System Design

Overall:

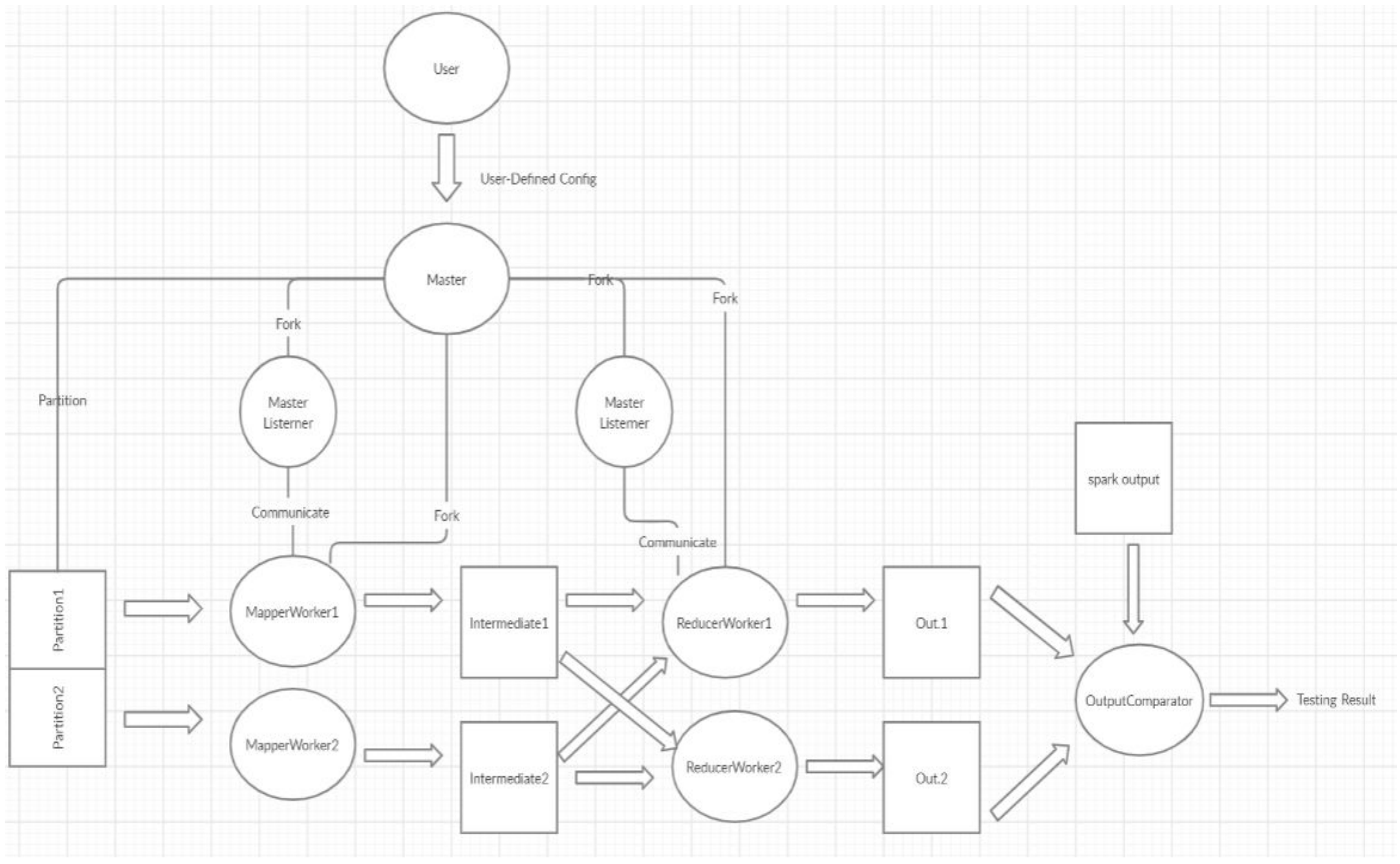


Figure1: Overall System Design Diagram

User:

The user will create a Master instance and then provide Master with a configuration file specifying the: mapper/reducer function class, location of an input file, the location of the output files, and a constant *N*.

Master:

- Master will launch N mapper processes, which will each scan a separate partition of the rows of the input file.
- Master will create N mappers and reducers, telling them the partition or intermediate file locations and coordinating them to perform normally
- Master uses a MasterListener to communicate with each individual mapper/reducer worker (through web socket) and aggregate the information to coordinate the MapReduce operation

MasterListener:

Each MasterListener is responsible for communicating with one single worker through socket. When a worker finishes their work or fails, it will report the information to the Master.

MapperWorker:

MapperWorker represents a mapper process created by master, it is responsible for performing mapper job according to the user-defined mapper function and its own partition. It will generate intermediate files for the partition and notify Master the locations. Once the mapper job is finished, it will pass "Over" to the master listener to inform the master.

ReducerWorker:

ReducerWorker represents a reducer process created by master, it is responsible for performing reducer job according to the user-defined reducer function and its own assigned intermediate files. It will generate output files for the intermediate files and notify Master once the reducer job is finished through passing "Over" to the master listener.

UserConfig:

A class to store all attributes of the user-defined configuration, it includes:

- appName: User Application's name
- numOfMapper: Number of Mappers
- numOfReducer: Number of Reducers
- inputFile: Path and file name for input file
- outputDir: Path for output directory
- mapperFunction: Class name of reducer function for the application
- reducerFunction: Class name of mapper function for the application

MapperFunction:

Define Mapper function interface for user to implement

ReducerFunction

Define Reducer function interface for user to implement

OutputComparator:

Class OutputComperator is used to compare the output result with Spark output

User Applications

There are three applications in this project:

1. **Word Count:** Consider the problem of counting the number of occurrences of each word in a large collection of documents.
2. **Distributed Grep:** The map function emits a line if it matches the supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.
3. **Count of URL Access Frequency:** The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.

How it Works

Workflow

Please refer figure(1) for the following explanation

Initialization Stage:

1. User launch a special process, Master, and tell Master the location of user-defined configuration

2. Master knows the input file from user-defined configuration and partition the input file equally by row

3. Master creates N mapper process. For each mapper, Master will tell it the location of the input file, the partition, and Mapper/Reducer function class through parameter passing.

3. For each mapper, Master will assign a new thread called MasterListener to communicate with

Mapper Stage:

1. Each MapperWorker will use Java Reflect to load the Mapper function class and perform mapper operation on the specified partition. Once the mapper process finished, output launched the <key,value> pairs to the classified intermediate by using a hash function. Here we use a hash function to decide which intermediate file the <key, value> should put in. For example, if there are N reducers, MapperWorker will calculate the value: k = [hash(Key) mod N], and then output the <key,value> pair to the k-th intermediate.

2. Once a single MapperWorker finishes processing its own partition, it passes a message "Over" with the location of all intermediate files to the MasterListener by web socket. Master will wait for each MasterListener to hear the message "Over".

3. After Master hear "Over" from all the MapperWorkers, Master received the locations of all intermediate file

4. [Fault Tolerance for Mapper] If any mappers return an error message or failed, then the Master will clean the intermediate files and re-execute all Mappers. Since the MasterListener keeps pinging the assigned MapperWokrer, it can detect mapper fail by the pinging mechanism.

Reducer Stage:

1. Master will create N reducers and tell each single reducer the intermediate it assigned, reducer function, and the output file location through parameter passing
2. Master assign each reducer a MasterListener to communicate with, MasterListener will wait for reducers to finish their job. For each reducer, it load the reducer function class and then reads all <Key,Value> pairs in intermediate files and sort them by key
3. Perform the user-defined reducer function on each <Key,Value> pair and record the result in an output map. Then output the map to the location user specified
4. Callback to MastListerner with "Over" message through web socket. Master wait until each reducers to finished processing
5. [Fault Tolerance for Reducer] If any reducer returns an error message or fails, the Master re-execute the failed reducer only by creating a new ReducerWorker. Since the MasterListener keeps pinging the assigned ReducerWorker, it can detect reducer failure by the pinging mechanism.

Final Stage:

1. Master return the output file name and path back to user

Communication between Master and Works (Mappers/Reducers)

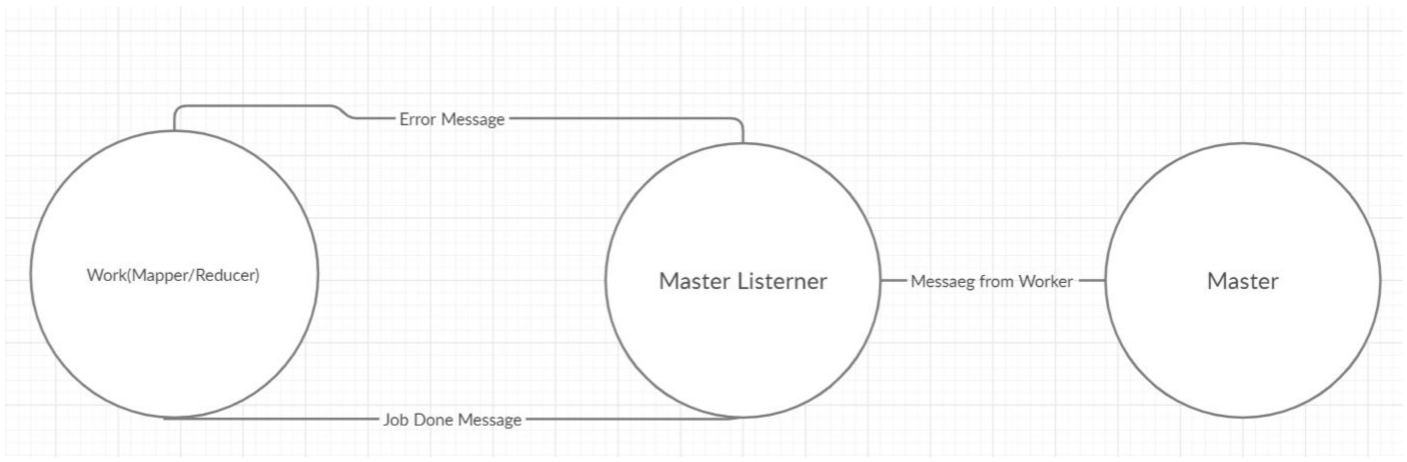


Figure2: Master-Worker(Mapper/Reducer) Communication Diagram

We use a MasterLister to communicate with each worker through a web socket. The Mapper/Reducer worker will report its status like “Over” or “Error” to the MasterListener. MasterListener then reports Master the status of each worker.

Fault Tolerance:

We simulate the failed mapper/reducer by killing the mapper/reducer process directly in the program, user can decide whether to simulate a faulty mapper or a faulty reducer, here is how I implement the fault tolerance mechanism:

1. Master assign each slave worker a MasterListener, the MasterListener keeps listening on each mapper/reducer worker
2. Once a mapper/reducer worker failed, its corresponding MasterListener knows that since the pinning operation will report error
3. Upon MasterListener detect a failed slave, MasterListener tell the Master that there is a slave failed
4. Master will coordinate all MapperWorkers/ReducerWorker failure with the following logic:
    - If a mapper failed, Master will clean the intermediate files and all complected mapper and the failed mapper will be re-executed
    - If a reducer failed, the Master will clean the failed reducer’s output and only re-executed the failed reducer

Validation

1. We run spark program in advance to get correct output for the corresponding testing input
2. Use batch script to compile each class and invoke USER process to execute each application (Including fault tolerance simulation)
3. Use JAVA class, “OutputComperator” to compare the output file with the generated spark output file and print testing results

Design Tradeoffs

Pass Message by TCP Socket vs Pass Message by UDP

We finally choose TCP for message transmission because we don't want to lose any message from the slave but we are able to tolerate a little latency.

- Pros: TCP is a reliable protocol and we don't need to worry about losing message or get corrupted message
- Cons: TCP needs handshake and dedicated connection and may cause some latency

Additional Class (MasterListener) to handle to communication with Slave v.s Single Master to handle communication

We finally decided to create a new class, MasterListener, that is not in the paper to handle the communication with the mappers/reducers. Since we want to separate the logic of communication and coordination and make our code scalable, we chose to create a new class to handle the communication logic:

- Pros: Decouple the logic of communication and coordination, it makes our code extendable and provides us with flexibility to modify feature in the future
- Cons: Need additional effort to maintain a new class

Sort v.s Not to Sort Intermediate Files

We finally decided to sort intermediate files by key using TreeMap even if it cost additional time to do that. Since we want the stability of the final result and the performance of the following sequential search.

- Pros: Reduce the following sequential search time and guarantee the stableness of result after fault tolerance mechanism
- Cons: Cos additional O(n*log(n)) processing time on sorting

Partition file by bytes v.s Partition file by rows

We finally decide to partition the input file by rows as we want the readability and simplicity of the program.

- Pros: Simplify the logic and make the program readable
- Cons: Might lose some performance when the content in each row is very different (Ex. Some row has a few words while other rows have huge amount of words)

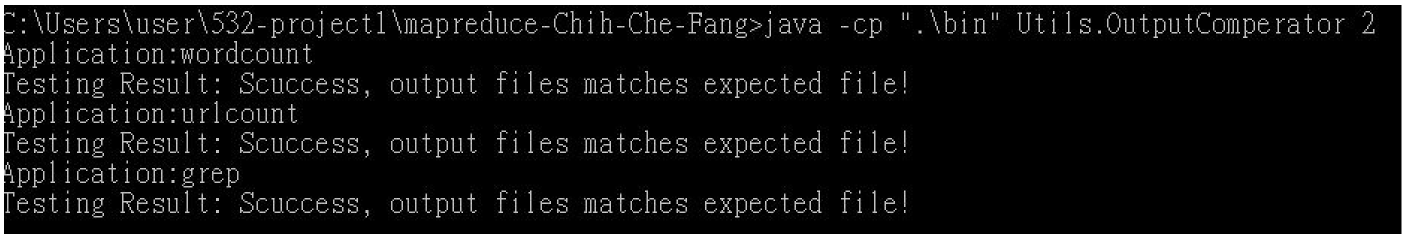
Split files into different packages vs. All in one package

We finally decided to split class into several different packages since we want the scalability and maintainability of the project. As the project becomes very large, we can still easily extend the functionality.

- Pros: Easy to maintain and extend functionality in the future
- Cons: Increase the difficulty of writing make file and invoke class

How to run the program

1. Switch to the root directory of this project (Ex. cd /mapreduce-Cih-Che-Fang) and confirm the path contains no "blank"
2. Perform **run_test.bat** on Windows OS (With JDK installed and with JDK environment variable set), and will output all the files to "output_[APP_NAME]" folder. (Ex. wordcount example will output result to output_wordcount folder, grep example will output to output_grep folder)
3. See the testing result on the console, it will tell you if the output is equal to the Spark output. Logs like:



```
C:\Users\user\532-project1\mapreduce-Chih-Che-Fang>java -cp ".\bin" Utils.OutputComperator 2
Application:wordcount
Testing Result: Scuccess, output files matches expected file!
Application:urlcount
Testing Result: Scuccess, output files matches expected file!
Application:grep
Testing Result: Scuccess, output files matches expected file!
```

Directory/Files Description

- *Bin:* Compiled JAVA class
- *SparkValidation:* Source code of Spark validation program for comparing output
- *Src:* Project source code
- *Run_test.bat*: testing script
- *Local_refresh:* For local debugging use
- *Docs:* Design documents
- *Input_grep:* Input files of application “Distributed Grep”
- *Output_grep:* Output files of application “Distributed Grep”
- *Input_wordcount:* Input files of application “Word Count”
- *Output_wordcount:* Output files of application “Word Count”
- *Input_urlcount:* Input files of application “Count of URL Access Frequency”
- *Output_urlcount:* Output files of application “Count of URL Access Frequency”
- *Intermediate:* Intermediate files
- *Read.md:* Readme file