IMPLEMENTATION OF GOSSIP ALGORITHMS WITH ERLANG: DESIGN, IMPLEMENTATION AND ANALYSIS

Akshay Peshave
peshave1@umbc.edu

Karan Gill
kargil1@umbc.edu

Milind Patil
milindp1@umbc.edu

Veeresh Halagegowda
vp67051@umbc.edu

Computer Science Department, UMBC

Abstract
Information exchange and message communications in an arbitrarily connected network of nodes in a distributed asynchronous environment operate under limited computational, communication and energy resources, we analysis the gossip algorithms under the above circumstances where in a node communicates with a randomly chosen neighbor.

Our main focus of this implementation is to analyze our algorithm to find the time required to arrive at a desired average implying this to the real world scenario of how fast the data disseminates in an arbitrarily connected network of nodes. To analyze this we define ring, chord network topology and a fully connected graph each connected to log (n) nodes, where (n) being the total number of nodes.

Apart from the averaging, our experiments also include finding the Min, Max, Median and the time required to arrive at all these values.

Introduction
The advent of sensor, wireless ad hoc and peer-to-peer networks has necessitated the design of asynchronous, distributed and fault-tolerant computation and information exchange algorithms. This is mainly because such networks are constrained by the following operational characteristics: (i) they may not have a centralized entity for facilitating computation, communication and time-synchronization, (ii) the network topology may not be completely known to the nodes of the network, (iii) nodes may join or leave the network (even expire), so that the network topology itself may change, and (iv) in the case of sensor networks, the computational power and energy resources may be very limited. These constraints motivate the design of simple asynchronous decentralized algorithms for computation where each node exchanges information with only a few of its immediate neighbors in a time instance (or, a round). The goal in this setting is to design algorithms so that the desired computation and communication is done as quickly and efficiently as possible.

By gossip algorithm, each node communicates with its neighbor only once at each time slot. Thus, given a graph G, we determine the averaging time, which is also topology dependent.

Finally, we study the performance of the gossip algorithm by experimenting in four different topologies: A ring topology, ring topology with may nodes, A CHORD like node implementation, fully connected graph and lastly.

1. Motivation
Considering a system that has N nodes and a single large data structure F with floating-point numbers. The data has been split into various fragments, which are placed at various nodes of the system. Each fragment is placed at one or more nodes. The fragments may not all have the same size in this case different values. The motivation is to perform the following three tasks:
a. Compute the minimum and maximum value in F and store them at node 1.
b. Compute the average of the values in F and store it at all the nodes.
c. Compute the median of the values in F and store it at all the nodes.
d. 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.
e. Retrieve the contents of fragment i from any node that may have an up-to-date copy of it. The retrieval is requested by node 1, with the user specifying the fragment number.

2. System Model

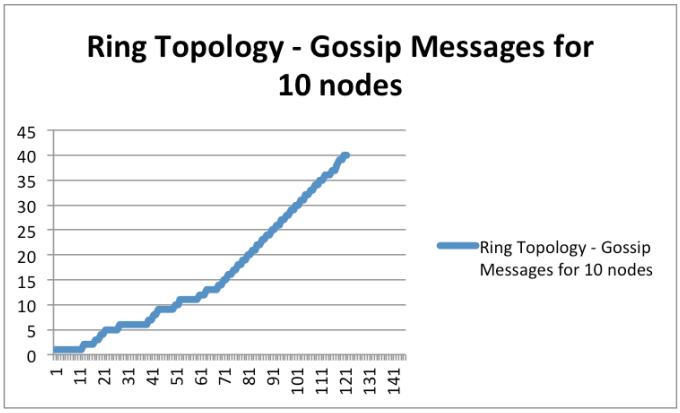
We consider a network consisting a large number of nodes that consists of unique identifiers called the PID and communicate through message exchanges. We assume that none of the nodes fail and the network is fixed once the algorithm is initiated, i.e. the node count will not change. However, please do note that even if nodes fail, our implementation will not stop working and although the current computation might skew the results, in the next computation, results will be accurate once more. Most of our algorithms also take replication into account and perform fairly accurately with them in mind. The message communication may have certain amount of delays to ensure the overhead of message loss.

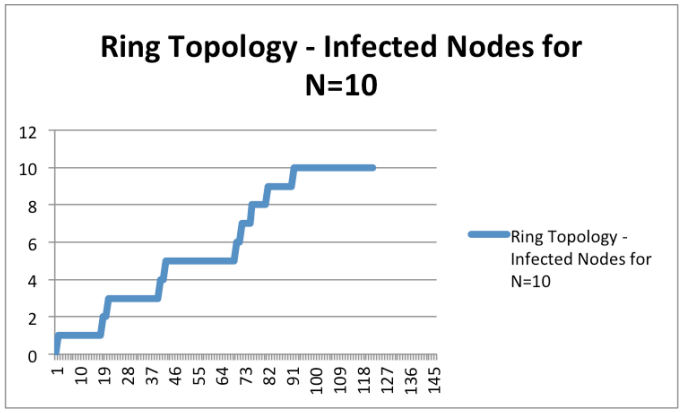
3. Topologies Considered

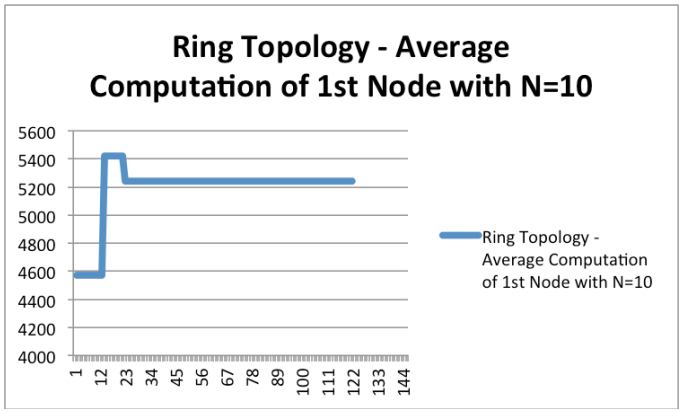
For experimental purpose we have implemented a ring topology, ring topology with many nodes, a CHORD like node implementation and a fully connected graph. We will be explaining the topologies in detail in this section.

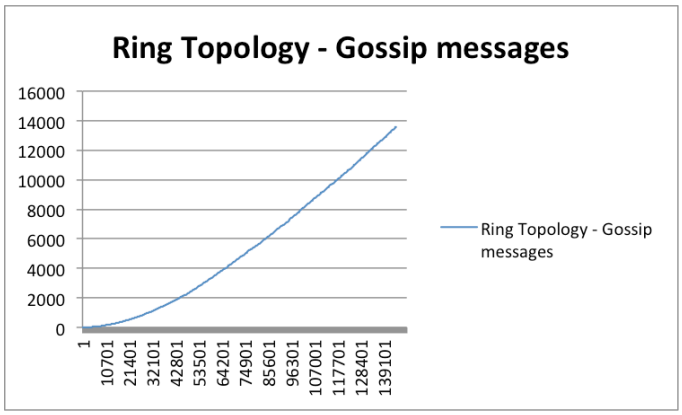
A. Ring Topology

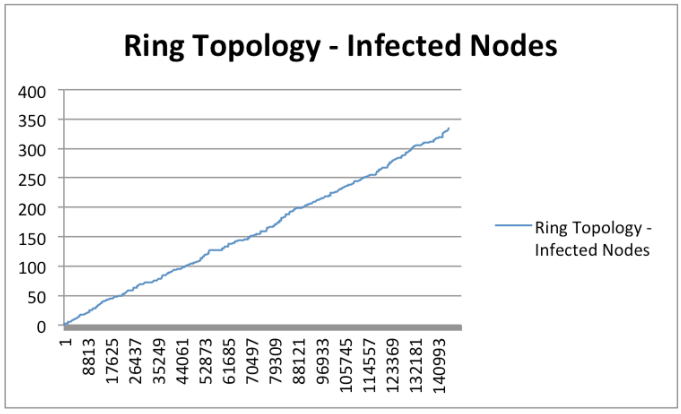
In this topology each node is connected to a maximum of one other node, forming a network of ring structure. Consider a network of four nodes a, b, c and d. Here a->b, b->c, c->d and d->a, are the connected nodes. The data dissemination starts from 'a' node spreading the information to b and so on. The disadvantage of this topology is that the convergence does not occur quickly and is a constant after a few iterations. This topology cannot be used for large number of nodes.









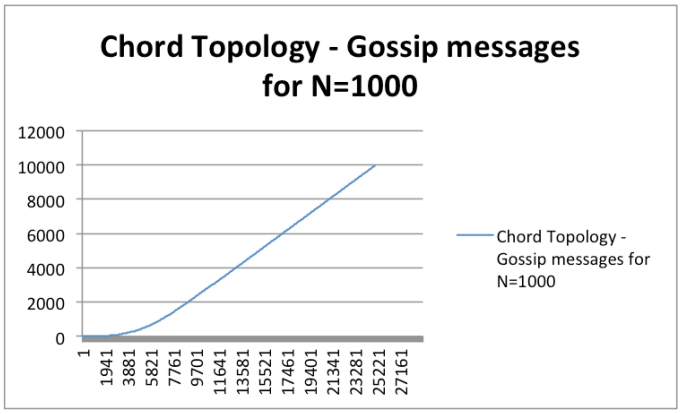


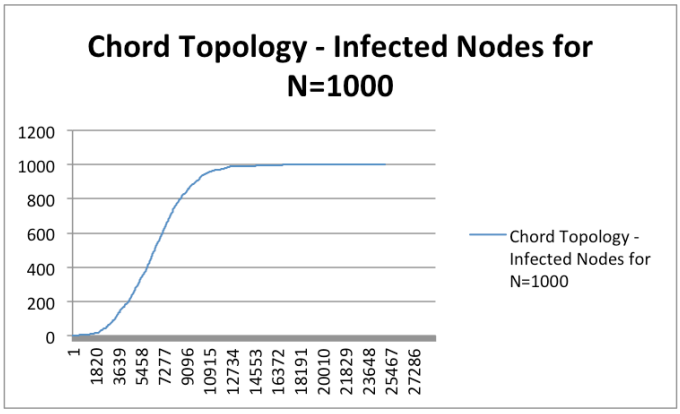
B. Ring Topology (2 and 3 Edges)

In this topology each node is connected to previous node and two to three corresponding nodes forming a bigger ring topology. Consider a network of six nodes a, b, c, d, e and f. Here a->f, a->b, a->c, b->a, b->c, b->d and so on. The data dissemination starts from node 'a' spreading information to all connected nodes, but one node at a time. The convergence happens much quicker than the normal ring topology and the value moves towards the intended value. This topology can be used for slightly larger networks for optimal efficiency.

C. Chord-like Topology

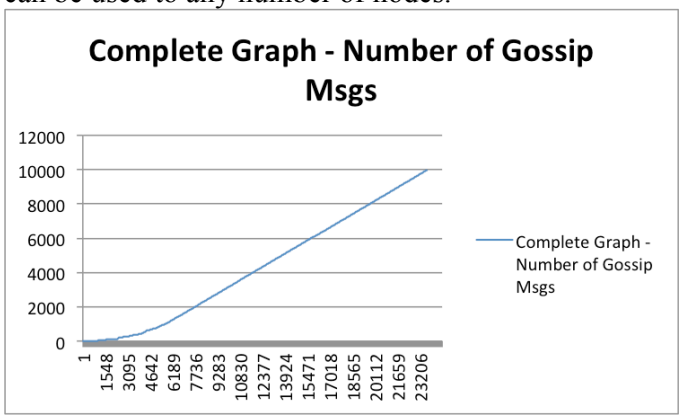
In this topology each node is connected to log (n) number of nodes where in each node is connected to other nodes by plugging in values (n + 2 pow(k-1))mod 2 pow (m) and log (n) connections to itself there getting the probability of contacting itself to 0.5 . The data dissemination starts from node 'a' and converges very quickly with accurate values and given message frequency.

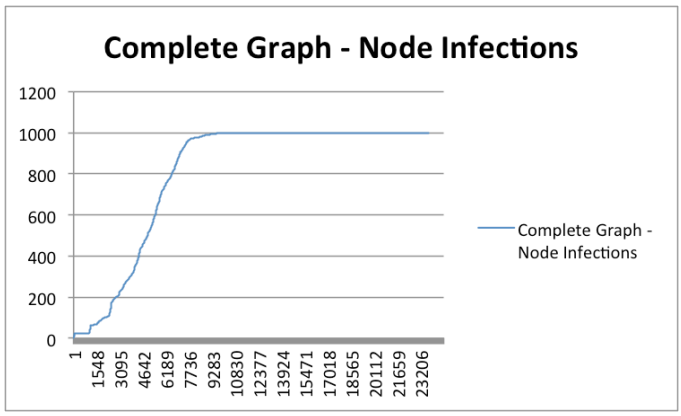




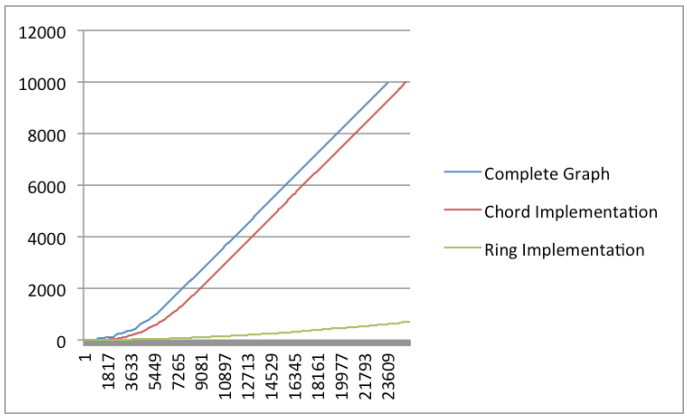
D. Fully Connected Graph

In this topology each node is connected to all other nodes. The probability of a node getting infected to itself is 1/N and rest of them with equal probability. The data dissemination starts from node 'a' and converges very efficiently with accurate values and given number of message frequencies. This topology can be used to any number of nodes.

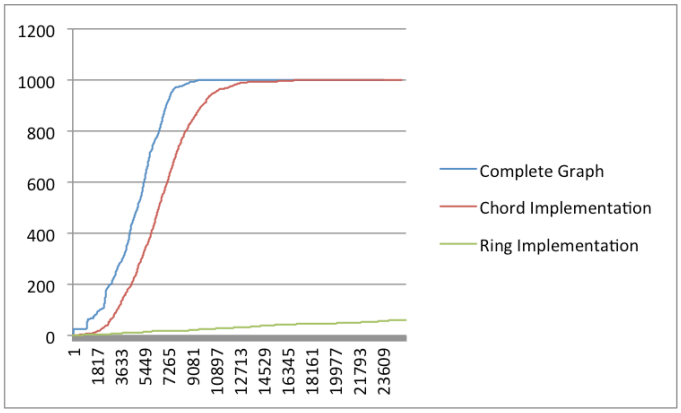




E. Comparison of the Topologies with N = 1000



Gossip on various topologies



Infections of nodes on various topologies

4. Implementation of the Given Problems:

A. Implementation logic:

We have different Erlang files for every problem provided and later on we will provide problem specific logic in every problem, but for the most part, the implementation logic is exactly similar. Also, note that the code for problems 1, 2, 4 has been implemented differently than 3 & 5. We will also discuss the details as and when the need arises.

The bootstrapper code spawns Erlang processes on the 2 VM's and also initializes them with some randomly generated data & later on with their neighbor lists. Once initialized, all processes are in a state of stasis, i.e. inactive. The bootstrapper then finally triggers a computation on node 1 thereby infecting the node 1 which then spreads the "infection" throughout the system.

The data structures for the problems considered includes given N number of nodes we are generating for each node, a list of floating point numbers in the range in the range of [1-10000] and the size of list is randomly chosen between [1-30] i.e. every node in N will have [1-30] values in the range [1.5 – 10000.5].
At the end of every execution we also print out the statistics of the data that we have randomly generated (min, max, average, median) for verification purposes. The output was analyzed by redirecting the Linux shell output using the Linux tee command to a file & then the file was analyzed in excel and graphs generated. Now you know why we were only able to plot graphs up to 1000 nodes, although for most of our problems, our implementation ran just fine even for 10k-20k nodes (see the gigantic output file attached for gossip12.erl which runs problems 1 & 2 for 10k nodes with almost negligible memory requirements).

The basis for our gossip implementation can be explained thus, once all the N processes are spawned, they're all "uninfected". At this point, every Node is actively "pulling" data from other nodes every Delay seconds, to see if there is a "secret" or a computation that it needs to be a part of (which of course at the start means they get no replies). Once Node 1 is infected by our bootstrapper, the Node 1 has a computation to perform & thus it stops pulling and enters the "push" mode instead. In every Delay seconds duration, it will choose a neighbor at random (its a simplistic probabilistic implementation) and push it the computation data, thereby infecting that node as well. At the same time, any node that is uninfected continues pulling data from its neighbors & if it happens to contact an infected one, it gets infected as well.

We have an additional concept of KCount, which you may have noticed while starting the bootstrapper. It is more or less a construct we've used so as to send only K*log N messages and pause so that we can take the snapshot of the state. In real-world implementation, KCount would rather be a part of the computation task itself & would rather be defined in terms of time, because of course, size of the network varies in a P2P system and log N is just a estimate here.

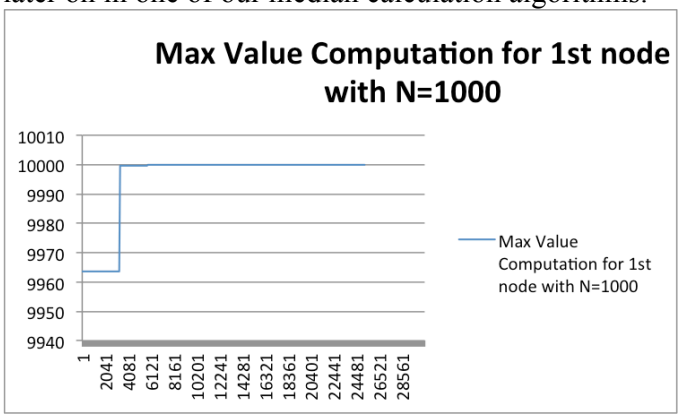
There is an additional tuning parameter called Delay, which is used extensively in our code, which is essentially the tuning parameter, which dictates the time to wait between push/pull rounds & time to wait for a response for a push request.

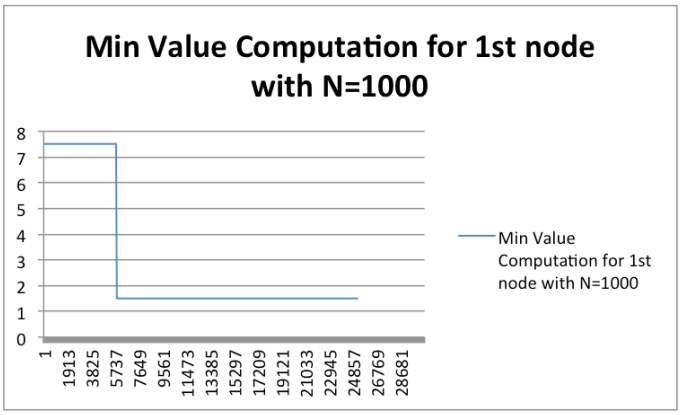
The KillTime parameter in our code is simply the time after the infection initialization before all processes are sent a {exit} messages, receiving which all of our processes simply print their states & exit (so that they're GC'ed etc).

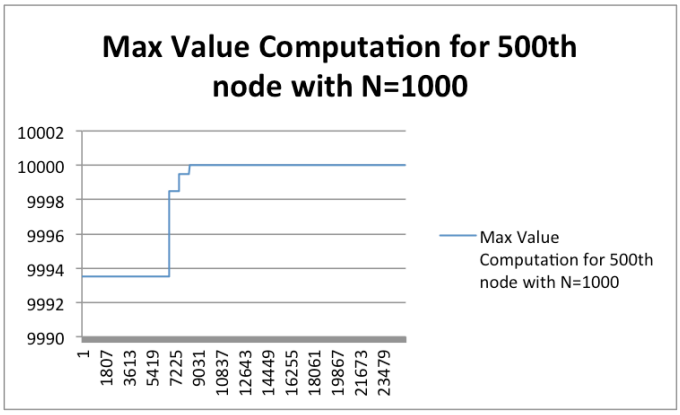
A special note about message counts. For a single push, the overall cost is 2 messages, the push message itself & the response message. For a pull, however, the overall cost is 1 or 3 messages, the pull request to a node. If the node is not infected, it does not respond back, or if it is, it initiates the push paradigm. Thus, please note that, in our graphs, we've plotted statistics against the ACTUAL cost of messages and not the number of PUSH/PULL.

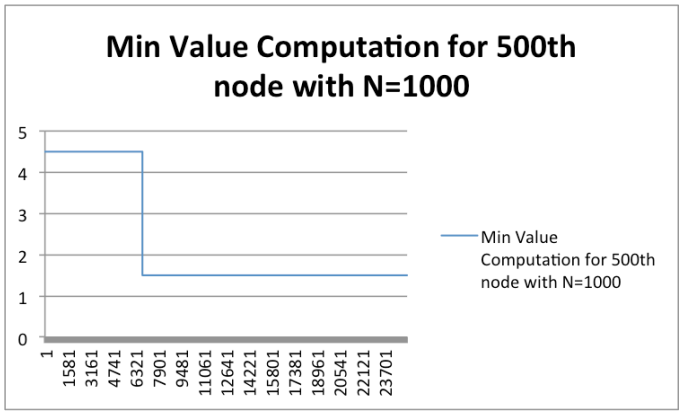
B. Compute the minimum and maximum value in Fragments F and store them at node 1:

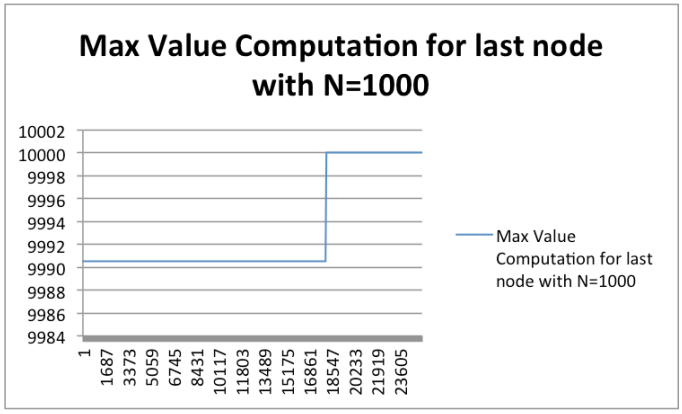
The easiest algorithm we had to implement amongst the 5 probably. The memory requirement is really low & given an appropriate topology, it converges really fast. Implementation is shown in gossip12.erl. A useful byproduct of our implementation (and I suppose everyone's) , is that by the end of the epoch (i.e. the computations lifetime), with a very high probability , ALL the nodes in the network will end up having the Global minimum & maximum. As you will note later, this fast & efficient algorithm will be used later on in one of our median calculation algorithms.

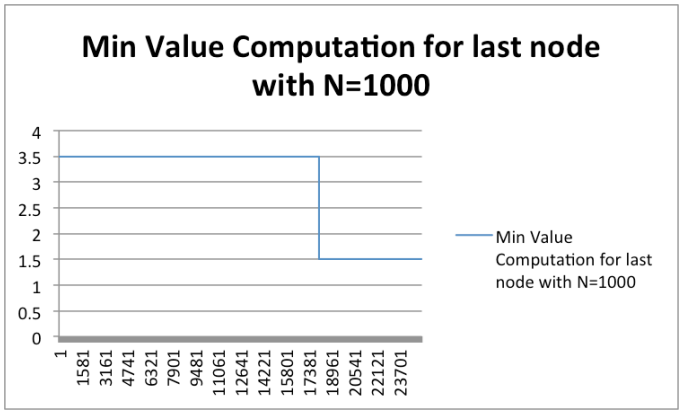












Algorithm

a. Every node stores 2 additional variables, MIN & MAX, which initially are the node's local minimum & maximum.

b. In every gossip exchange, these 2 variables are exchanged, and the end of an exchange, both nodes will store the minimum & maximum.

c. Now provided that the network is completely connected & enough rounds are allowed to pass, we find that every node's MIN & MAX will move monotonically towards the globally correct values.

Results

As the graphs above & in section 3 show, Node 1 gets the value of Global minima & maxima much before log N rounds of gossip have completed in a chord-like

topology, & very slightly quicker still in a completely connected graph. In ring-topology however depending on where they are situated, convergence is much slower & not guaranteed in a single set of log N rounds, which is expected.

Assumptions

No assumptions were made for this implementation. It will work just as well in case of replication, failures, etc. Only if the node with global minima/maxima fails, will the values be off for that epoch.

C. **Compute the average of the values in Fragments F and store it at all the nodes.**

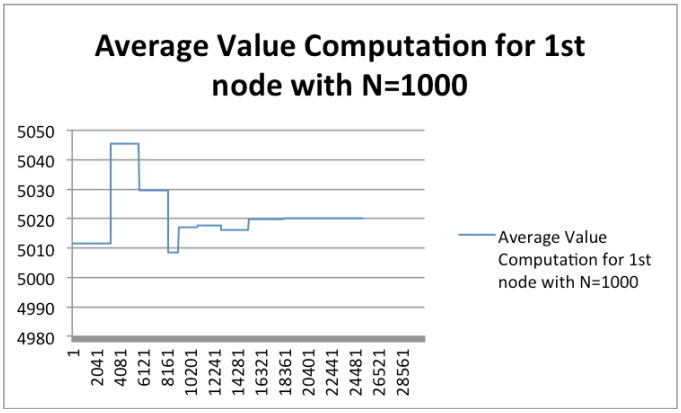
This is a much more interesting case than the rather prosaic min/max problem, especially made much more interesting by the fact that each node hosts a LIST of values & size of the list at each node may be different. Add in the factor of replicas, this problem quickly goes from the rather straightforward textbook question to something we had to mull over for some time before coming up with 2 solutions, which one may use depending on the need of the situation.

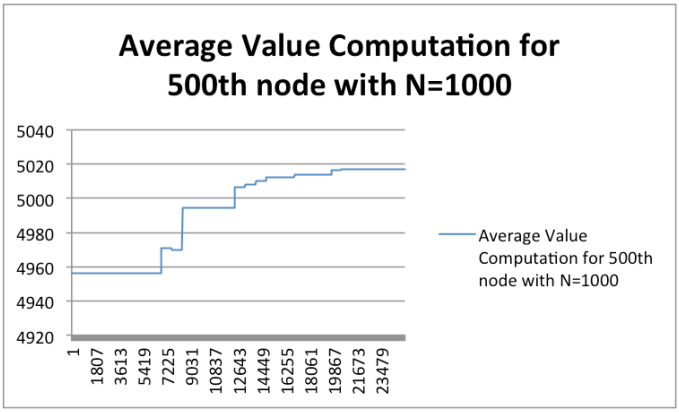
Before we get into the results & the nitty gritties, lets discuss the criterion of this project. Our target is to maximize accuracy while minimizing latency & communication costs i.e. minimizing number of messages as well as minimizing the size of a message.

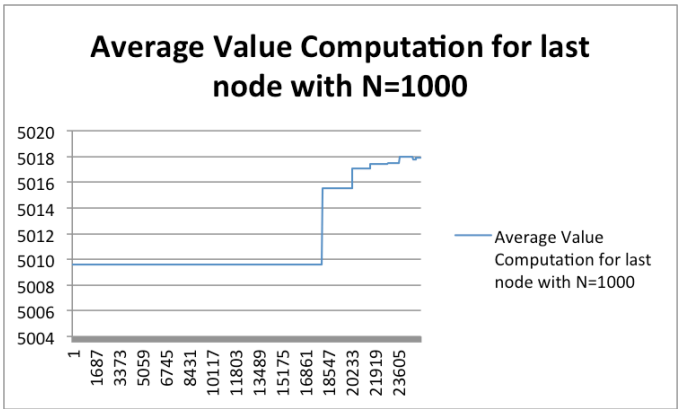
The first algorithm we propose is highly network & memory efficient (infact its communication cost is just as much as the previous min-max problem), but it trades off accuracy with it. Our results with this algorithm are varied. The second algorithm is highly accurate but significantly increases the communication & memory requirements. Depending on the need, an end-user may use either.

Algorithm 1: Sum of Sums & Number of Numbers

a. Every node is initialized with 2 additional variables, Total_sum & Total_num that initially is just the sum & count of the nodes own list.
b. When 2 nodes gossip, they exchange these 2 variables, adding them to their own variables & thereby can compute an average. Essentially, the idea is that after a lot of exchanges, each node will have taken into account all the numbers in the system and can approximate an average out of it.





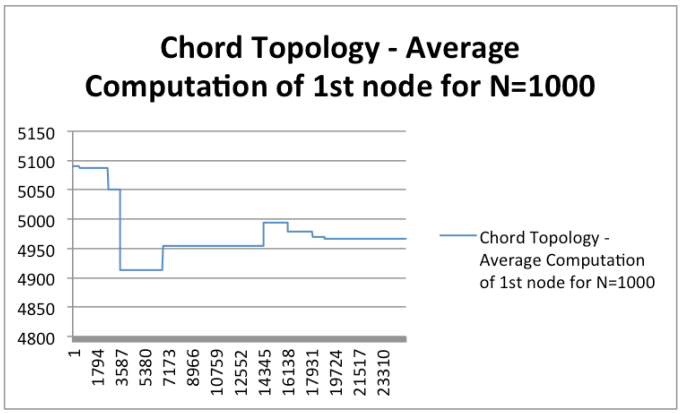


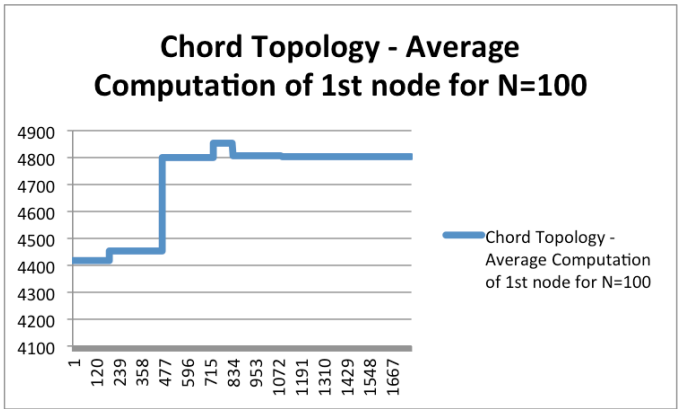
Results 1:

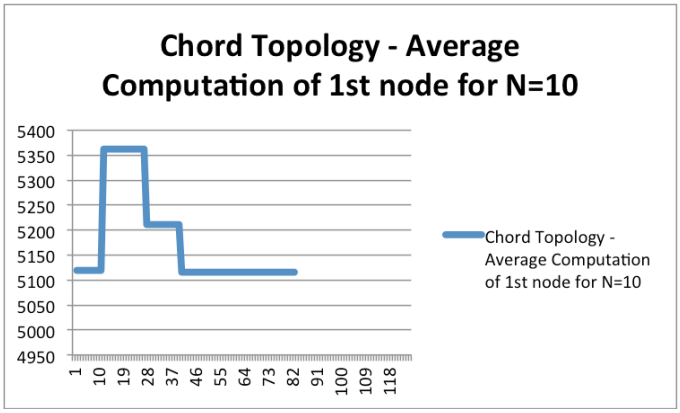
a. This is a highly memory efficient algorithm, implemented in gossip12.erl, as shown in the graphs we have attached below, clearly shows that the values do converge to an approximation of the true global average. As again in problem 1, ring topology fairs the worst, whereas a chord like topology does extremely well & almost as good as the fully connected, although do note that this may simply be because in my code, the probability of a node choosing itself in the chord-like topology is 0.5,whereas in fully its only 1/N.

b. A limitation of this algorithm is that it converges rapidly only if the range of values is distributed without any bias. In our experiments, we saw huge deviations from the global average when the values of each fragments were of the type, [1.5,2.5],[2.5,3.5].....[999.5,1000.5] across 1000

nodes. In such a case, computed average would quickly converge to approximately 186 and then very slowly move upwards towards the true global average that was about 500.







Algorithm 2: Maintain HashMap of fragment id & its sum & length

a. Every node maintains a Dictionary (hash map), whose key is fragment id & value is the fragments sum & length and is initialized with the node's own fragment's information.

b. In every gossip exchange, the 2 nodes exchange their dictionaries and update their own with the information of the fragments they didn't find in their own.

Results 2:

a. Note that the dictionary sizes will soon reach up to size N and thereby each bi-directional transfer would be huge. We optimized this slightly by returning ONLY the difference in the dictionaries when a Node B is sent Node A's dictionary.

b. There are further ways we could optimize this, but we haven't had the time to test it. One way would be to always send only about X randomly chosen fragment's information from Node A to B, and Node B would send back some Y randomly chosen frag info from its own dict. This would greatly reduce n/w transmission in a single burst, but will probably lead to delayed convergence.

c. In all our test cases, apart from the ring topology, we always converged to the true global average across ALL nodes well within log N rounds of gossip (shown in the graphs).

d. Even with any degrees of replication, this algorithm works perfectly for the obvious reasons.

D. Compute the median of the values in Fragments F and store it at all the nodes.

Computing the median involves multiple implementations of the gossip algorithm that starts with calculation of Local Median and the Count of the number of nodes. These parameters are maintained for the computation, however other parameters like sum are utilized based on the implementations.

Brute Force

Before we get into the more complicated algorithms, we'd like to point out the obvious. Median is easily computable in a brute force way by disseminating the entire data throughout the system so that every node ends up with all the values in the system and every node can calculate the true global median.

Accuracy & latency-wise, probably no other algorithm can beat this, but efficiency & memory-requirement wise, this is probably the worst thing ever, least practicable. Which is why in the next section we talk about our attempts at implementing different algorithms that try to give a good approximation of a median value while maintaining some semblance of overall efficacy.

Note that however, we've attached a fully functional and partially optimized (like 4.2 for reducing communication latency) implementation for median calculation which converges within log N for our chord-like & fully connected topologies, and has been tested for well over a 1000 nodes over 2 VM's on our test machine.

A. Other Algorithms Implemented:

"compute_local_median(MyMedianComputation, NodesMedianComputation) ->"
First of our implementations "gossip_median.erl" involves calculating the Median (M) locally at every node as well as the sum of numbers (T) the node has seen and the count (C) of numbers it has seen. The fragment at each node is then rewritten as:

a. The median M repeated C/2+1 times.
b. The balance total (i.e. T-(M*(C/2+1))) split into C-(C/2+1) parts.

Rewriting in this manner preserves the total sum of numbers and count of numbers each node contributes a calculation.
When two nodes exchange their rewritten fragment the fragments are merged, sorted and a new values are calculated as follows:

a. Median M is the centre element
b. Tnew=T1+T2
c. Cnew=C1+C2

This update will always show the count of numbers and sum of numbers each node has seen till any particular instance in time.

"compute_local_median(MyMedianComputation, NodesMedianComputation) ->"
This implementation in "median.erl" follows the same approach as the above method but preserves only count of numbers seen and not the sum of numbers seen. Thus instead of calculating the balance total it simply rewrites the local fragment by repeating the locally computed median (M) C times, where C is the count of numbers considered to compute the locally computed median, M.

"compute_local_median(MyMedianComputation, NodesMedianComputation) ->"
This implementation in "medianminmax.erl" also preserves the total count of numbers (C) locally seen at every node as well as the min and max values seen thus far. It does not however attempt to preserve the sum of numbers seen so far. Thus when rewriting the fragment at any node the:

a. The min, max and median computed locally are considered once
b. The balance count is split into two parts (i.e. (C-3)/2).

i. One part is rewritten by repeating the left central value [(min+median)/2] in the rewritten fragment (C-3)/2 times
ii. The other part is rewritten by repeating the right central value [(median+max)/2] the same number number of times.

B. Results

In O(ln(N)) time we have failed to achieved guaranteed convergence on the median value. Although part of the network converges very well towards the actual median in the first and third method not all nodes do so. In cases for small network sizes good convergence and accuracy of median at all nodes is achieved for the first algorithm.

E. 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.

We have made certain assumptions for this case and we make our case as follows. We assume that every Node that hosts a Fragment also has the fragment's version number stored along with it. Now, the version number can indeed just be a version number or a logical or real timestamp, which is left up to the real implementation. For our sake, we simply assume that every replica of Fragment i have its version V associated with it. And that Node 1 is simply going to disseminate the new values of version V+1 throughout the system and we ensure that every replica is updated with the new version's data.

Algorithm:

a. Every node has a data set of values {Data_Values, Version, Fragment_Id}.
b. Node 1 wants to update Values of Fragment i with new data & a new version number
c. In every round of gossip exchange, {update, Frag_Id, New_Version,New_Values} is unilaterally passed on to the node that has been called unilaterally in a PUSH or to the node that called a PULL.
d. When a node gets this message, it continues to disseminate it for a certain epoch period (we call it KCount to verify if in a topology, dissemination is logarithmic or not).
e. If & when this message reaches a node that holds the data of that particular fragment, it will check the incoming version number. If newer, then it updates its own values & also begins disseminating this message for the rest of the epoch (very important, we'll come to this later), else it does nothing.

Results:
a. Like 4.1 & 4.2, dissemination is Logarithmic in our topologies of chord-like & complete and very poor in a ring or multi-ring.
b. We rely on the inherent reliable nature of TCP in the sense that we do not wait for a response from any node in the network. This reduces the overall message cost in our network significantly. So even if rounds of gossiping remain the same, the actual cost of gossip is reduced.
c. Because even a update Node continues disseminating the update, we are pretty much guaranteed with a probability close to 1 that ALL nodes hosting replicas of the fragment will receive the update. In one of our experiments, we had stopped disseminating update when one of replicas received the update, but that replica happened to be one of the edge nodes providing connectivity between 2 separate partitions in the graph, which caused the rest of the nodes in the 2nd partition never to get the update.

F. Retrieve the contents of fragment I from any node that may have an up-to-date copy of it. The retrieval is requested by node 1, with the user specifying the fragment number.

Algorithm:
The algorithm is based on read one/write all method. The fetch for fragment i is initiated by one node.

- The node starts disseminating a {fetch,true} message.
- When the initiating node receives the fragment it had requested, it starts disseminating a {fetch, false} telling nodes to stop fetching and exit.
- When a node which is assigned the fragment i receives the {fetch,true} request it replies with the actual data fragment in a {fetch, dataIsHere} message and starts disseminating the same.
- When a node receives a {fetch, dataIsHere} message it starts disseminating the same message along with the data fragment associated with the message.
- When a node receives the {fetch, false} message it disseminates this message.

Every node disseminates any kind of message ln([number_of_nodes_in_network]) times and gets into a pure receiving mode thereafter in attempt to receive a message indicating that it should change its mode. The only exception being, when a node is done disseminating the {fetch, false} message it simply exits. All leftover nodes who are victims of no receipts will automatically exit by timing out.

Results:
The algorithm manages to fetch the i^{th} fragment in logarithmic time, the time being a function of the network size. We have ensured that every node skips participating in a round of gossip every now and then by selecting itself as the recipient. This ensures that the infection spread a considerable amount before every node participates in a gossip round again. In a network where fragments are replicated (which is our assumption for simulating this algorithm) this works to our advantage and hence results in a very quick fetch response.

Conclusions
We conclude finally that the gossip algorithms were implemented successfully from the experiments performed above and it is evident that our experiments were scalable and robust and could be implemented for any size of networks. The Min and Max algorithm is very robust and almost gives the results instantaneously. From the experiments it is evident that for topologies like CHORD and fully connected graph, our averaging algorithm operates and converges perfectly. There were many experiments done on calculating the median that arrives at a closer value, but more insight would be required in this direction.

References

[1]Jelasity et al, Gossip-Based Aggregation in Large Dynamic Networks 2005
[2]Karp et al, Randomized rumor spreading 2000
[3]Demers et al, Epidemic algorithms for replicated database maintenance 1987
[5] http://en.wikipedia.org/wiki/Gossip_protocol
[6] www.erlang.com
[7]http://www.erlang.org/course/concurrentprogramming.html
[9] http://stackoverflow.com/
[10] http://learnyousomeerlang.com/ - Finest resource , Erlang for Dummies