CMSC 621 Erlang Gossip Project Report

Yin Huang, Anuja Kench, Shrinivas Kane, and Abhishek Sethi
Professor: Kostas Kalpakis
University of Maryland, Baltimore County
Computer Science and Electrical Engineering
{yhuang9, akench1, skane3@umbc.edu}

I. Introduction

Our implementation of Gossip-based aggregation and data update and retrieval in Erlang aims at performing the following four computational tasks as required in the project description.

A single large file F with floating-point numbers have been split into M fragments F_1, F_2, , F_M which have been placed at various nodes of the system. Each fragment is placed at one or more nodes.

- Compute the minimum (min) and maximum (max) value in F and store them at Node 1.
- Compute the average (avg) of the values in F and store it at all the nodes.
- Update the contents of fragment i at each node that may have a copy of it.
- Retrieve the contents of fragment i from any node that may have an up-to-date copy of it.

The report is organized in the following sections: Section II is the two types of network deployed in our experiment, Section III explains the gossip algorithm, Section IV focuses on the min, max, avg, read and write operation. Section V demonstrates the research results. Section VI is the analysis and discussion, followed by conclusion and future work in Section VII and Section VIII.

II. Network types

Two types of network are simulated with two virtual machines hosted on two different laptops:

- Ring (Figure 1)
- Mesh (Figure 2)

The ring topology is a static topology, and assumes that no nodes have failed or caused an exception. Each node contains the listed variables/lists shown in figure 1. This information is still light weight in terms of memory usage, allowing the processes in Erlang to remain light weight.

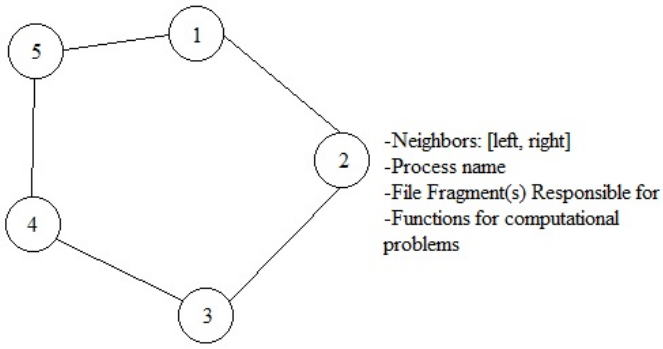

Fig. 1. Diagram of the Ring network. An example of the filled in data structures for node 2 would be the neighbors are [1, 3], process name is P_2, fragments would be [F_2, F_5, F_1], and then the functions for min, max, average, read, write are found in the last part

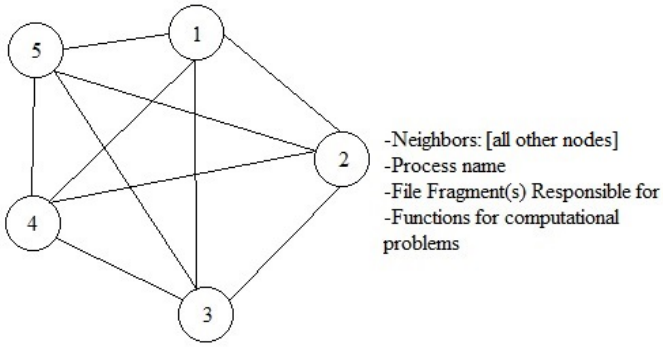

Fig. 2. The Mesh network used in the application. The only difference from Figure 1 in terms of data saved is that there are more neighbors now. For example, node 2s neighbors would be [1, 3, 4, 5] now.

The Mesh network is similar to the ring network, except in the Mesh network, each node is connected

to each other, meaning each node is assumed to know about every other node in the network. Figure 2 diagrams the Mesh network used here.

The key idea with both topologies is that they are overlay networks. These overlay networks reside on multiple physical machines. For future reference, the physical machines will be called network machines, and each node is represented as an Erlang process in the application. In order for true distribution, these nodes were not spawned on a single network machine. Instead the nodes were created across multiple network machines. In order to ensure an even split of the nodes, thus hopefully distributing the load evenly, the nodes were assigned based on modulo arithmetic.

In our implementation, we define a node as an Erlang process with process name P_# and each node stores its fragments F_#. Every node maintains a neighbor node list. This list differs as in these two types of networks. However the gossip mechanism are the same. A computational task request is initialized at node 1, and node 1 will start spreading out the request to its neighbors. Whenever a neighbor node receives a request, this node will first exchange information with the requesting node, and then spread the request to its neighbors. In such a way this request will eventually reach out to the whole network. In order to pick a neighbor, we use a random number generator to pick a node from the neighbor list. Meanwhile, the user can specify the number of iterations for every node to send out the gossip.

III. Gossip algorithm

In large distributed, heterogeneous networks, applications are asking for a set of functions that provide components of a distributed system access to global information such as network size, average load etc. Gossip-based aggregation protocol works in a fully decentralized manner where all nodes exchange the local information with its neighbors and thus infer the global information. There are two types of protocols, reactive and proactive protocols. We focus on the proactive protocol which pushes the query from any issuer to all nodes in the system.

We use asynchronous push-pull gossip protocol in our implementation. The task of a proactive protocol is to continuously provide all nodes with an up-to-date information held by current set of nodes. Two services are provided in each node, one is to receive request, the other to update information and send out the new information.

Every node maintains a neighbor list as its neighbor nodes. Whenever a node receives a request message, it will randomly pick up a node in the neighbor list, and forward the request. We set up a threshold parameter named Itr to determine the number of iterations. So the gossip for each node will be executing Itr times. We will report our convergence ratio in our result section with regard to the size of the network, topology of the network, and the size of the file, number of segmentation of the file etc.

IV. Computation implementations

A. Min, Max, Avg implementation

In order to collect the global minimum (min) and maximum(max) value in the distributed system, we adopt a push-pull model for computing the min and max. Initially each node will have a local min and max stored in memory, say process dictionary in Erlang. When a request for computing either is received for a node, this node will begin to gossip local min/max to one of its neighbor from its neighbor list. Whenever a request is received, the node will start comparing the current local min/max with received value and update its local min/max. In such a way we claim that a global min/max has been reached when the local value becomes stable. Experiments demonstrate the efficiency for both networks.

For average, the problem is a little different from previous cases due to the replicas. In our implementation, we first compute a local sum for the file as well as the local count of the number of values in the fragment. Whenever a request is received from a neighbor, we update the local sum to the sum of both segmentations as well as the local count of the number. When the local sum and local count at every node remains the same, we claim we have reached the global information, and thus the average is computed by dy dividing the sum by the count. Experiments will show how fast we get the average and how accurate the average is compared to the real average.

B. Update and retrieval

We have wrapped both the read and write operation in a single process called Update(). The reason to put these two operations in the same process is to avoid the problems incurred by multiple accesses to the same file. Update() will only execute either write or read to the segmentation. Since Erlang has the FIFO mailbox, we simply utilize this mailbox as a job request queue. This approach, however, sacrifices the consistency since we fail to ensure the write update has been done successfully to all replicas. This is a trade-off in a gossip-based approach since we have no the information regarding the locations of replicas.

Whenever a process receives a write or read message, the process will check its own data directory to see if it contains the target segmentation. For write operation, if the process contains the segmentation, it will continue and update the segmentation, else it will pick a random neighbor and forward the request. For read operation, if the process contains the segmentation, it will reply the values of the segmentation to the request node, otherwise it will forward the request to its neighbor. We have two operation modes: a. synchronized b. non-synchronized. In Synchronized case, the request node will wait for all replies for target nodes until it stops gossiping. In non-synchronized case, the request node will not wait for any replies but rather work on other things. The synchronized case gives better consistency while the non-synchronized has better performance (time response). The former is more complicated than the latter. Currently we only implement the non-synchronized mode.

In our implementation, we take the non-synchronized mode where the write request is sent out and gossip a fixed number of iterations while the read request is only waiting for the first reply from any matching node that has the segmentation. The advantage of such design is that we can guarantee that no race condition for the file will occur, hence a higher accuracy. The disadvantage is the response latency because every time only one node is allowed to do operation on the fragmentation.

V. Experimental results

Our experiment is conducted in two laptops with two virtual machines, each VM has a single core of i7 processer with 2.5GB RAM. We have generated a file with 3000 floating-point numbers. And we have tested the performance on both network types in terms of the accuracy of average, the gossiping messages exchanged in the network, and the time to converge. And we mainly focus on the min, max, and average operations. The following graphs show both the exchanging gossip messages and time to converge with regard to different numbers of processes.

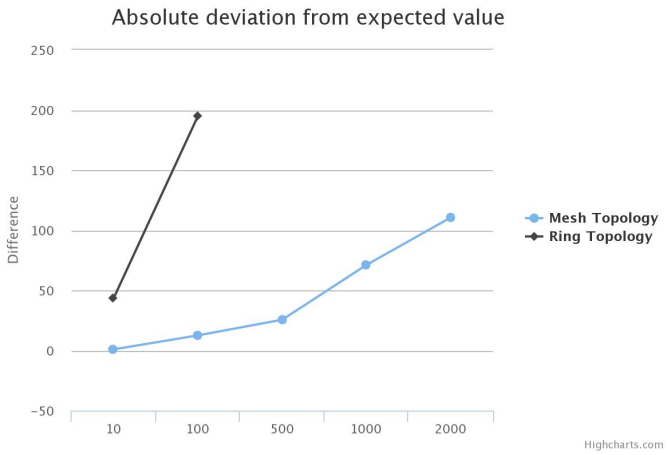

Fig. 3. Absolute deviation from expected value

Figure 3 shows the devation of average from expected value, which is a indicator of accuracy. In Figure 3, we can see in the Mesh network we have better accuracy compared to the Ring network. Mesh network tends to render stable accurate results whereas the ring network tends to render less accurate results.

Figure 4 and Figure 5 shows the number of messages it takes to converge for Ring and Mesh. As we can see it take almost the same messages in both cases to find the min and max. However for the average, the Ring would take significantly more messages than the Mesh to converge. And overall the average operation takes more messages to converge. The reason behind this is the way we compute the average is different than min and max. To compute average, we need both the sum and

count to be stable to consider as convergance. But in Mesh it takes less messages to compute averge is because the Mesh gives nodes more opportunity to communicate with new nodes in the network since every node knows all other nodes as opposed to two neighbors in the Ring.

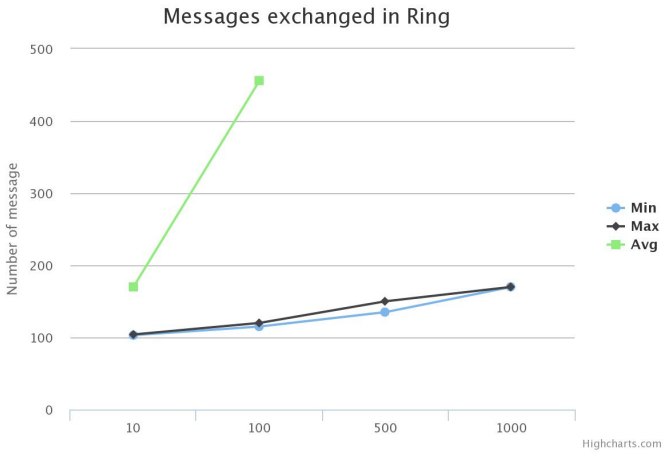

Fig. 4. Messages exchanged in Ring

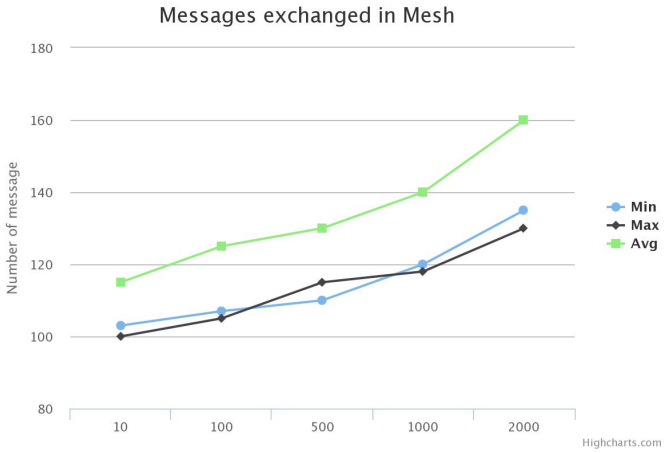

Fig. 5. Messages exchanged in Mesh

Figure 6 and Figure 7 shows the time to converge for the Ring and the Mesh respectively. It takes less time in the Ring than the Mesh to converge. And Figure 7 shows a very abnormal behavior regarding the converging time before reaching 500 nodes. We do not have a good explaination for this for now as it should be linear to the size of the network.

In Figure 3 and Figure 4, there are only two nodes for the Ring network. The reason lies in the way average is computed by summing the local values from two neighbors. The more exchanging messages, the larger the value becomes. And this soon exceeds the number limit in Erlang and causes errors.

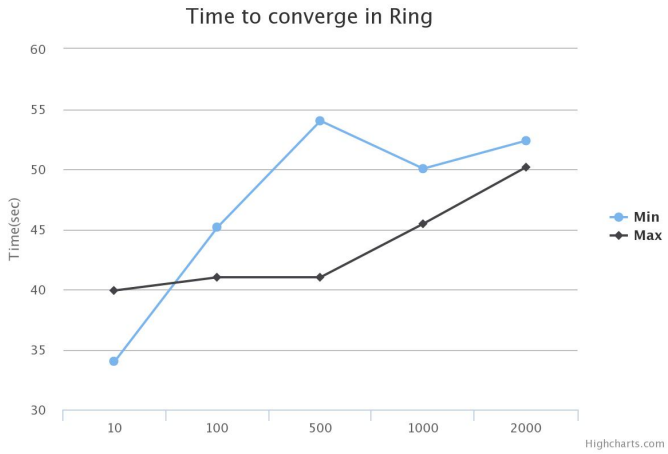

Fig. 6. Time to converge in Ring

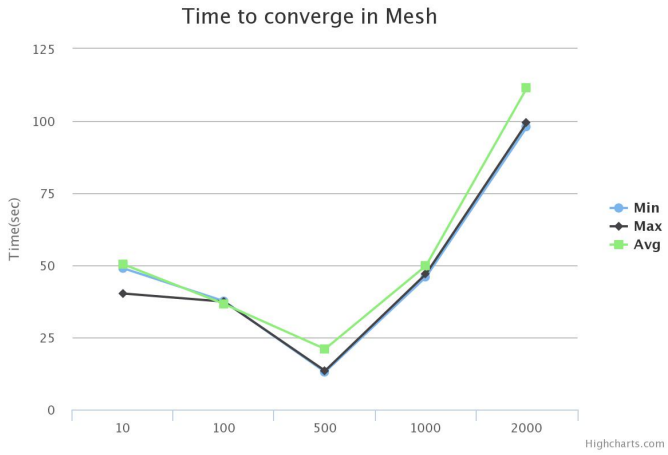

Fig. 7. Time to converge in Mesh

VI. Analysis and Discussion

The key in the Ring network is that each node only has 2 neighbors to gossip with, drastically reducing the number of neighbors to choose from compared to the Mesh network. This makes communication quick and effective. The problem here would be if for a large ring, the file that is needed for reading is on the other side of the ring. For example, if the entry point is node 4, but the file resides in node 2, and if 4 randomly gossips with node 5, then 5 gossips with 1, and 1 gossips

with 2, we finally reach our destination. For this small of a ring, it is not an issue, but when the ring reaches sizes of 10,000 and more, latency increases. The ability to have 2 neighbors is beneficial, but it also may cause increases in latency in worst case scenarios.

The key in the Mesh network is that each node supposedly knows the existence of all the other nodes in the network. It is also a static topology, just like the ring network. The issue here would be that during gossip, latency would increase greatly with the number of nodes. In the worst case scenario, since a random neighbor is chosen to gossip with, if that neighbor does not end up being the node that contains the desired fragment, it is possible that the message passes across all the nodes in the network before it reaches the desired destination. Other than this, the Mesh network proved to be decent for scalability as shown in the results.

Current design for retrieval is not able to guarantee that a read request will always get the latest version, our implementation only accepts the first response without checking the replies from other nodes. Solution to this problem comes in the following directions. First, we could maintain a list of replicas for each fragmentation, when update the fragmentation, we send out update requests to all these nodes holding the fragmentation and wait for acknowledgement. We call this schema write-to-all and read-once. The problem with this solution is the high latency for the update operation. Also we need global information about the nodes that hold the replicas. Second, we could use a vector clock to record the number of updates for each fragmentation, and read the reply with the highest number.

To solve the consistency problem, we can potentailly use a quorum-based approach to access the same fragmentation, for the replicas, we also need such information stored in a master node which initializes the file distribution. This global information, however, defeats the purpose of gossip approach in the first place. Alternatively, we could use gossip message exchange to find the replicas for each segmentation, and build this global replica table for all nodes.

VII. Conclusion

In this project, we have successfully implemented Gossip-based aggregation and data update and retrieval for a large distributed system using Erlang. The three computational tasks have been demonstrated in terms of efficiency, effectiveness and accuracy. We have compared the convergence performance of gossip in different types of networks, say ring and mesh, and different sizes of networks. The time and exchanging messages to converge in both networks are linear to the size of the network. However, in the Mesh network it took longer to converge than Ring network. In addition, Mesh network tends to give better accuracy than the Ring. To sum up, in the Ring network, it is faster to converge but renders less accurate result whereas in the Mesh network, it is slower to converge but renders more accurate result. Both performance are linear to the size of the network.

VIII. Future work

- Current computation of average does not take into consideration of the replicas of the fragmentation, we would consider to differentiate the values from the same fragmentation, and only get the sum of different fragementations and the count of total fragmentations. This, however, will cause longer latency
- Current retrieval operation only wait for the earliest reply when the specified fragment is found. Based on different applications, we could extend this to collect all replies.
- Current update operation is based on a gossip approach which fail to ensure the update of all copies of the file fragments. We can either gossip the system to collect the location of the replicas, and then synchronize the update operation to all copies to ensure consistency, or we can use a global table to store this information when the file is distributed.
- Implement quorum-based approach to deal with the mutual exclusion problem caused by multiple access to the same fragmentations.
- Deal with node failure and message loss by setting time outs.

REFERENCES

[1] Tanenbaum, Andrew S., and Maarten Van. Steen. “Distributed Systems: Principles and Paradigms.“ Upper Saddle River: Pearson Prentice Hall, 2007. Print.

[2] Armstrong, Joe. “Programming Erlang: Software for a Concurrent World.“ Raleigh, NC: Pragmatic helf, 2007. Print.

[3] Demers, Alan. “Epidemic Algorithms for Replicated Database Maintenance.“ Palo Alto, Calif: XEROX Corp., Palo Alto Research Center, 1989. Print.

[4] Damon Mosk-aoyama and Devavrat Shah. “Fast distributed algorithms for computing separable functions.“ IEEE Trans. Inform. Theory, pp 2997-3007 2007

[5] Demers et al. “Epidemic algorithms for replicated database maintenance.“ 1987

[6] Karp et al. “Randomized rumor spreading.“ 2000

[7] Jelasity et al. “Gossip-Based Aggregation in Large Dynamic Networks.“ 2005

[8] Erlang, http://www.erlang.org/