Failure Model for Gossip and PushSum Algorithm

Description

Our main aim of developing this model is to facilitate the convergence to accommodate any failure during the gossip / pushsum execution.

We have exceptions pertaining to the failures listed below-

1. **Crash failure** : A node halts, but is working correctly until it halts. The exception associated with it is –
   ```
   case class nodeCrashFailureException(smth:String) extends Exception(smth)
   ```

2. **Omission failure** :
   - Receive omission: A node fails to receive incoming messages. The exception associated with it is –
   ```
   case class nodeRecieveOmissionFailureException(smth:String) extends Exception(smth)
   ```
   - Send omission: A node fails to send messages. The exception associated with it is –
   ```
   case class nodeCSendOmissionFailureException(smth:String) extends Exception(smth)
   ```

3. **Timing failure** : A node's response lies outside the specified time interval. The exception associated with it is –
   ```
   case class nodeTimeoutFailureException(smth:String) extends Exception(smth)
   ```

4. **Response Failure**: The node's response is incorrect/ Arbitrary

Handling the Failures

Master actors watch over their workers, just as in real life, taking care if they are in problem while executing the work. Each actor defines its own supervisor strategy, which tells Akka how to deal with certain types of errors occurring in your children. We have implemented our OneForOneStrategy instead of sticking to the default strategy.

There are four possibilities after a failure happens

- Resume the node
- Restart the node
- Stop the node
- Escalate the node

In our model Master stops the node as the value contained during pushsum will be lost once it fails.

Challenges Faced

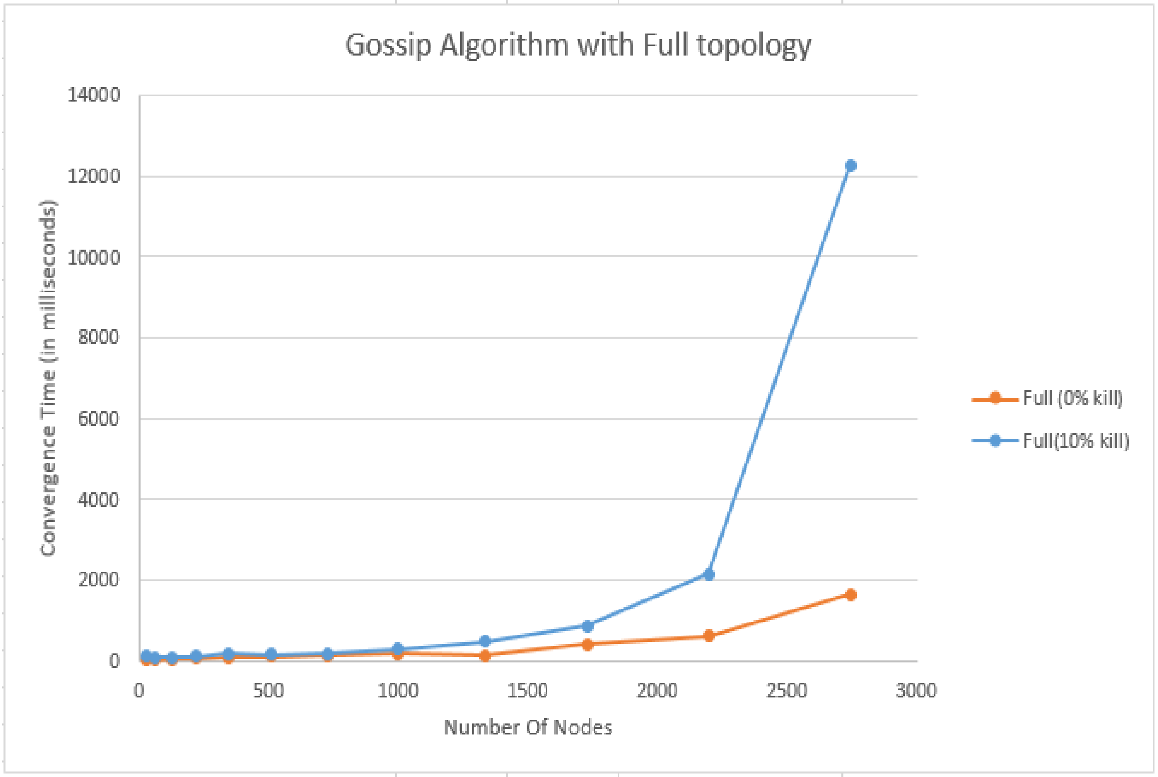
1. Multiple nodes can fail simultaneously. Handled it by,

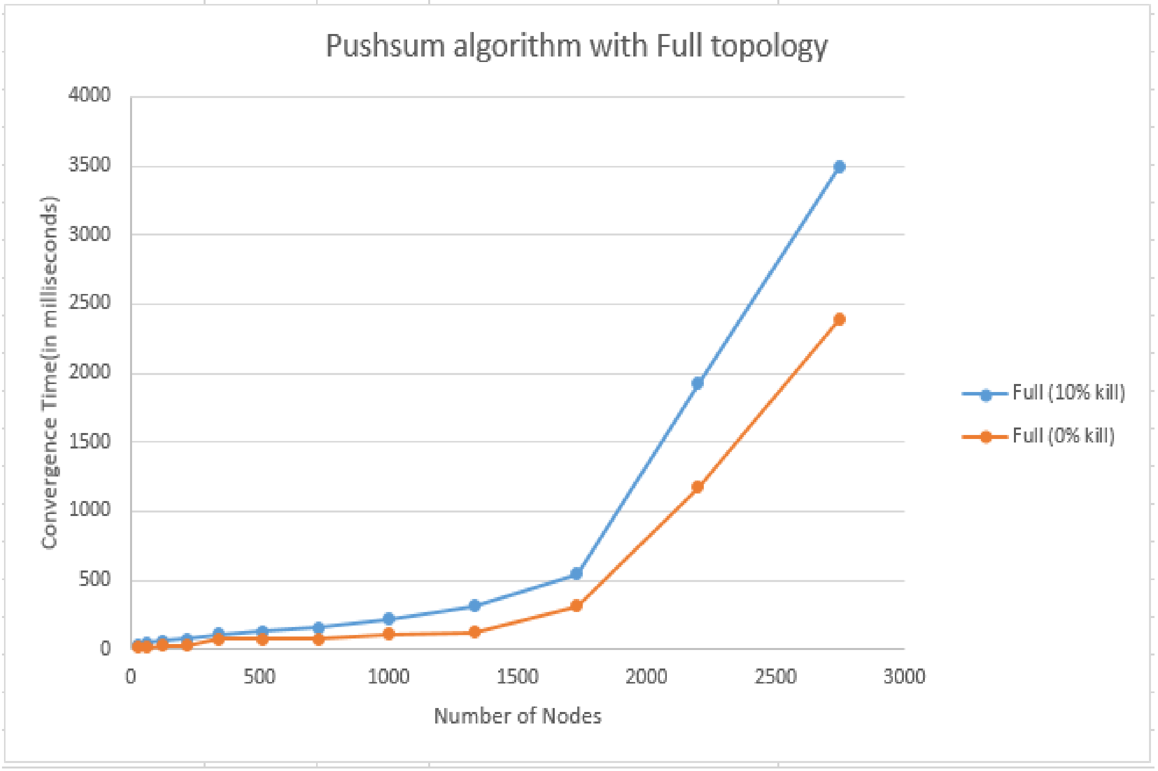
    - If the node to start is a failed node, we select the node again.
    - Whenever the random neighbor selected has failed, we reselect the node to be sent.
    - Optimized it by remembering what all nodes are no more into the system to avoid recomputations
    - State of each actor, i.e., whether it is alive or dead is maintained by the flag "isAlive". It is initialized to 1 whenever a node is created and upon failure of a node it is set to 0 and the node is killed.

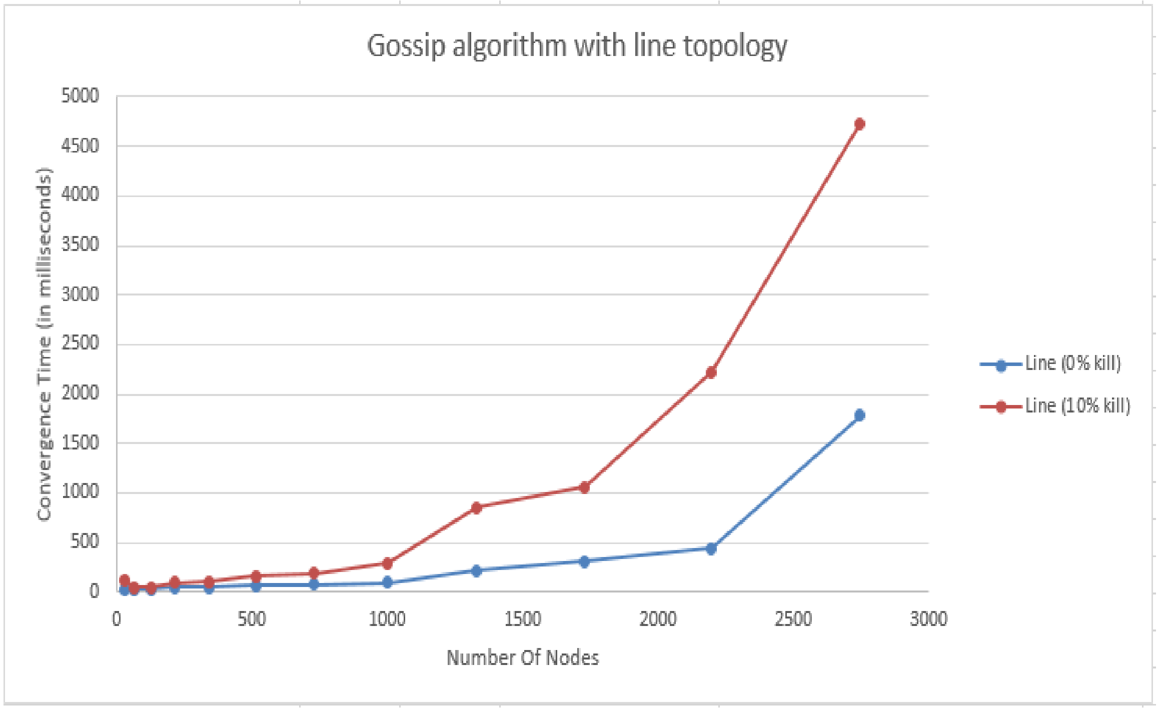
2. The random number generation of scala is not very random(needed in order output a set of distinct random nodes which fail.). Handled it by,
    - Written our own optimized random function using existing rand library.

Conclusion

1. This is a graph which shows the contrast between when 10% of the nodes in the system are killed and when 0% i.e. none of them are killed. We can see how the convergence varies, with the increase in time taken for the nodes to converge when the nodes are killed.


Gossip Algorithm with Full topology
14000
12000
10000
8000
6000
4000
2000
0
0
500
1000
1500
2000
2500
3000
Convergence Time (in milliseconds)
Number Of Nodes
Full (0% kill)
Full(10% kill)


Pushsum algorithm with Full topology
4000
3500
3000
2500
2000
1500
1000
500
0
0
500
1000
1500
2000
2500
3000
Convergence Time(in milliseconds)
Number of Nodes
Full (10% kill)
Full (0% kill)



2. Based on the observations, it was found that Imp3D and full topology works better as compared to the line topology. That is because as the number of failed nodes increase, time taken increases but the convergence is still achieved.

<u>Limitations of Model</u>

1. Convergence is definitely achieved but at the same time after a threshold of no of nodes, dead letters become unavoidable.

2. Based on the observations, we can conclude that this failure model implemented is very resilient.

<u>References</u>

1. http://doc.akka.io/docs/akka/snapshot/scala/fault-tolerance-sample.html#full-source-code-of-the-fault-tolerance-sample
2. http://danielwestheide.com/blog/2013/03/20/the-neophytes-guide-to-scala-part-15-dealing-with-failure-in-actor-systems.html