**Advanced Operating Systems**

**CMSC 621 Project 1**

**University of Maryland, Baltimore County**

**Gossip based Distributed application System Using Erlang**

**{David Lachut, Mahbubur Rahman, Kaavya Srinivasan, Shrikant Bhangdiya}**

**Problem Statement**

Consider a single large text file F and a system that consists of N computer nodes. Split the file F into M fragments F\_1, F\_2, … , F\_M and place them at various nodes of the system. Place each fragment at one or more nodes (that means that you should replicate some of the fragments). The fragments may not all have the same size. Perform the following computational tasks:

* Find the longest word in F and store it at all the nodes
* For specific word(s) find in which nodes this/those word(s) is/are stored. Start this task at node 1, with the user specifying the word(s).
* Find the most frequent word in the text file F and store it at node 1.
* Update the contents of fragment i at each node that may have a copy of it. The update originates at node 1, with the user specifying the fragment number.

In doing these computations, our users are willing to trade-off accuracy for latency and communication costs. The highest accuracy for given latency and communication costs is highly desired.

Design and develop a distributed application that performs these computational tasks. Your solution must utilize gossip and be robust, efficient, and effective at solving the problem(s) at hand. Your application should be implemented using the Erlang distributed computing system.

**Introduction**

A Gossip Protocol is a type of communication protocol used mostly in large scale distributed systems. It is based upon gossip conversations which are common in many social circles. Gossip Protocols are popular in distributed application due to their simplicity, scalability, and high reliability even in constantly changing environments. Gossip Protocols involve periodic message exchanges between node pairs. These eventually result in information being spread throughout the system, similar to human gossiping. The term “epidemic protocol” is sometimes used as a synonym for a gossip protocol, because gossip spreads information in a manner similar to the spread of a virus in a biological community. In the case of large scale distributed systems, instead of diseases, they spread information. [1]

**File Fragmenting**

The distributed system starts the task in the master node where a sample file is taken as the input. The contents of the file are read into a list and fragmented. The algorithm for the fragment is given below,

**Algorithm**

Fragment (**File, NumNodes**)

***UnitList*** *= Divide the* ***File*** *into unit list words*

***REM*** *= Divide the* ***UnitList*** *into* ***NumNodes****-5*

*Combine the list of words into single phrase*

***Tuple*** *= Attach fragment number to each phrase*

*Replicate randomly picked fragments,* ***REM*** *times*

The basic working of the algorithm is to get an input file and split the entire file into a list of single words. Based on the number of nodes passed as input, the file is divided into equal size fragments. The reason we did it into equal size fragments is that, if randomness is involved in splitting the words then it might be possible that all the words go into a single fragment. So we divide the word list into sub lists of equal size. Each sub list is then combined into one single phrase of bit string. A record of fragment is maintained which keeps track of the fragment and the number associated with it. The number 5 is an arbitrary number that we have chosen to reduce the fragment size from the node size. After fragments are created based on number of nodes – 5, to support replicated, a random 5 fragments are picked from the existing ones and added to the list of fragments. To explain how the fragment works, consider the below sample,

**Sample Output**

“*Indigenous peoples lived in what is now the United States for thousands of years and developed complex cultures before European colonists began to arrive”*

This is reduced to individual words in UnitList

*Indigenous people lived in what is now*

*the United States for thousands*

*of years and developed Complex cultures*

*before European colonists began to arrive*

Total number of words in phrase = 24

Number of nodes = 10

Number of original fragments generated = 6

Number of replicated fragments generated = 4

The fragment produces a tuple as below,

[{fragment,<<"Indigenous peoples lived in">>,1},

{fragment,<<"what is now the">>,2},

{fragment,<<"United States for thousands">>,3},

{fragment,<<"of years and developed">>,4},

{fragment,<<"complex cultures before European">>,5},

{fragment,<<"colonists began to arrive">>,6},

{fragment,<<"United States for thousands">>,3},

{fragment,<<"complex cultures before European">>,5},

{fragment,<<"colonists began to arrive">>,6},

{fragment,<<"what is now the">>,2}]

**Gossip Implementation**

The architecture behind our implementation of gossip is given below.

Master Node

Fragments

**Fig. 1.1 System Architecture at time 0**

Fragments

Master Node

Issue direct commands to Node0

**Fig. 1.2 Node Architecture at Time t.**

The master node is the trigger point of our application. The master node module contacts the fragments module to get the fragments of the file. It then spawns processes for each of the worker nodes and sends out a fragment to store in that worker node. Master node sends out messages only to Node 0. Node 0 then starts the gossip process. The worker nodes then start gossiping the data. Node 0 communicates the result to the master node.

Master node starts by spawning processes for each of the nodes. The start module creates a data segment that it shares with its co-workers. The data segment consists of the following,

* Worker – The module to be run for each spawned process.
* startNode – Module inside the Worker module that sends and receives data.
* F – Fragment associated with each data
* #neighbor{pid=Node0,age=0} – Each node initially has Node 0 as its neighbour which later changes to a random list of neighbours. Age is counter that indicates the most recent neighbour that a node shared its data with. Consider an example to see the way it works,

Say a node in the system, Node 10 has a list of neighbours {2,5,7,6,8} currently. When the gossip process starts, it selects another random say Node 2 to share data. After exchange, the new neighbour list for Node 10 would be a combined list of neighbours from Node 10 and Node 2. The neighbour combine function sorts the neighbour list based on its age. It picks half of the most recent neighbours from Node 2 and combines it with Node 10. Node 10 would now have a list of neighbours that it recently shared the data.

* Initial Time – The gossip starts at time 0.
* MasterNode - The master node’s PID is stored in each worker node. This enables a worker to contact the master node anytime.
* IsNode0 – For task3, node 0 stores a dictionary of words with frequencies updated for every exchange.

Every worker is limited with a neighbour list of 5. It maintains the 5 most recent workers that it shared data with at any point of time in the system. If any node doesn’t receive any messages from its co-workers, then it starts information exchange among themselves. The master node can request information from Node 0 anytime. Node 0 then responds with the data that it has.

Initially the neighbour list for Node 0 is empty. Since every worker has Node 0 as its only neighbour at the start of the system, all the workers exchange data with Node 0. When the workers exchange data, they send their own PID to Node 0 so that Node 0 can maintain a current list of neighbours.

Node 0 then sends back its own data with a list of its neighbours at that particular time to the worker. After a certain amount of time has progressed, each worker will have a random list of 5 neighbours to itself.

Each worker also maintains a clock value that is periodically incremented. This is an implementation of Lamport’s logical clock. As long as the nodes are alive, the clocks get incremented for every information exchange.

The master node can also kill all the nodes, by sending a die all message, which kills all the nodes.

**Find the longest word in F and store it at all the nodes**

As a part of the gossip, the longest word is continuously updated. The longest word in a worker’s data is updated only if its own copy of the longest word is less than the one that it received from another worker node. If the length of its own copy is larger no updates are carried out.

Master node sends out the longest word message to Node 0. If no message is received from Node 0 after 5 seconds, it means that the message is lost. Otherwise Node 0 returns the longest word that it has.

Fig 1.3, shows a sample gossip network, everyone will get to know that “Honorificabilitudinitatibus” as the longest word in the network.

I had no idea my neighbour’s house was robbed yesterday!!!

This news seems interesting!

So did you like the gift that I sent you?

****

Granny, what’s the recipe to make a cheese cake?

Dude, did you know the longest word in Shakespeare’s works? It’s “Honorificabilitudinitatibus”

Hey!! Where did you get those shoes from?

**Fig. 1.3**

**For specific word(s) find in which nodes this/those word(s) is/are stored**

We have implemented search in two ways.

**Method I**

The request to search for a word is basically flooded to the network. The master node sends a request to Node 0 to find a specific word in its fragment. Node 0 then sends requests further to its neighbours. Whenever a word in found in a fragment, that worker sends a message to the master node saying that a word is found. A tuple record for search is maintained,

-record(floodsearch,{word,repeats}).

In the above tuple, word is the word to be searched. repeats indicates the number of times the search is forwarded. In order to limit the number of requests that gets flooded in the network, when a node receives a request to search for a word, the repeats value is increased by 1. It forwards the request till the repeats value reaches 2. When the value reaches 2, the worker node loses the request. This way the network is saved from flood traffic. When there are any repeats left for a worker, it broadcasts the search query to its neighbours.

Below is the brief algorithm of the task,

*searchCombine (* ***Data, Query*** *)*

*If* ***Query.repeats*** *< 2*

*Broadcast the query to neighbours.*

*If* ***Data*** *has* ***word*** *in* ***Query***

*Return the data to the master node.*

**Method II**

The second approach that we made was to search for a fragment through gossip. The workers keep exchanging data as a part of the gossip process. If the requested word is found in any fragment contained by the worker, the worker directly responds back to the master node with its PID.

*search(* ***Data, Query*** *)*

*If* ***Data.lastSearch = undefined***

*Check if the node’s fragment has that word.*

*If* ***Data.Fragment*** *has* ***Word****,*

*Return the PID of the node to the master node.*

*Else*

*Update the* ***lastSearch*** *in Data*

*Else*

*Update the* ***lastSearch*** *in Data*

**Sample Output**

Let us look at an example for the same fragment list used above,

Node 0 - {fragment,<<"Indigenous peoples lived in">>,1},

Node 1 - {fragment,<<"what is now the">>,2},

Node 2 - {fragment,<<"United States for thousands">>,3},

Node 3 - {fragment,<<"of years and developed">>,4},

Node 4 - {fragment,<<"complex cultures before European">>,5},

Node 5 - {fragment,<<"colonists began to arrive">>,6},

Node 6 - {fragment,<<"United States for thousands">>,3},

Node 7 - {fragment,<<"complex cultures before European">>,5},

Node 8 - {fragment,<<"colonists began to arrive">>,6},

Node 9 - {fragment,<<"what is now the">>,2}]

If the following message is sent from the master node,

Node0! {search, <<"to">>}

Node 5 and Node 8 respond to master node with their PID’s that they have the word in their fragment.

**Find the most frequent word in the text file F and store it at node 0.**

As a part of the gossip process, data is continuously shared between nodes/workers. When the master node wants to know the most frequent word in all of the fragment list, it sends a frequent word message to Node 0. Node 0 stores the frequencies of all words that are distributed in the system. Node 0 is updated continuously through the gossip process. When master node requests the frequent word in the File, Node 0 sends back the most frequent word. Below is the algorithm of the process,

**Algorithm**

MostFreq(**DataWithFreq, Dictionary**)

*If this is Node 0*

*Update* ***DataWithFreq*** *with* ***Dictionary***

*Else*

*Transmit the* ***DataWithFreq*** *to Node 0*

When a process is spawned along with the start module, it checks if it is Node 0. If it is Node 0, it creates a dictionary of words using the fragment of Node 0 and its frequencies. This dictionary is maintained to track the word frequencies of every update. All the workers send the frequencies of their own fragment to Node 0. Node 0 updates the list as and when it receives a request from the worker nodes.

**Sample Output**

Let us look at an example for the same fragment list used above,

Node 0 - {fragment,<<"Indigenous peoples lived in">>,1},

Node 1 - {fragment,<<"what is now the">>,2},

Node 2 - {fragment,<<"United States for thousands">>,3},

Node 3 - {fragment,<<"of years and developed">>,4},

Node 4 - {fragment,<<"complex cultures before European">>,5},

Node 5 - {fragment,<<"colonists began to arrive">>,6},

Node 6 - {fragment,<<"United States for thousands">>,3},

Node 7 - {fragment,<<"complex cultures before European">>,5},

Node 8 - {fragment,<<"colonists began to arrive">>,6},

Node 9 - {fragment,<<"what is now the">>,2}]

After time 0, it creates the following dictionary in Node 0

Dictionary = {1, {<< Indigenous>>, 1}, {<< peoples>>, 1}, {<<lived>>, 1}, {<<in>>, 1}}

After time t1 if Node 2 shared its frequencies for words, it produces the following fragment,

Dictionary = {1, {<< Indigenous>>, 1}, {<< peoples>>, 1}, {<<lived>>, 1},

{<<in>>, 1}

2, {<<what>>, 1}, {<<is>>, 1}, {<<now>>, 1}, {<<the>>, 1}

}

**Update the contents of fragment i at each node that may have a copy of it**

The master node sends an update message to Node 0 along with the fragment number and the new content for the fragment. Node 0 forwards the message to a random neighbour in its list. If the fragment number matches, an update is made to the existing fragment with the new fragment. Longest word and frequencies for word are then updated for the new fragment. Below is the algorithm of the process,

**Algorithm**

*processUpdate(****Data, Update****) ->*

*if* ***Data.fragmentNum*** *=* ***Update.fragmentNum***

*Update fragment, word frequencies, longest word and the last updated value.*

*Else*

*Update the last updated value.*

*If*

*If this node is the target node*

*Return the new Data [Updated with New Dictionary]*

*Else*

*reportWF(****Update****.Node0,****Update****.FragmentNum,****Update****.wordFreqs),*

*reportWF(****Node0, FragNum, Freqs****)*

*Node0.pid ! {****wfReport, FragNum, Freqs****}.*

**Sample Output**

Let’s consider the following example,

Node 0 - {fragment,<<"Indigenous peoples lived in">>,1},

Node 1 - {fragment,<<"what is now the">>,2},

Node 2 - {fragment,<<"United States for thousands">>,3},

Node 3 - {fragment,<<"of years and developed">>,4},

Node 4 - {fragment,<<"complex cultures before European">>,5},

Node 5 - {fragment,<<"colonists began to arrive">>,6},

Node 6 - {fragment,<<"United States for thousands">>,3},

Node 7 - {fragment,<<"complex cultures before European">>,5},

Node 8 - {fragment,<<"colonists began to arrive">>,6},

Node 9 - {fragment,<<"what is now the">>,2}

Master node sends the following command,

Node0! {update, 3, <<" Hello this is the new fragment ">>}

The nodes where the fragment 3 is present is then updated to this content.

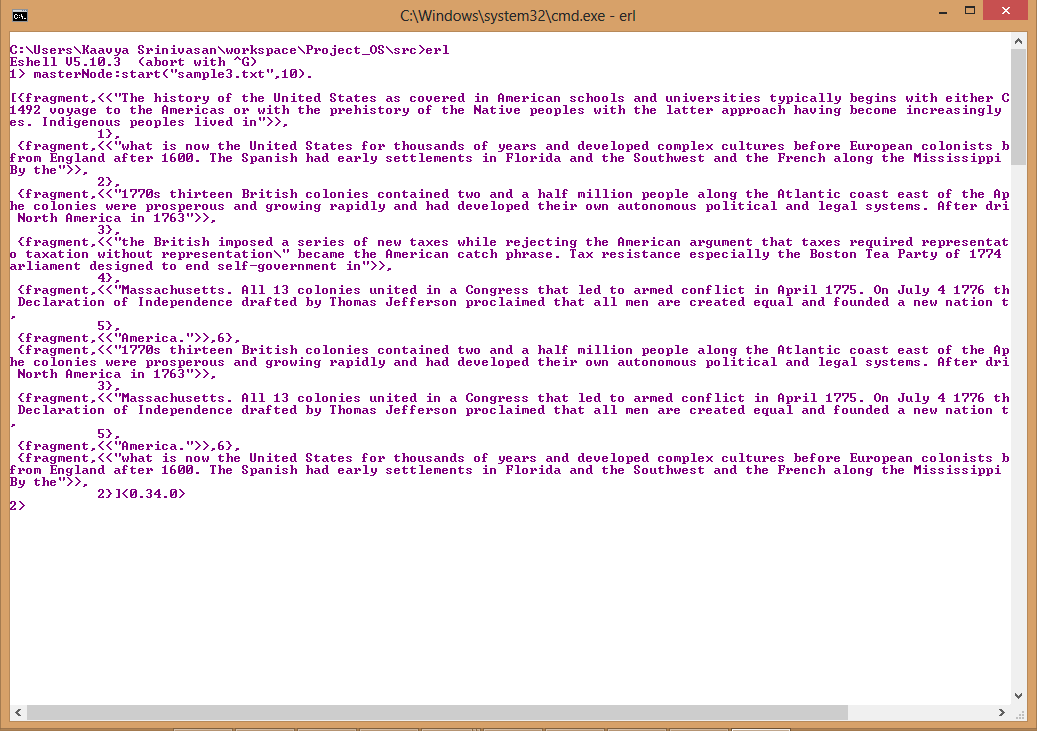
Node 2 - {fragment,<<" Hello this is the new fragment ">>,3}

Node 6 - {fragment,<<" Hello this is the new fragment ">>,3}

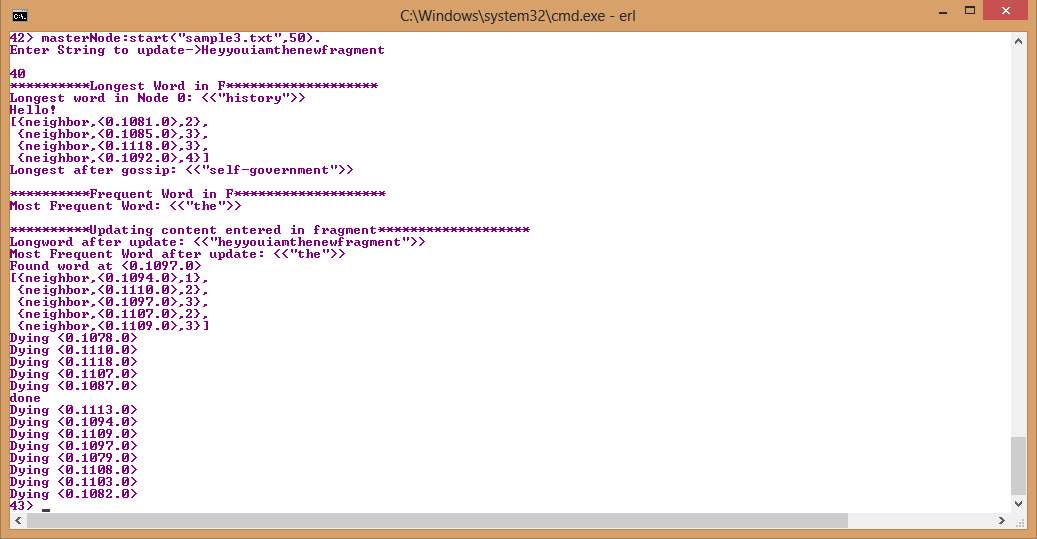
**Screen shots for each Task**

Splitting file into fragments, following is the sample file used,





Combined output of the other tasks for the same “sample” text file used above.



References

[1] Wikipedia’s Link on Gossip Protocol - <http://en.wikipedia.org/wiki/Gossip_protocol>