(values.hasNext()) {
15       sum += Long.parseLong(values.next());
16     }
17
18     out.collect(key, Long.toString(sum));
19   }
20
21 }
22
```

c) InputFormat

The InputFormat in our system is actually an Iterator. The programmer needs to implement the hasNext() and next() functions. The InputFormat provides the file operator to manipulate the I/O. A typical implementation of InputFormat is shown in the following figure:

```java
public class WCInputFormat extends InputFormat {

  public WCInputFormat(String filename, Long offset, Integer blockSize) throws IOException {
    super(filename, offset, blockSize);
  }

  @Override
  public boolean hasNext() {
    /* check if the block ends or not */
    try {
      return this.hasByte();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return false;
  }

  @Override
  public Record next() {
    try {
      /* read next line */
      String line = this.raf.readLine();

      String key = Integer.toString(line.length());
      String value = line;
      /* return a record built with the key and value */
      return new Record(key, value);
      // return new Record(line,line);
    } catch (IOException e) {
      e.printStackTrace();
    }
    return null;
  }
```

d) OutputFormat

The OutputFormat generally encode the key/value pair into String. To implement an OutputFormat, we only need to implement the format() function. A typical implementation is shown below which encode key/value into a line separated by a tab.

```java
public class WCOutputFormat  extends OutputFormat {

  @Override
  public String format(String key, String value) {
    return key + "\t" + value + "\n";
  }

}
```

e) Partitioner

The main job of Partitioner is computing the id of reduce task to which the given key should be assigned. To implement a Partitioner, getPartition() and getReducerNum() are required. A typical implementation is shown below:

```java
public class WCPartitioner implements Partitioner {
  private int reducerNum;

  public WCPartitioner(Integer reducerNum) {
    this.reducerNum = reducerNum;
  }

  public int getPartition(String key) {
    return Math.abs(key.hashCode()) % this.reducerNum;
  }

  public int getReducerNum() {
    return this.reducerNum;
  }
}

```

f) Main

After implementing all above necessary components for a job, we need to assemble all these components together by writing a JobConf. A typical setting is shown below:

```java
public static void main(String[] args) {
  if (args.length != 3) {
    System.out.println("Usage: wordcount <input_path> <output_path> <jar_path>");
    return ;
  }

  JobConf jconf = new JobConf();
  jconf.setJobName("WordCount");

  jconf.setInputPath(args[0]);
  jconf.setOutputPath(args[1]);
  jconf.setJarFilePath(args[2]);

  jconf.setBlockSize(100000);

  jconf.setMapperClassName("example.wordcount.WCMapper");
  jconf.setReducerClassName("example.wordcount.WCReducer");

  jconf.setInputFormatClassName("example.wordcount.WCInputFormat");
  jconf.setOutputFormatClassName("example.wordcount.WCOutputFormat");
  jconf.setPartitionerClassName("example.wordcount.WCPartitioner");

  jconf.setReducerNum(4);

  Job job = new Job(jconf);
  job.run();
}
```

First, you could give a name for this job, by the setJobName() function;

Then, specify the input, output and code file path by the setInputPath(), setOutputPath() and setJarFile() functions;

Next, specify the input split block size by setBlockSize() function, and the unit is byte;

Then, set the class names, including the package name, for Mapper, Reducer, InputFormat, OutputFormat and Partitioner.

Next, set the number of reducers by the setReducerNum() function.

After all of these steps, create a Job using this JobConf and run it.

g) Two Sample Jobs

We provide the following 2 sample MapReduce jobs for our system. We also provide the input data "patents.txt", which is a graph data of US patents.

- **WordCount**

Count the number of all unique words in the given input.

- **DegreeCount**

Count the in-degree and out-degree of all nodes in the given input graph.

4. Limitations

Considering that this is a course project and thus the time is really limited, as well as the fact that the complexity of such MapReduce system is huge, we have to make several compromises in our design, which results in the following limitations:

- Out system does not support generic Key/Value types. Currently, we assume that the Key and Value are both Strings. So the programmer might need to do some explicit type casting in the Mapper or Reducer. Hadoop builds an entire new type system, and we might do the same thing in the future to make our system more general.
- The current task scheduler of our system does not consider the load balance issue. The default scheduler of our system only tries to feed a TaskTracker as many tasks as possible at once. Since we have already leave the task scheduler interface open, it would be easy to implement smarter scheduler in the future.
- When a job failed, even though the system removes those tasks that are in the task queue, it could do nothing to those running tasks of this job, because currently our system does not build the procedure to allow JobTracker to inform those running task to stop.
- The percentage of completeness for each task is not very accurate currently. In current system, we just apply a very simple percentage calculation formula, which could only give a rough estimation.
- Hadoop provide many other mechanisms to make its system robust, such as task speculation scheduling, by which the system would initiate backup tasks for those ones that greatly lag behind. However, our system does not have these kinds of mechanisms so far.