15640 Project 3: Map Reduce

Yangdong Liao, yangdonl
Kuang-Huei Lee, kuanghul

This report contains 3 sections:

1. The first section describes the design of our MapReduce framework, the various tradeoffs and future improvements.

2. The second section is for the users: it explains how to define a custom job using our framework, the configurations required, and how to start and monitor a job

3. The last section will give the TA's a step-by-step guide of how to build, deploy, and run our demo examples.

Section 1 - Framework Overview

This section describes how components in our framework work together. This project can be described using **three** main components: the **Job,** the **Master**, and the **Participants**.

Component 1 - Job

The job is a group of abstract classes that the users have to extend to define a mapreduce job. There are **four** classes that an user will have to define, we'll be giving examples of how our sample job defines these classes in section 2:

1. **MapperOutput**

   This is the most flexible class, you can define any data structure or variables in it. This class represents the results that reducers expect to receive from mappers after their mapping step.

   The MapperOutput class contains a function called getKey, which returns a key used to assign a reducer.

2. **MapperFunction**

   This is an abstract functor class that is used by mappers. The user implements the map function, which takes a file block (a list of strings representing a subsection of a file) and returns a list of MapperOutputs. Each element in this list will be sent to different reducers based on their keys.

3. **ReducerFunction**

   This is abstract functor class that is used by reducers. The user implements the reduce function, which takes in a list of MapperOutputs, and generates an output file.

4. **KeyToReducerFunction**

   This is another abstract functor class that is used by mappers. It takes a String key from a MapperOutput, and it returns information about the reducer (host, port) that the output should be sent to.

The abstract **Job.java** class then aggregate all these classes into a single class: it contains a list of file blocks, and it contains functions to get instances of any of the functors above.

Component 2 - Master

The master serves as a central coordinator, it contains a **job** and connections to a list of **participants**, the participants consist of reducers and mappers. The purpose of the master is to assign each of the file blocks from the job to mappers, and after all mappings are complete, the master will tell the reducers to begin reducing.

This is achieved with two threads: **check alive** and **scheduler**.

Check Alive Thread

The check alive thread periodically sends check alive messages to the participants, and if confirmation is not received from a certain participant, a failure event would then be triggered.

If the failure source is a mapper M, the master knows which file blocks were assigned to M previously, these blocks would be marked as unassigned, and the scheduler will assign them to available mappers in the future.

If the failure source is a reducer R, what the master does is restart the whole job after removing R from the participants. There are two reasons that we chose this solution, and they both relate to simplicity:

1. If R died during the reduce phase, we do not have to worry about stopping another reducer R', finding all the mapper outputs O that R had, and sending O to R' before telling R' to restart the reduce phase.

2. The mappers would not have to store the results after they have been sent to reducers. They would not have to worry about the fact that the results may be used again in the future incase a reducer dies.

In the case that there are no mappers available after the failure, or no reducers available after the failure, the master stops the job and informs the user that the job cannot continue.

Scheduler Thread

The scheduler is responsible for assigning file blocks to mappers. This is done using a very simple heuristic: the scheduler stores an availability counter A_i for each mapper i, where A_i is initialized to the number of CPUs in mapper i's machine, this is read from our configuration file. Then the scheduler runs the following loop:

```
Find an unassigned block B
Find a mapper i with $A_i > 0$
Assign B to mapper i, decrement $A_i$ by 1

Whenever we receive a message from a mapper i saying that it has
finished processing its block, we increment $A_i$ by 1
```

The assignment is done through sending messages, which we will talk about in a later section.

Once all blocks have been processed, the master would then send messages to all reducers tell them to start reducing. The job ends when reducing finished messages have been received from all reducers.

Component 3 - Participants

There are two types of participants - Mappers and Reducers. Every participant runs a message receiving loop, and reacts depending on the messages that they receive.

Mapper

The mapper class is responsible for processing file blocks, a mapper instance does the following based on the messages that it receives :

- If it receives a “check alive” message, it returns a confirm alive message

- If it receives an “assign block” message (master is assigning it a file block to process), it starts a new thread and does the following:

  - Take the **MapperFunction F** within the message
  - Take the **FileBlock B** within the message
  - Take the **KeyToReducerFunction K** within the message
  - Get a list of **MapperOutputs O** by running F(B)
  - For each output in O, the output is sent to reducer **K(O.Key)**
  - Store the number of confirmations = length(O) required before able to notify that a block has been processed

- If it receives a “mapper output received” message (sent by reducers), it decrements the number of receives requires for that specified file block. If the result after subtraction is 0, the mapper has received all confirmations, and can send a message to master saying that it has finished mapping a block and that the reducers have received the results

Reducer

The reducer class is responsible for processing the mapper outputs, a reducer instance does the following based on the messages that it receives :

- If it receives a “check alive” message, it returns a confirm alive message

- If it receives a “receive mapper output” message (sent from mappers), it adds the output taken from the message into a list of outputs. The reducer contains a list of outputs for each job ID to allow concurrent jobs.

- If it receives a “start reducing” message (sent from master), it starts a new thread and does the following:

    - Take the **ReducerFunction F** within the message
    - Find the list of MapperOutputs **O** corresponding to the job ID of the message
    - Run reduce using **F(O)**, the function F will handle writing output to file
    - Notify the master that it has finished reducing

- If it receives a “reset” message, it knows that a reducer has died and the master is restarting the job. It then clears the mapper output list corresponding to the job ID to avoid duplicate outputs.

Messages

Communications within the system are done using messages, all messages carry a **sender ID** and a **job ID** to identify their associations. Here we explain the purpose of each of our message types:

Check Alive: Sent from master to participants to ensure that they are reachable

Confirm Alive: Sent from participants to master to confirm that they are reachable

Assign Block: Sent from master to mappers to assign a file block for mapping. Contains the file block (our means of replication), the file block ID, a MapperFunction functor, and a KeyToReducerFunction functor.

Receive Mapper Output: Sent from mappers to reducers, contains a MapperOutput for a file block in a job

Mapper Output Received: Sent from reducers to mappers, telling the mapper that the output for the specified file block has been received.

Mapper Finished: Sent from mappers to master specifying that it has finished processing a file block. This message is sent after the mapper has received "Mapper Output Received" confirmations from all reducers associated with the file block.

Begin Reduce: Sent from master to reducers telling them to begin reducing. This is sent after the master has received "Mapper Finished" for all file blocks. This message contains a ReducerFunction functor for the reducer to use. It also contains a boolean flag specifying whether we want our output to be sorted.

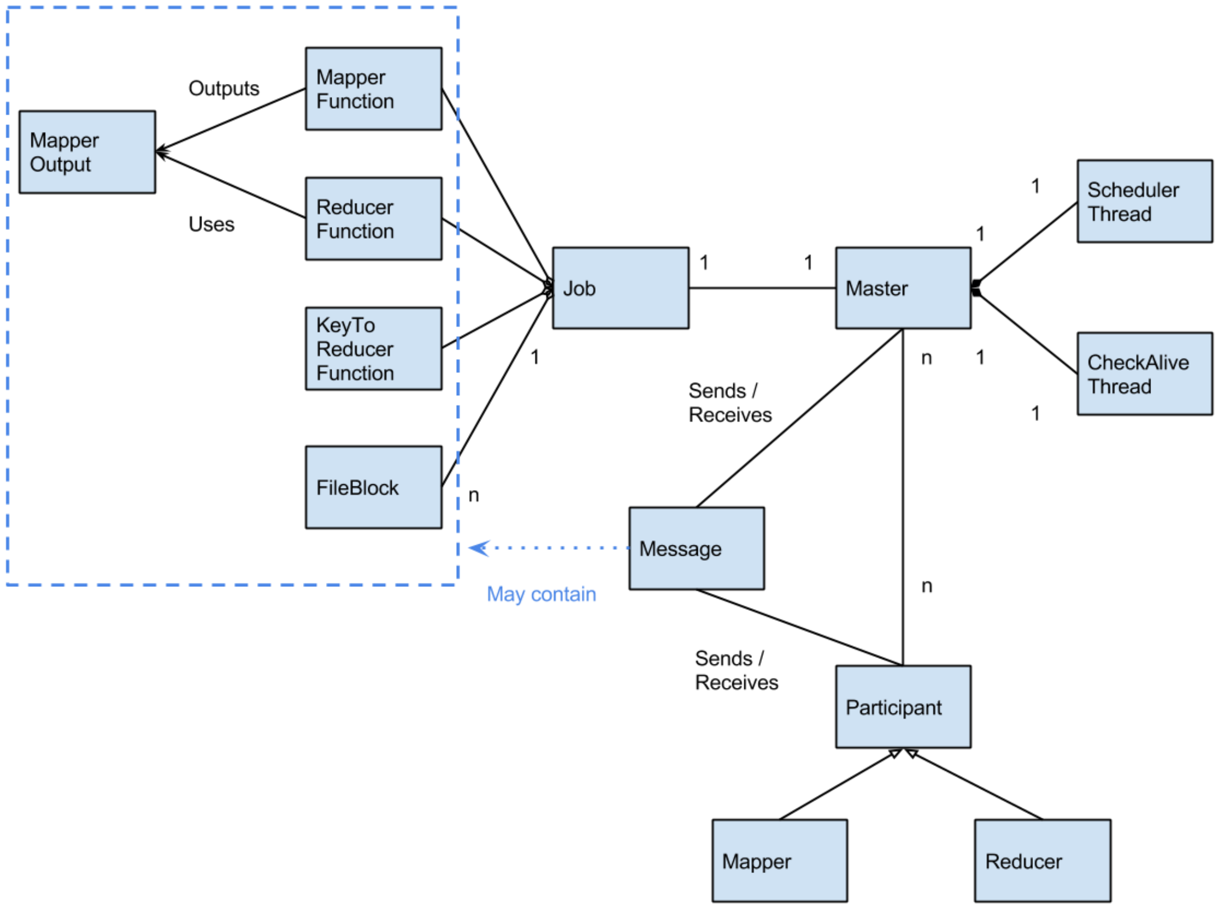
Reduce Finished: Sent from reducers to master specifying that reducing is complete. Once the master has received this message from all reducers, the job is complete.

Reset Job: Sent from master to all participants when a job is restarted (this happens when a reducer dies). This message tell the participants to clear any existing data associated with the job to prevent inconsistencies (i.e. if a reducer does not clear the list of mapper outputs from the previous job, the list will contain duplicate data when the job restarts).

Stop: Sent from master to participants, used as a part of the monitoring tool to kill participants.

Class diagram

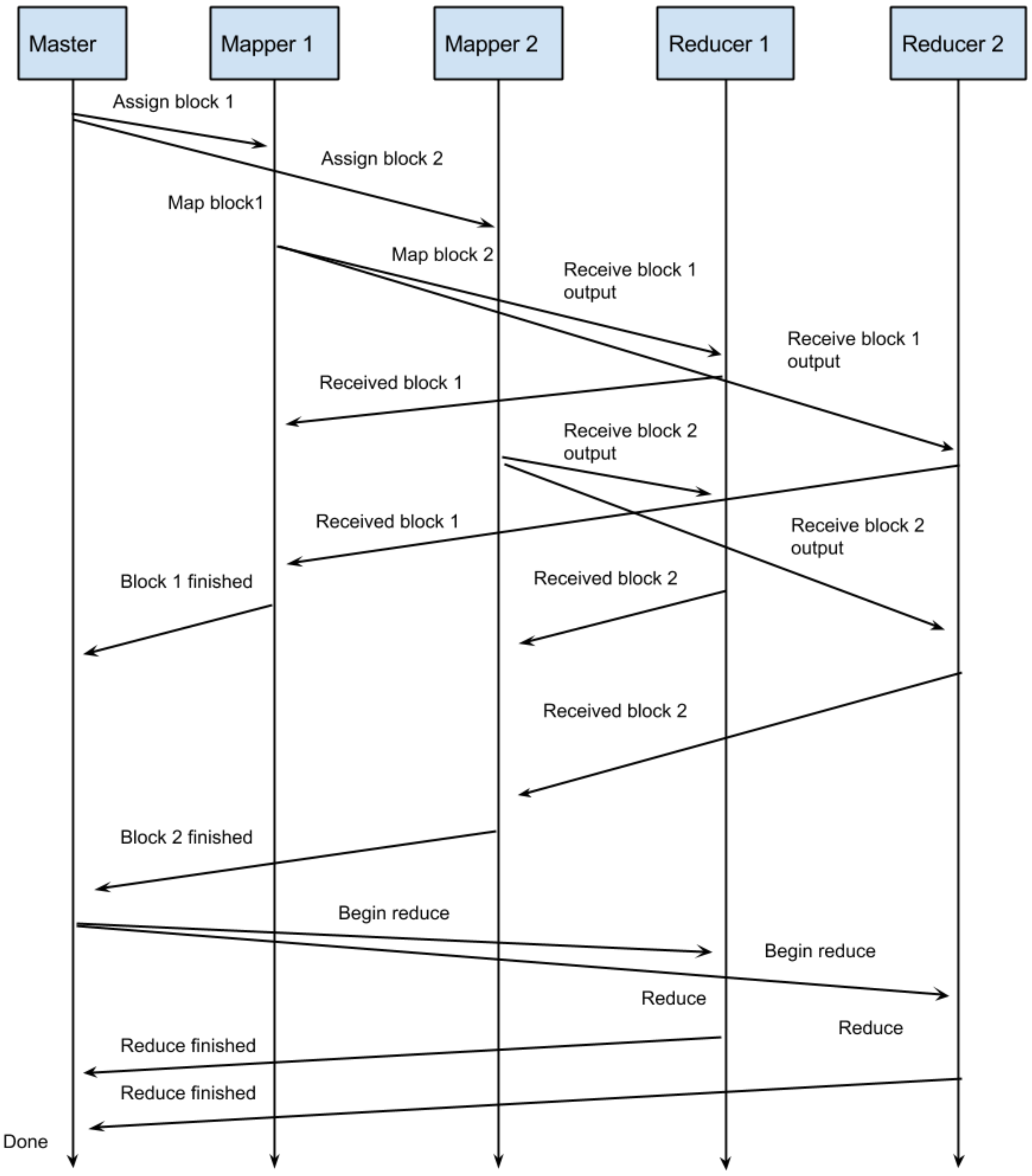
Here is the class diagram of our project, it is simplified by omitting the various Message subclasses:



Message flow

Here is a message flow diagram of a sample job, assume our job contains 2 file blocks.

Note the message receiving / sending orders may differ between mappers and reducers during runtime depending on how long processing takes, this is just one potential event ordering. The check alive / confirm alive messages are also omitted for simplicity:



Tradeoffs

Here is a list of major decisions and their tradeoffs:

- **Sending functors rather than using RMI**
  By sending functors (MapperFunction, ReducerFunction, KeyToReducerFunction) from master to participants for mapping and reducing, we get portability and availability.

  A big advantage of this is that mappers and reducers **do not have to be rebuilt and redeployed** whenever there are new mapping, reducing, or hashing functions. Only code on the master node have to be rebuilt, then mappers and reducers would be able to use the new functions from the messages because everything conforms to a predefined interface (the abstract job classes).

- **Restarting the job if a reducer dies**
  We restart the job when we detect that a reducer failed due to simplicity. One reason is that mappers do not have to store their outputs after they have been sent to a reducer, because there isn’t the case that a mapper have to find the outputs that were sent to the failed reducer, and send them to another available reducer. The advantage is that the program is simpler, there are less intermediate storage. The disadvantage is that there would be redundant processing from restarting the job.

  Another advantage is that if a reducer R failed during the reduce phase, we would not need to worry about stopping another reducer R’, find and send the values that R had to R’, and then having R’ to restart the reduce phase.

- **Not restarting the job if a mapper dies**
  We do not restart the job when we detect that a mapper failed for efficiency. Since the master knows which file blocks that each mapper was responsible for, it is easy to mark those file blocks as unassigned, and the scheduler would pick this information up and assign the blocks to other mappers.

- **Hashing by key instead of by file block**
  We originally planned to select reducer using hash by file block rather than key. The downside of hashing by file block is that in a multiple reducer case, with a word count job, any reducer may contain only a partial word count. To get the total count of each word, we would need another phase of mapreduce using a single reducer. However, if we did do hashing by block, it would make handling reducer failures much simpler: we would not have to restart the whole job, rather we can just assign the file blocks that the failed reducer was responsible for to another reducer, and remap those blocks.

Replication Design / Tradeoff

We assume that the file is located originally on the master node.

Replication is done by sending each of the file blocks (subsections of the file) to mappers in “assign block” messages. These message also contain the functor MappingFunction for the mappers to use.

We do not replicate file blocks to a mapper unless it is assigned to that mapper. This reduces traffic. However, if the master fails, we would not be able to access any of the unassigned(unreplicated) file blocks.

We believe that this is okay for two reasons:

1. Failure (in this case on the master) is rare

2. Since the master is the coordinator, it contains the scheduler. Even if we can retrieve various replicated file blocks after the master fails, they won’t be very useful with the scheduler not running. While it is true that we might be able to replicate the scheduler / job status across the participant to handle this, it would require a lot more implementation work.

Furthermore, it would not be hard for us to implement replication of file block from one participant to another, our master already keeps track of which participant has which blocks. If we want, we can have the master send a replication message to a participant to tell it to replicate a file block to another participant.

Requirements Met/Not Met, Future Improvements

We met all the core requirements for the Map Reduce framework:

- Users are able define custom mapreduce jobs
- These jobs can be carried out in parallel using any number of mappers and reducers defined by a configuration file
- Mapper and reducer failures are handled
- We provide tools(commands) for monitoring a job
- The framework is able to support concurrent jobs (see our test case later)
- We provide I/O in terms of writing output to file
- We try to optimize by dividing work based on the number of CPUs available

One thing that we can improve upon is how we handle reducer failures; right now we restart the whole job when we detect that a reducer fails, this means that all the work that mappers have done will have to be done again.

We can avoid this redundant work by doing the following: have mappers store their output to intermediate files. When a reducer fails, we find all outputs that the reducer had by looking through these intermediate files, and have the mappers send them to another available reducer. If failure occurs during the reduce step, we can stop another available reducer X, send X outputs that the failed reducer had, and tell X to restart its reduce step.

Another thing that we can improve upon is taking advantage of replication: if a particular mapper M dies. Assume that M’s file blocks were replicated on another mapper M’, we can have the master tell M’ to remap those blocks without sending the file blocks to M’.

Section 2 - Running a Custom Job

This section describes how an user can define a custom job and run it within our framework, it also describes how to configure the machines, deploy them, and then start/monitor a job:

How to implement the job package

To define a custom job, a programmer using our framework has to extend the abstract classes in the “`Job`” package. The purpose of various classes are defined in page 2, here are examples of how our two example classes implements the Job packs:

WordCount Job:

1. **WordCountMapperOutput** extends MapperOutput

   This is what reducers can expect to receive from mappers. The class contains a list of strings. The strings are all of the same word W, i.e. we have a list of W’s. The key of this class is W.

2. **WordCountMapperFunction** extends MapperFunction

   This is used within a mapper. It takes a file block representing a subsection of a file, tokenizes each line in the file, and creates many **WordCountMapperOutput’s**. For each unique word W that appears N times in the file block, we will have an output containing a list of N W’s.

3. **WordCountReducerFunction** extends ReducerFunction

   This is used within a reducer, it takes a list of **WordCountMapperOutput’s** from various mappers, it then counts the number of times that each word appears, and outputs these values into a file. If sorting is enabled, the output file will be sorted by the words alphabetically.

4. **WordCountKeyToReducerFunction** extends KeyToReducerFunction

   This is called by the mapper to map outputs to reducers. It is implemented as follows:

   - If the mapper output key’s first character is within A-Z, 0-9, these are evenly distributed to the reducers. For example, if we have two reducer R1 and R2, any keys starting with A-R will be given to R1, any keys start with S-Z, 0-9 will be given to R2

- For simplicity, any keys beginning with a non-alphanumeric character (i.e. #, $, %, etc) will be sent to the first reducer R1

Our **WordCountJob** class is then implemented as follows: for **getMapperFunction()**, it returns a **WordCountMapperFunction** instance. For **getReducerFunction()**, it returns a **WordCountReducerFunction**, and for **getKeyToReducerFunction()**, it returns a **WordCountKeyToReducerFunction**.

Inverted Word Index Job:

InvertedIndexMapperOutput extends MapperOutput

This class defines the format of data tuple and what would be the key that reducers can expect to receive from mappers. Programmer has to define the mapper output data tuple they want to word with in this class. For the inverted index example, the class contains words, as keys, and the corresponding line number, the index.

InvertedIndexMapperFunction extends MapperFunction

This is used within a mapper. Programmer has to implement `map` method which takes a file block representing a subsection of a file, tokenizes each line in the file, and creates many **InvertedIndexMapperOutput's**. For the inverted index example, words showing in the document will be parsed into tuples containing the word and the corresponding line number.

InvertedIndexReducerFunction extends ReducerFunction

This is used within a reducer, it takes a list of **MapperOutput** from various mappers, while one key goes to one unique reducer. Programmer has to implement `reduce` function to define the way reducer combines and compresses the information from mappers in this class. For the inverted index example, `reduce` function summarizes indices of the by keys into lists of indices and sort the indices of each key.

InvertedIndexKeyToReducerFunction extends KetToReducerFunction

Programmer can specify how the data tuples will be distributed to reducers by keys. For the inverted index example, it is implemented as follows:

- If the mapper output key's first character is within A-Z, 0-9, these are evenly distributed to the reducers. For example, if we have two reducer R1 and R2, any keys starting with A-R will be given to R1, any keys start with S-Z, 0-9 will be given to R2

- For simplicity, any keys beginning with a non-alphanumeric character (i.e. #, $, %, etc) will be sent to the first reducer R1

InvertedIndexJob extends Job

Programmer will define general information of the custom job and implement getters to get instance of key functions in the classes above.

For example, our **InvertedIndexJob** class is then implemented as follows: for **getMapperFunction()**, it returns a **InvertedIndexMapperFunction** instance. For **getReducerFunction()**, it returns a **InvertedIndexReducerFunction**, and for **getKeyToReducerFunction()**, it returns a **InvertedIndexKeyToReducerFunction**.

Configuration file

The configuration file lists the mapper and reducer machines that master will be contacting. Here is the format of tuple representing a machine:

<hostname> <port> <CPU count> <Mapper or Reducer>

For example, the configuration file defines 4 mappers and 2 reducers on localhost will be:

```
localhost 5100 2 Mapper
localhost 5200 1 Mapper
localhost 5300 1 Mapper
localhost 5400 1 Mapper
localhost 5500 1 Reducer
localhost 5600 1 Reducer
```

Here, we are specifying that the first mapper contains 2 CPUs, while the rest of the machines have only 1 CPU.

The following are instructions to deploy, start, and monitor a job. We'll be giving even more detailed instructions in section 3 for running our examples.

How to deploy participants

1. Open a terminal
2. cd into the "**src**" folder
3. Run "**java Core/Main <hostname> <port> <CPU count> <node ID> <Mapper/Reducer>**"
4. Repeat 2. until all participants defined in config file are launched

How to start a master and a job

1. Open a terminal
2. cd into the "**src**" folder
3. Define config file
4. run "**java Core/Main <job name> <outputFileName> <job ID> <delay time> [auto]**"

With auto enabled, the master will automatically print out events during the job, no command line inputs will be needed to manually monitor the job.

Monitoring Commands

If auto is specified, we'll automatically display every event during the job's life cycle, since these events are printed to the console, it makes entering commands and reading the output more difficult. Therefore we disable monitoring by commands when auto output is true.

If the auto is not specified, users can monitor the status of the job manually with the following commands:

"status": Displays what phase (map / reduce) the job is in, displays the health of participants

"blocks": Displays the status of files blocks (unassigned, processing, processed)

"kill": End the job and kill all participants

Section 3 - Step by Step Test Guide for TA’s

Building

1. Open a terminal, cd into the “**src**” folder
2. Type “**make**” to build our project

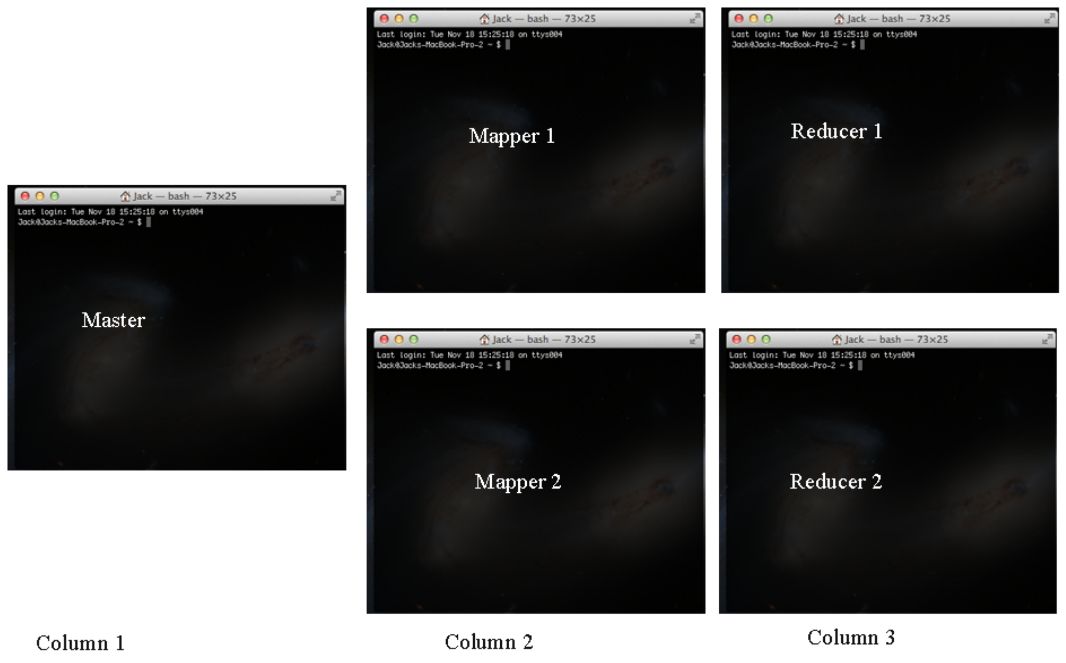
Deploying Mappers and Reducers

Deployment is based on the contents of our configuration file:

```
ghc51.ghc.andrew.cmu.edu 5100 2 Mapper
ghc52.ghc.andrew.cmu.edu 5200 1 Mapper
ghc53.ghc.andrew.cmu.edu 5300 1 Reducer
ghc54.ghc.andrew.cmu.edu 5400 1 Reducer
```

Please open 5 separate terminals for best visualization, we recommend arranging the terminals into 3 columns:

- Column 1 has one terminal, used for master
- Column 2 has two terminals, these will be used to visualize mappers
- Column 3 has two terminals, these will be used to visualize reducers



```
Last login: Tue Nov 18 15:25:18 on ttys004
Jack@Jacks-MacBook-Pro-2 ~ $
```

Deploying mappers and reducers:

On column 2, row one (this will be the first mapper):

- cd into the "src" folder
- ssh into `ghc51.ghc.andrew.cmu.edu, run:`
- `java Core/Main ghc51.ghc.andrew.cmu.edu 5100 2 1 Mapper`

On column 2, row two (this will be the second mapper):

- cd into the "src" folder
- ssh into `ghc52.ghc.andrew.cmu.edu, run:`
- `java Core/Main ghc52.ghc.andrew.cmu.edu 5200 1 2 Mapper`

On column 3, row one (this will be the first reducer):

- cd into the "src" folder
- ssh into `ghc53.ghc.andrew.cmu.edu, run:`
- `java Core/Main ghc53.ghc.andrew.cmu.edu 5300 1 3 Reducer`

On column 3, row two (this will be the second reducer):

- cd into the "src" folder
- ssh into `ghc54.ghc.andrew.cmu.edu, run:`
- `java Core/Main ghc54.ghc.andrew.cmu.edu 5400 1 4 Reducer`

In our example, mapper 1 has 2 CPUs while the other machines have 1 CPU.

** If any of the GHC machines not working when you test it, please edit "src/Config.txt" to change the hostname of participant machine **

Test Cases

Input File

Our input file “SampleInput.txt” contains 500 lines, 20 unique words, each appearing 100 times. Our example breaks the file into chunks of 50 lines, resulting in 10 chunks.

One of our example is to count the appearing of each word, the other example is to display the inverted index of each word, i.e. the lines numbers that each word appears.

Running a Job

We have two ways of running jobs, an auto output mode, and a manual output mode.

In the auto output mode, the master will print all events and status of the job automatically, there are no commands for manually monitoring the status of the job. This is because it’s hard to type in command prompts while the master is constantly printing events to the console.

In the non auto output mode, since the master is not constantly printing out events, you can enter commands to monitor the status of the job.

Test Case 1 - Word Counting

On the master terminal:

- cd into the “src” folder
- type “java Core/Main wordcount test1 1 100 auto”

What you should see:

On master:

- In the map phase, you can see that the master is assigning the 10 blocks between the mappers
- Mapper 1 is assigned the first 2 blocks because it has 2 CPUs
- After all blocks are processed, you can see that the reduce phase is starting, and 2 reducers are running
- When the reducers are done, job complete is printed

On mapper:

- You can see that a message is printed each time it is assigned a block, and each time it finishes mapping a block

On reducer:

- You can see a message each time that it received an output from a mapper
- You can see a message when it begins and finishes reducing

To see the resulting outputs, we’ll look at them on the master terminal (in reality it would be on the reducer terminals, but we are on AFS and it makes things easier as we can leave the reducers running)

You can open the files **test1_reducer3** and **test1_reducer4** to see the outputs. **test1_reducer3** should contain the first half of the words, each showing a count of 100, while **test1_reducer4** should contain the second half of the words, each showing a count of 100.

Note: we saw that sometimes andrew machines have connection issues and results in connection refused exceptions. If this happens, please replace the unreachable machine with another one, and change the host in the configuration file appropriately. If this test case can run properly, every other test case is guaranteed to work.

Test Case 2 - Inverted Word Index

This example is used to find the line numbers that each word occurs in. We'll only use this job in this test case to demo the fact that we can support multiple custom jobs. We use the word counting example mainly because the resulting output is much easier to read.

On the master terminal:

- type "java Core/Main invertedIndex test2 2 100 auto"

The events that you see on all terminals should be very similar to test case 1, the only difference is the resulting output files

After the job completes, you can open the files **test2_reducer3** and **test2_reducer4** to see the outputs. **test2_reducer3** should contain the first half of the words, each showing the line numbers that the words occur in, while **test2_reducer4** should contain the second half of the words, each showing their line numbers.

Test Case 3 - Handling a mapper failure

This example shows that we are able to recover from mapper failure, at this point, all mappers and reducers should still be running on columns 2 and 3.

On the master terminal:

- type “java Core/Main wordcount test3 3 1000 auto”

Note: the “1000” is a delay that makes the mapping and reducing functions run slower, so you have time to kill a mapper.

Now immediately kill (control + c) one of the mappers.

What you should see:

On the master, there should be a line of the form:

```
!!!-----Mapper X connection lost-----!!!
```

Any of the file blocks assigned to X previously are later assigned to the other mapper, the job goes on and completes successfully, you can verify by looking at the files **test3_reducer3** and **test3_reducer4**

Test Case 4 - Handling all mapper failures

Restart the mapper you killed in test case 4 (run the same command you used in the deploy phase). This example will show that the job ends gracefully if all mappers fail.

On the master terminal:

- type “java Core/Main wordcount test4 4 1000 auto”

Now immediately kill (control + c) all of the mappers.

What you should see:

On the master, you should see

```
!!!-----Mapper 1 connection lost-----!!!
!!!-----Mapper 2 connection lost-----!!!
Stopping job due to no mappers reachable
```

Test Case 5 - Handling a reducer failure

Restart the mappers you killed in test case 4 (run the same commands you used in the deploy phase). This example shows that we are able to recover from reducer failure.

On the master terminal:

- type “java Core/Main wordcount test5 5 1000 auto”

Now immediately kill (control + c) one of the reducers.

What you should see:

On the master, there should be a line of the form:

```
!!!-----Reducer X connection lost-----!!!
Restarting job due to reducer 3 connection lost.
```

The whole job then restarts, the master shows that only 1 reducer is running during the reduce phase.

Open the file **test5_reducerX**, where X is the reducer that is alive, you should see that it contains all of the words, each with a count of 100.

Test Case 6 - Handling all reducer failures

Restart the reducer you killed in test case 5 (run the same command you used in the deploy phase). This example will show that the job ends gracefully if all reducers fail.

On the master terminal:

- type “java Core/Main wordcount test6 6 1000 auto”

Now immediately kill (control + c) all of the reducers.

What you should see:

On the master, you should see that after the first reducer connection lost is printed, it tries to restart the job. After the second reducer connection lost is failed, the job ends with the message:

```
Stopping job due to no reducers reachable
```

Test Case 7 - Demoing Configuration File

Here we want to show you that we can in fact support varying number of mappers and reducers.

Open **Config.txt** under the src folder, you should see:

```
ghc51.ghc.andrew.cmu.edu 5100 2 Mapper
ghc52.ghc.andrew.cmu.edu 5200 1 Mapper
ghc53.ghc.andrew.cmu.edu 5300 1 Reducer
ghc54.ghc.andrew.cmu.edu 5400 1 Reducer
```

Now you can add any number of mappers or reducers in a similar format:

```
<hostname> <port> <CPU count> <Mapper or Reducer>
```

Ssh into any new <hostname>, cd into src folder and run:

```
java Core/Main <hostname> <port> <CPU count> ID <Mapper or Reducer>
```

ID here refers to the ID of the participant, which corresponds to its Line Number in the config file, for example if you added a fifth participant, its ID would be 5.

On the master terminal, type “java Core/Main wordcount test7 7 300 auto”, you should see that the job runs successfully, and there are as many output files as reducers.

The simplest way to test this is to just delete the last line in the config file, leaving one reducer, this way you won’t have to open a new terminal. Please restore the configuration file to the original afterwards before continuing to the rest of the test cases.

Test Case 8 - Demoing Job Management Tool

Here we want to show you that we have tools (in the form of commands) to track the status of the job. We will not enable auto output in this test case.

On the master terminal:

- type “java Core/Main wordcount test8 8 2500”

Notice that auto output is not in the arguments, and we have set the delay to 2500 to give you more time to play with the commands.

Type “**status”** to display the phase that the job is in, as well as the connection status with the mappers and reducers, sample output:

```
---------------------
Status:
---------------------
Job ID: 8
Current phase: Map Phase
Mapper 1: Healthy
Mapper 2: Healthy
Reducer 3: Healthy
Reducer 4: Healthy
```

If you kill any participants during the job, their status will update appropriately when you run this command.

More commands on next page:

Type “**blocks”** to display the status of file blocks: they may be **unassigned, processing on mapper x,** or **processed by mapper x**, sample output:

```
---------------------
Block Status:
---------------------
Block 0:  processed by mapper 1
Block 1:  processed by mapper 1
Block 2:  processed by mapper 2
Block 3:  processing on mapper 1
Block 4:  processing on mapper 1
Block 5:  processing on mapper 2
Block 6: Unassigned
Block 7: Unassigned
Block 8: Unassigned
Block 9: Unassigned
```

Type “**kill”** before the job ends to stop the job and stop all participants from running.

Test Case 9 - Concurrent Jobs

Here we demonstrate that we can support concurrent jobs, for simplicity (in case you kill any mappers or reducers), redeploy all reducers and mappers like in page 19.

In additional to the original master terminal, open another terminal and ssh into any andrew machines.

cd into src, run “java Core/Main wordcount test9 9 500 auto” on the original master terminal.
cd into src, run “java Core/Main wordcount test10 10 500 auto” on the new master terminal.

Both jobs should complete successfully and you can see the outputs in the files:
test9_reducer3, test9_reducer4, test10_reducer3, test10_reducer4