

Final Design Document

The design document has been divided into five sections:

1) Run.py:

The run.py has four methods that are described in depth below. We first define the *startSocket* method. The method creates a socket object that binds and listens to the server local host (127.0.0.1) and port number 12345. It also runs a continuous while loop where it listens to any incoming requests and assigns a new thread to the incoming request.

```

16     def startSocket(N):
17         try:
18             host = "127.0.0.1"
19
20             # reverse a port on your computer
21             # in our case it is 12345 but it
22             # can be anything
23             port = 12345
24             s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
25             s.bind((host, port))
26             print("socket binded to port", port)
27
28             # put the socket into listening mode
29             s.listen(2 * N)
30             print("socket is listening")
31
32             # a forever loop until client wants to exit
33             while True:
34
35                 # establish connection with client
36                 c, addr = s.accept()
37
38                 print('Connected to :', addr[0], ':', addr[1])
39
40                 # Start a new thread and return its identifier
41                 start_new_thread(MapReduce.threaded, (c,))
42                 s.close()
43             # and except it when Ctrl + C like happens
44         except KeyboardInterrupt:
45             pass

```

The *threaded* method decodes the incoming data and then converts it into a list. The last element of the list tells us whether the function called is either a Map or a Reduce function. The second from last element tells us the location of the input/output file. If data transferred is null, then the *threaded* method just prints out 'Bye'.

```

47     def threaded(c):
48         # data received from client
49         #data = c.recv(1024)
50         data = c.recv(1048576)
51         # Decode received data into UTF-8
52         data = data.decode('utf-8')
53         # Convert decoded data into list
54         data = eval(data)
55         #check if map or reduce is called
56         maporReduce = data[-1]
57         data = data[0:len(data) - 1]
58         #check for location of input/output file
59         fileLocation = data[-1]
60         data = data[0:len(data) - 1]
61         if not data:
62             print('Bye')
63
64         if maporReduce == 'map':
65             for x in data:
66                 UDFmapreduce.map(fileLocation, x)
67         else:
68             #take in list of intermediate file locations
69             values = data[-1]
70             data = data[0:len(data) - 1]
71             #iterate through each key and call reduce
72             for x in data:
73                 UDFmapreduce.reduce(x, values, fileLocation)
74         # connection close
75         c.close()
76         print("thread closed")

```

Next, we launch the UDF Map function for the given input file. If the reduce function is called instead, we launch the UDF reduce method. Once either the Map or Reduce method is processed, we terminate the socket connection and print out ‘thread closed’.

The *createClient* method creates a new socket and connects to the server (master) on the local host (127.0.0.1) using port number 12345. It sends the data to the server and then terminates the socket connection.

```

78         def createClient(data):
79             s = socket.socket()
80             s.connect(("127.0.0.1", 12345))
81             s.send(data)
82             s.close()

```

The *mainFunc* method which acts as a master. It takes in the input file, UDF and the output file location. The `file.readlines()` method converts the input txt file into a list where each element is a line from the input txt file. The master starts the server socket and then creates N equal partitions for the N mapper nodes. The *createClient* method creates a new socket, connects to the server(master) and then sends the data over the socket.

We impart a delay of 10s using the time module to check if all mapper threads have finished processing. Next, we search for the unique set of keys in the intermediate files that were created by all mappers. For each intermediate file, 'words' is a list where each element contains the word from the text file and the mapped value (e.g. ['apple 1', 'banana 1']). We then run a for loop on the 'words' list to store the word as a key by splitting the elements by a space and then taking the first half as the key(e.g. ['apple', '1']). All the keys are stored in the `self.keys` list.

Note: The words in the input text file are split by " " without taking into account any punctuation. This means that 'apple' and 'apple.' are two separate keys.

Next, we launch the UDF reduce method using a for loop. The reduce method is called for each individual key in the `self.keys` list. The reduce method will produce the output in the form of a text file where each file contains only 1 key and corresponding word count. If there are M keys, we will get M output files each containing only one key and the word count.

2) Config.txt file:

The config file has four parameters - `inputfileLocation`, `outputfileLocation`, N (number of worker threads) and `UDFmapreduce` (name of the Python UDF file)

```

1  inputfileLocation = "C:\Users\BASEDMACHINE\Desktop\CS532Project1\hamlet.txt"
2  outputfileLocation = "C:\Users\BASEDMACHINE\Desktop\CS532Project1"
3  N = "1"
4  UDFmapreduce = "wordCountUDF"

```

3) UDF:

The UDF contains two methods: `map` and `reduce`. The `map` method takes in two arguments `key` and `value`. Key here is the `inputFile` and value is a single line from the input text file. It then creates a random file name which starts with 'intermediate' and then writes all the words and the mapped value to the intermediate file.

E.g., input for the map method: value → 'Ber. Who's there?'

Intermediate file written for the input line:

'Ber. 1'

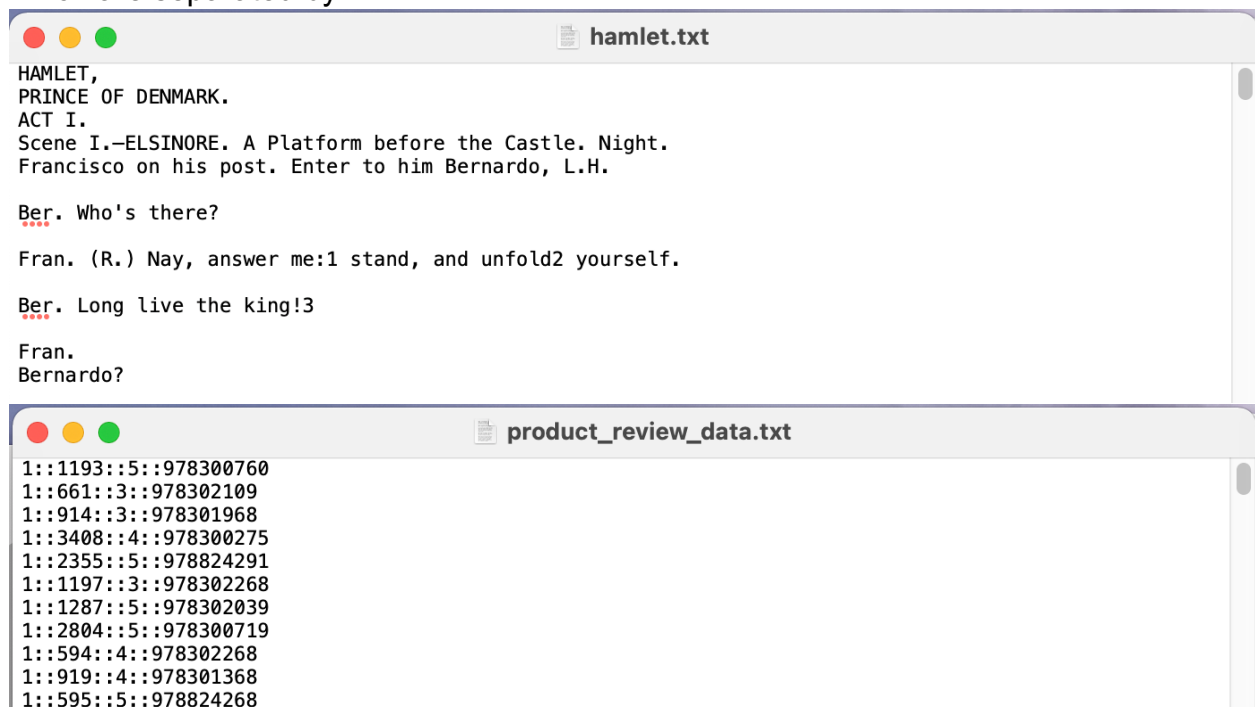
'Who's 1'

'there? 1'

The reduce method goes through all the intermediate files that were created and then searches for the key. If the key is found, then the counter is incremented by 1. After going through all the files, the final counter and the key are written in the output file. **Note:** Each output file will only contain one key and the counter value.

4) Input file:

First file is the hamlet.txt file and the second is a dataset from an e-commerce website which contains four columns - user_id, product_id, ratings_out_of_5 and time_stamp which are separated by '::'.



```

hamlet.txt
HAMLET,
PRINCE OF DENMARK.
ACT I.
Scene I.—ELSinORE. A Platform before the Castle. Night.
Francisco on his post. Enter to him Bernardo, L.H.

Ber. Who's there?

Fran. (R.) Nay, answer me:1 stand, and unfold2 yourself.

Ber. Long live the king!3

Fran.
Bernardo?

product_review_data.txt
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
1::595::5::978824268
  
```

5) Fault Tolerance:

To check for fault tolerance of the mapper processes, we are keeping track of the intermediate files generated after every mapper function is executed. Since the number of intermediate files will be equal to the number of rows in the input text file, we can keep track of how many intermediate files are generated after each mapper function is finished. If the total number of intermediate files are not equal to the total number of rows in the input file, we know that at least one mapper has failed. This way we can detect faults in the MapReduce program.