CSCI B 534 Distributed Systems

Assignment 3: Distributed MapReduce System

Abstract:

In this Assignment, we aim to design a distributed MapReduce System. MapReduce operates in 2 fundamental stages: the Mapping stage and the Reducing stage. In the mapping step, input data is divided into smaller chunks and processed independently in parallel by multiple compute nodes. Each chunk is transformed into a set of key-value pairs. In the reducing stage, the output from the Map phase is shuffled, sorted, and aggregated based on the keys. The Reduce step then takes these intermediate key-value pairs and performs any necessary aggregation or summarization to produce the final result.

Architecture:

The MapReduce program starts by running the driver program which takes the test case number as the argument.
The main components of the MapReduce framework are the Master Node, Mapper Nodes, and Reducer Nodes.

Master Node:

1. The driver reads through the input configuration json file and spawns the Master Node.
2. While spawning the Master Node, the driver passes the necessary parameters to the Master process like the number of mappers, number of reducers, list of mapper and reducer processes, and the ports they should run on, map, and reduce function names.
3. The Master Node spawns the Mappers and the Reducers and passes the required parameters to them.

Mapper Nodes:

1. When a mapper node is spawned by the Master, the mapper calls its run method inside the constructor. All the communication threads are managed by this run method.
2. A Mapper node sends a pulse message to the Master according to the PULSE_INTERVAL specified. In my Implementation, this PULSE_INTERVAL is set to 5 seconds. These pulse messages are crucial for detecting unresponsive or dead mapper nodes.
3. The mapper maps the input data and writes the corresponding result to a file ‘output.txt’ inside its local directory space.
4. After execution of the mapping task, each mapper process sends a DONE message to the Master, indicating that it has finished the mapping task.
5. After the Master receives a DONE message from each mapper, it asks all the mappers to send the data to the reducing stage. These DONE messages act as a distributed barrier, preventing the mappers from sending data to reducers unless all the mapper nodes are done mapping the data.
6. The mapper then iterates through the mapped output data and calculates the hash and depending on the hash value, it sends the data to the respective reducer.

Reducer Nodes:

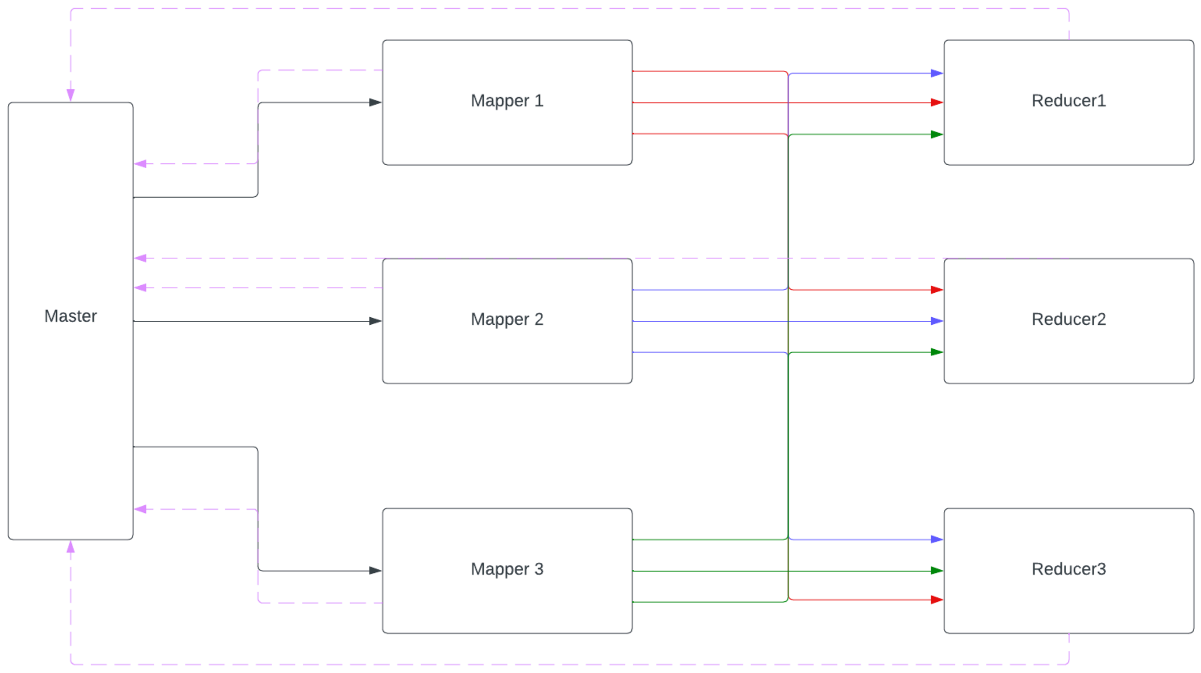
1. The Master spawns the reducers when it receives DONE messages from all the mappers.
2. Similar to the mapper nodes, the reducer nodes also send pulse messages to the Master according to the set PULSE_INTERVAL.
3. When the mappers are done sending the data to the reducers, they send a special message conveying that the data transmission is over.
4. When a reducer receives such messages from all the mapper nodes, it starts its reducing operation.
5. When a reducer is done with its task, it sends a DONE message to the Master.
6. When the Master receives DONE from all the reducer nodes, it performs the cleanup tasks, combines the outputs, and sends TERMINATE messages to the mappers and reducers.

Messages:

1. Each message sent over the system implements an abstract Message base class.

2. Classes implementing this base class implement an abstract serialize method which returns a stringified version of the message object that can be sent over the network.

   Diagram:

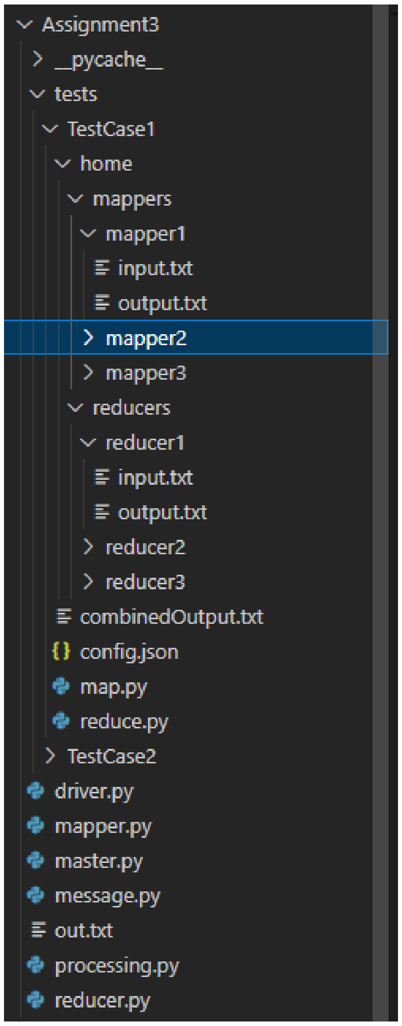


The dotted pink lines represent the pulse messages that the mappers and the reducers send to the Master Node. Each of the Reducer nodes gets the mapped data from each of the mapper nodes. A respective mapper sending data to reducers is represented by a different color in the diagram. Showing all other messages in the diagram is not feasible and is explained in the design specifications.

File Structure

There is a folder called tests which has folders for individual test cases. As of the current implementation, 2 test cases are supported namely word count and inverted index. Each of the respective test case folders has a config.json file specifying the hosts and ports for the mappers and reducers and their respective IDs. The folder also has a map.py and a reduce.py which give the definitions for the map and the reduce tasks as

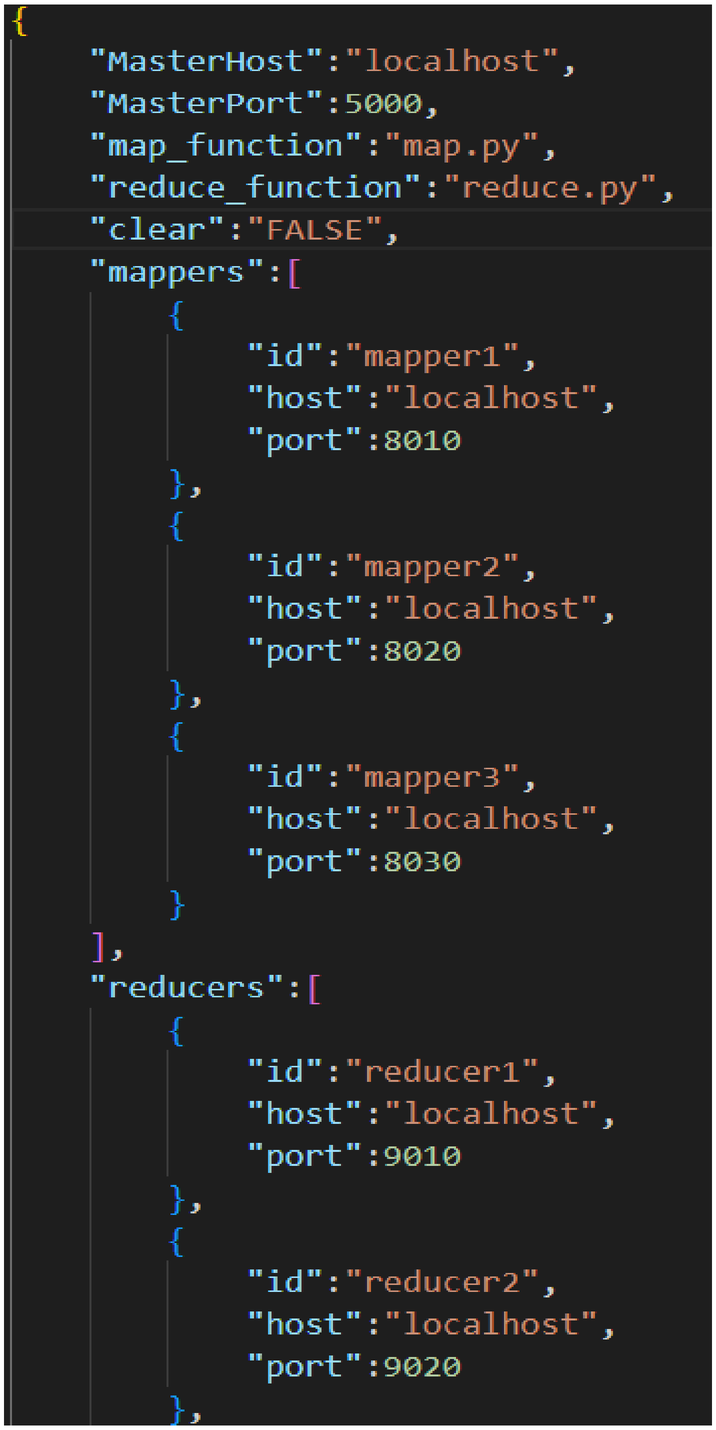
per the test case. Inside each test case folder, is a home folder which has 2 directories one for the mappers and the other for the reducers.
The mappers folder in turn has separate directories for each of the mappers. This directory has a text file called input.txt which is the source input file for the mapper. The mapper process reads the text data from this input file and applies the map logic. The mapper process writes the map output to a file called output.txt inside its directory. This data is then read and sent to the reducers.
Similar to the mappers, inside the reducers folder there are separate folders for each of the reducer processes. The reducer upon receiving data writes it to a file called input.txt inside its local folder. When it executes the reduce task, the final output is written to a file called output.txt inside its folder.
The output of every reducer is then combined into a file called combinedOutput.txt which is present in the test case folder.



```
Assignment3
  __pycache__
  tests
    TestCase1
      home
        mappers
          mapper1
            input.txt
            output.txt
          mapper2
          mapper3
        reducers
          reducer1
            input.txt
            output.txt
          reducer2
          reducer3
      combinedOutput.txt
      config.json
      map.py
      reduce.py
    TestCase2
  driver.py
  mapper.py
  master.py
  message.py
  out.txt
  processing.py
  reducer.py
```

Above is the visual representation of the directory structure.

The config file has the following format



```
{
    "MasterHost":"localhost",
    "MasterPort":5000,
    "map_function":"map.py",
    "reduce_function":"reduce.py",
    "clear":"FALSE",
    "mappers":[
        {
            "id":"mapper1",
            "host":"localhost",
            "port":8010
        },
        {
            "id":"mapper2",
            "host":"localhost",
            "port":8020
        },
        {
            "id":"mapper3",
            "host":"localhost",
            "port":8030
        }
    ],
    "reducers":[
        {
            "id":"reducer1",
            "host":"localhost",
            "port":9010
        },
        {
            "id":"reducer2",
            "host":"localhost",
            "port":9020
        },
```

Design Specifications:

When the Master node is spawned by the driver program, the Master first creates the necessary files for the master, and the reducer by calling the createFiles method from the processing library which is a helper library and contains the necessary helper and cleanup methods.

The Master then spawns the mappers and starts listening for incoming messages on its host and port. The mappers send a PULSE_M message after every PULSE_INTERVAL seconds. I have set his interval as 5 seconds for the mappers and the reducers. The Master maintains a dictionary that records the latest pulse received from each mapper. The Master continuously checks whether the difference between the current time and the last received pulse for each mapper is greater than the TIMEOUT which is set to 10 seconds. If it is, then the master declares that the respective mapper node is dead.

Each of the mapper processes executes the map function by reading the input file from their respective folders. The mapper writes the mapped data (key-value pairs) to a file called output.txt in its local folder. After a mapper is done writing the result to the output file, it sends a DONE message to the Master, indicating that it is done with the mapping phase. Each of the message types has its class which implements the Message base class. The Master continuously checks from which mappers it has received DONE_M messages, by maintaining a dictionary that sets the value of the mapper key to True if it has received a DONE_M from it. After the Master receives a DONE_M message from each of the mappers, it spawns the reducer nodes. To ensure effective testing, I have added sleep statements in between.

After spawning the reducers, the Master starts listening for reducer pulses and reducer DONE_R messages too. The Master then sends a message to all mappers instructing them to start the data transfer to the reducer nodes.
After receiving a message from the Master for sending the data to the reducer, the mapper goes through the output.txt file word by word and calculates the hash of the word. I have calculated the hash by converting the first letter of each word to its Unicode representation and taking its mod with the number of reducer nodes. The reducer node with the corresponding id is queried and the data is sent to the particular reducer. After all the data transfer is complete, the mapper sends a DONE_MAPPER_REDUCER message to each of the reducers conveying that the data transfer is over.

After a reducer process receives a DONE_MAPPER_REDUCER message from each mapper, it executes the reducing stage by reading the data from input.txt inside of its local folder. After the reduce step is done, the reducer process sends a DONE_R message to the Master.

After the Master receives a DONE_R message from every reducer, it starts the termination process by sending a TERMINATE message to all the mappers and the reducers. When a mapper or a reducer receives a TERMINATE message, they break out of all the thread loops and kill themselves. After sending the TERMINATE messages, the Master waits for specified time units and then breaks out of all thread loops, runs the combiner functions, performs the clean-up tasks, and terminates itself thus ending the MapReduce.

Note: As the assignment involved designing custom protocols for communication, I thought using sockets was a good choice due to the low level of control and flexibility they provide. However, to ensure platform independence and ease of use, RPCs can be used instead of sockets.

Bonus: For Fault Tolerance and testing
For testing Fault Tolerance, I have added a config file inside each mapper folder. This JSON file has an attribute called 'kill' which is currently set to FALSE. Setting this to TRUE will kill the mapper after some time when it has spawned.
The Master comes to know of the mapper's failure when it does not receive the pulse from this mapper. After detecting this failure, the Master sets the mapping done key of this mapper to False, as it will have to do the mapping again. One limitation of my system is that it is very hard to make the mapper fail while sending data to the reducers to variable delivery times. So the only failure happens when the mapper is in the mapping phase. The Master spawns a new mapper with the same ID and host and port details.
In my implementation, I ensure that whenever a mapper or a reducer is spawned, I clear its input/output buffers.
For reducers, I am successfully able to detect reducer outages by tracking its pulse, but due to lack of time, I could not get it to resist the failures.

Running Instruction:

Run the driver.py from the command line by running the following command python ./driver.py testcase_num. As of now, the system supports 2 test cases.

1. Word Count
2. Inverted Index

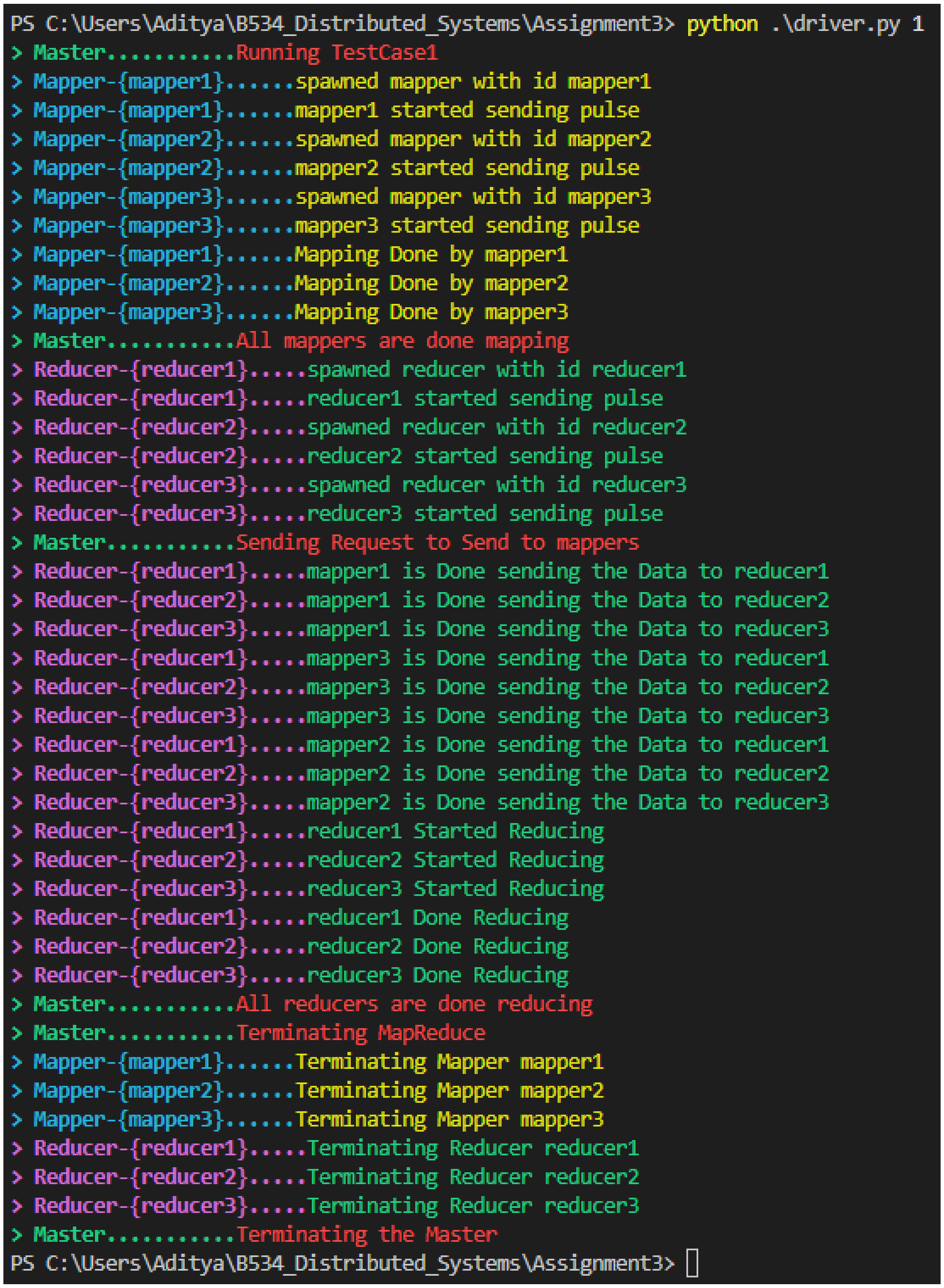
The input file for each mapper is already in place and does not change. The final output is available inside the combinedOutput.txt file inside the test case folder. The mapper output file and the reducer input and output file buffers are cleared every time that particular mapper/reducer is spawned. If you want to run the program again, make sure to clear the combinedOutput.txt file before rerunning the program.

Note: One limitation in my system is that, when the Master terminates, it also terminates the mappers and reducers. But when a mapper is terminated, it sometimes does not release the lock on the output file handle and when you try to run that particular test case again, you get the PermissionDenied error because the previous mapper process that got killed did not release the file handle. This usually happens when testing for fault tolerance and then testing normal execution after that.
Sometimes, the PermissionError is non-blocking and the execution continues as expected.
In case you want to run the program again, please close or kill the terminal session, refresh the screen, open the terminal again, and then try to run the testcase.
I am still trying to find a better and an elegant way to resolve this issue where I can run and test the system multiple times without exiting the terminal session.

Note:
If you continue to face the PermissionError, please close the terminal session, refresh the system, and then try running the system without testing for fault tolerance. The normal execution runs as it should.
Make sure to delete the log files for the Master, mapper and the reducers when the program again.

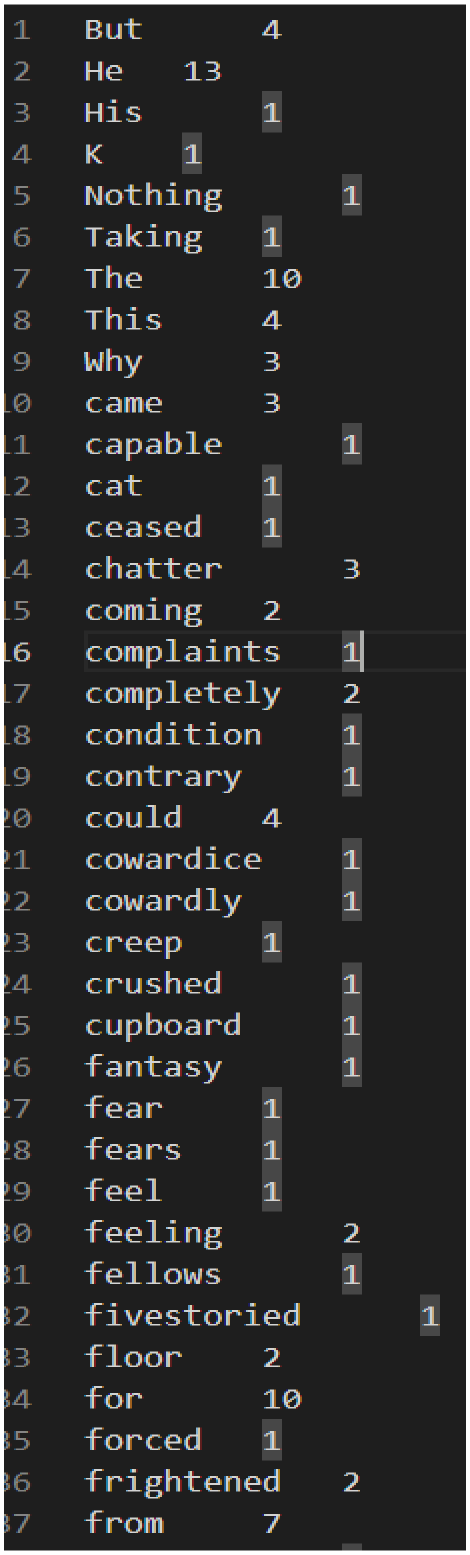
Test Cases:

This is the log Output of running test case 1 (Word Count)



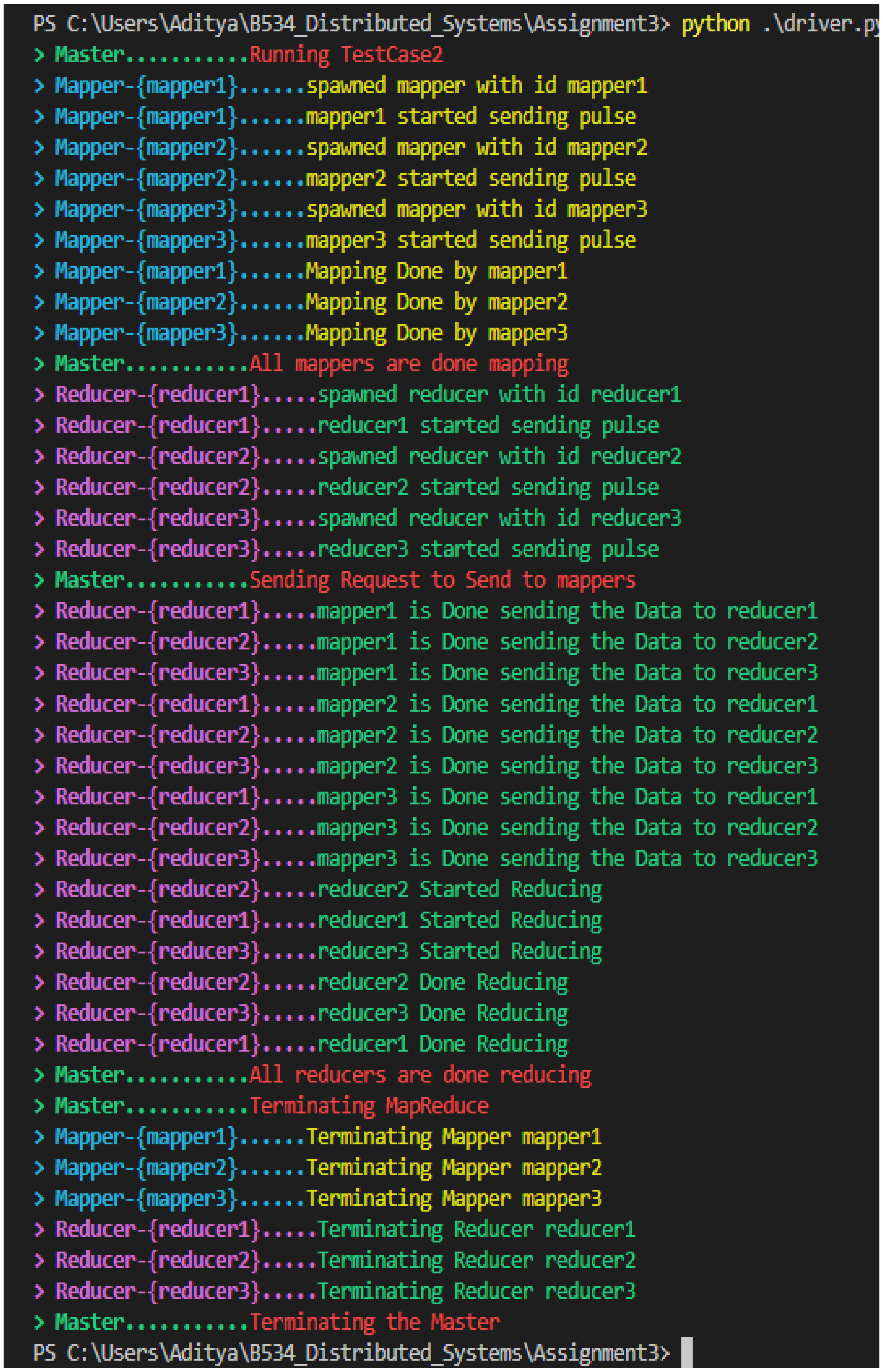
```
PS C:\Users\Aditya\B534_Distributed_Systems\Assignment3> python .\driver.py 1
> Master...........Running TestCase1
> Mapper-{mapper1}......spawned mapper with id mapper1
> Mapper-{mapper1}......mapper1 started sending pulse
> Mapper-{mapper2}......spawned mapper with id mapper2
> Mapper-{mapper2}......mapper2 started sending pulse
> Mapper-{mapper3}......spawned mapper with id mapper3
> Mapper-{mapper3}......mapper3 started sending pulse
> Mapper-{mapper1}......Mapping Done by mapper1
> Mapper-{mapper2}......Mapping Done by mapper2
> Mapper-{mapper3}......Mapping Done by mapper3
> Master...........All mappers are done mapping
> Reducer-{reducer1}.....spawned reducer with id reducer1
> Reducer-{reducer1}.....reducer1 started sending pulse
> Reducer-{reducer2}.....spawned reducer with id reducer2
> Reducer-{reducer2}.....reducer2 started sending pulse
> Reducer-{reducer3}.....spawned reducer with id reducer3
> Reducer-{reducer3}.....reducer3 started sending pulse
> Master...........Sending Request to Send to mappers
> Reducer-{reducer1}.....mapper1 is Done sending the Data to reducer1
> Reducer-{reducer2}.....mapper1 is Done sending the Data to reducer2
> Reducer-{reducer3}.....mapper1 is Done sending the Data to reducer3
> Reducer-{reducer1}.....mapper3 is Done sending the Data to reducer1
> Reducer-{reducer2}.....mapper3 is Done sending the Data to reducer2
> Reducer-{reducer3}.....mapper3 is Done sending the Data to reducer3
> Reducer-{reducer1}.....mapper2 is Done sending the Data to reducer1
> Reducer-{reducer2}.....mapper2 is Done sending the Data to reducer2
> Reducer-{reducer3}.....mapper2 is Done sending the Data to reducer3
> Reducer-{reducer1}.....reducer1 Started Reducing
> Reducer-{reducer2}.....reducer2 Started Reducing
> Reducer-{reducer3}.....reducer3 Started Reducing
> Reducer-{reducer1}.....reducer1 Done Reducing
> Reducer-{reducer2}.....reducer2 Done Reducing
> Reducer-{reducer3}.....reducer3 Done Reducing
> Master...........All reducers are done reducing
> Master...........Terminating MapReduce
> Mapper-{mapper1}......Terminating Mapper mapper1
> Mapper-{mapper2}......Terminating Mapper mapper2
> Mapper-{mapper3}......Terminating Mapper mapper3
> Reducer-{reducer1}.....Terminating Reducer reducer1
> Reducer-{reducer2}.....Terminating Reducer reducer2
> Reducer-{reducer3}.....Terminating Reducer reducer3
> Master...........Terminating the Master
PS C:\Users\Aditya\B534_Distributed_Systems\Assignment3> 
```

Output File looks like this



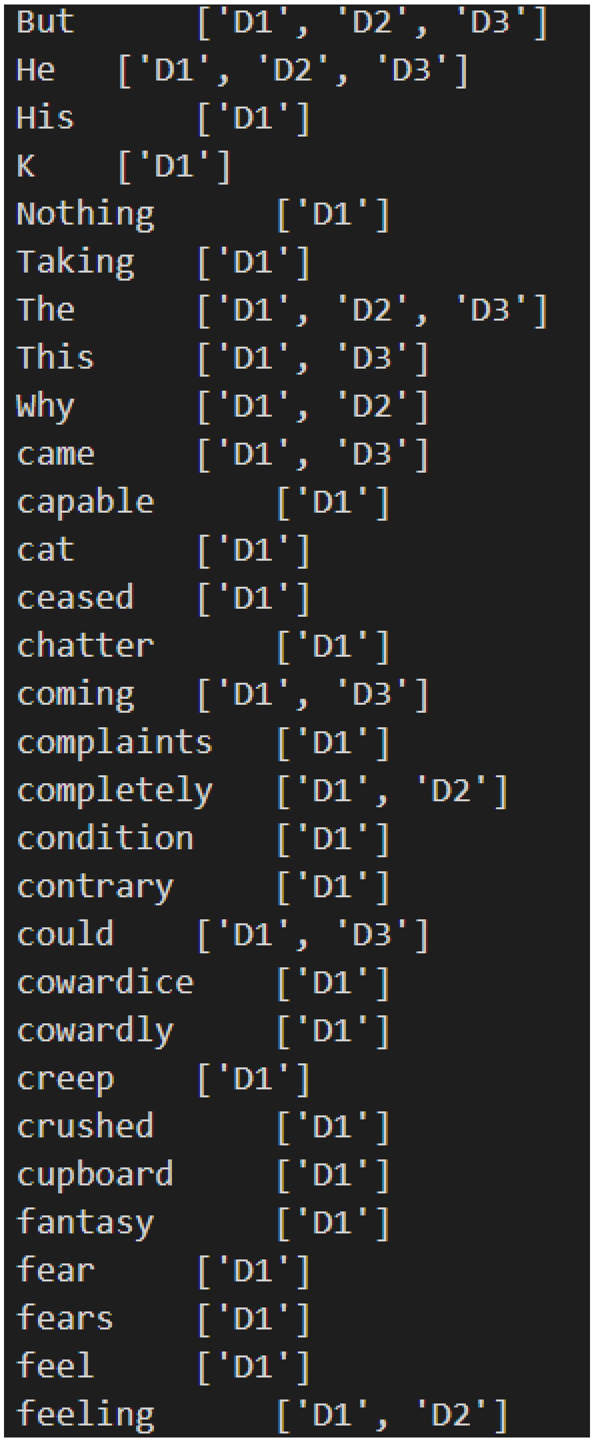
```
But     4
He  13
His     1
K   1
Nothing     1
Taking  1
The     10
This    4
Why     3
came    3
capable     1
cat     1
ceased  1
chatter     3
coming  2
complaints  1
completely  2
condition   1
contrary    1
could   4
cowardice   1
cowardly    1
creep   1
crushed     1
cupboard    1
fantasy     1
fear    1
fears   1
feel    1
feeling     2
fellows     1
fivestoried     1
floor   2
for     10
forced  1
frightened  2
from    7
```

This is the log Output of running test case 2 (Inverted Index)



```
PS C:\Users\Aditya\B534_Distributed_Systems\Assignment3> python .\driver.py
> Master...........Running TestCase2
> Mapper-{mapper1}......spawned mapper with id mapper1
> Mapper-{mapper1}......mapper1 started sending pulse
> Mapper-{mapper2}......spawned mapper with id mapper2
> Mapper-{mapper2}......mapper2 started sending pulse
> Mapper-{mapper3}......spawned mapper with id mapper3
> Mapper-{mapper3}......mapper3 started sending pulse
> Mapper-{mapper1}......Mapping Done by mapper1
> Mapper-{mapper2}......Mapping Done by mapper2
> Mapper-{mapper3}......Mapping Done by mapper3
> Master...........All mappers are done mapping
> Reducer-{reducer1}.....spawned reducer with id reducer1
> Reducer-{reducer1}.....reducer1 started sending pulse
> Reducer-{reducer2}.....spawned reducer with id reducer2
> Reducer-{reducer2}.....reducer2 started sending pulse
> Reducer-{reducer3}.....spawned reducer with id reducer3
> Reducer-{reducer3}.....reducer3 started sending pulse
> Master...........Sending Request to Send to mappers
> Reducer-{reducer1}.....mapper1 is Done sending the Data to reducer1
> Reducer-{reducer2}.....mapper1 is Done sending the Data to reducer2
> Reducer-{reducer3}.....mapper1 is Done sending the Data to reducer3
> Reducer-{reducer1}.....mapper2 is Done sending the Data to reducer1
> Reducer-{reducer2}.....mapper2 is Done sending the Data to reducer2
> Reducer-{reducer3}.....mapper2 is Done sending the Data to reducer3
> Reducer-{reducer1}.....mapper3 is Done sending the Data to reducer1
> Reducer-{reducer2}.....mapper3 is Done sending the Data to reducer2
> Reducer-{reducer3}.....mapper3 is Done sending the Data to reducer3
> Reducer-{reducer2}.....reducer2 Started Reducing
> Reducer-{reducer1}.....reducer1 Started Reducing
> Reducer-{reducer3}.....reducer3 Started Reducing
> Reducer-{reducer2}.....reducer2 Done Reducing
> Reducer-{reducer3}.....reducer3 Done Reducing
> Reducer-{reducer1}.....reducer1 Done Reducing
> Master...........All reducers are done reducing
> Master...........Terminating MapReduce
> Mapper-{mapper1}......Terminating Mapper mapper1
> Mapper-{mapper2}......Terminating Mapper mapper2
> Mapper-{mapper3}......Terminating Mapper mapper3
> Reducer-{reducer1}.....Terminating Reducer reducer1
> Reducer-{reducer2}.....Terminating Reducer reducer2
> Reducer-{reducer3}.....Terminating Reducer reducer3
> Master...........Terminating the Master
PS C:\Users\Aditya\B534_Distributed_Systems\Assignment3>
```

Output File looks like this



```
But	['D1', 'D2', 'D3']
He	['D1', 'D2', 'D3']
His	['D1']
K	['D1']
Nothing	['D1']
Taking	['D1']
The	['D1', 'D2', 'D3']
This	['D1', 'D3']
Why	['D1', 'D2']
came	['D1', 'D3']
capable	['D1']
cat	['D1']
ceased	['D1']
chatter	['D1']
coming	['D1', 'D3']
complaints	['D1']
completely	['D1', 'D2']
condition	['D1']
contrary	['D1']
could	['D1', 'D3']
cowardice	['D1']
cowardly	['D1']
creep	['D1']
crushed	['D1']
cupboard	['D1']
fantasy	['D1']
fear	['D1']
fears	['D1']
feel	['D1']
feeling	['D1', 'D2']
```

Known Limitations:

1. When the mapper process terminates, it sometimes does not release the file handles and the system halts while running the same test case again without killing and restarting the terminal session again.
2. The system can detect reducer outages but cannot tolerate them as of now.
3. We assume that the Master node is fault-tolerant by nature and it cannot die abruptly
4. The mapper with a large chunk of input data will lag and will cause the other mappers to wait due to the barrier.

Potential Improvements:

1. The mappers and the reducers should exit gracefully and should release all the file handles.
2. The Reducers can be fault-tolerant
3. Some state configurations can be exchanged between mappers and the reducers by using a shared Key-value store.
4. The mapper and reducer nodes can communicate over HTTP instead of sockets because the nodes need not be located on the same machine and can be located among multiple distinct servers.

References:

1. https://docs.python.org/3/library/socket.html
2. https://docs.python.org/3/library/threading.html
3. https://docs.python.org/3/library/multiprocessing.html
4. MapReduce Paper
5. Failure Detection in Distributed Systems