An Efficient Algorithm to Reconstruct a Minimum Spanning Tree in an Asynchronous Distributed Systems

Suman Kundu
Department of Information Technology
Jadavpur University
Salt Lake, Kolkata - 700098
sumankundu.nsec@gmail.com

Dr. Uttam Kr. Roy
Department of Information Technology
Jadavpur University
Salt Lake, Kolkata - 700098
u_roy@it.jusl.ac.in

***Abstract*—In a highly dynamic asynchronous distributed network, node failure (or recovery) and link failure (or recovery) triggers topological changes. In many cases, reconstructing the minimum spanning tree, after each such topological change, is very much required.**

In this paper, we have described a distributed algorithm based on message passing to reconstruct the minimum spanning tree after a link failure. The algorithm assumes that no further topological changes occur during the execution of the algorithm. The proposed algorithm requires significantly fewer numbers of messages to reconstruct the spanning tree in comparison to other existing algorithms.

I. Introduction

A distributed network consists of several nodes and connection among them. Each node is a computational unit, and the connections between them can send and receive messages in a duplex manner. Multiple paths may exist between a pair of nodes. A Minimum Spanning Tree (hereafter referred to as MST) of such a network is the minimally connected tree that contains all the nodes of the network. Applications of MST includes, effective communication in distributed systems, effective file searching and sharing for peer-to-peer network, gateway routing in local area network or bandwidth allocation in multi hop radio network and other computational scenario. Usually, a cost is associated with each link. The cost may indicate distance between two nodes, or the time required to send or receive data packets, or bandwidth of the communication channel, or any other parameters. A MST always contains the minimum cumulative cost within the network.

A distributed system is dynamic in nature i.e. in any distributed network topological changes occur with respect to time. A change can occur due to deletion or recovery of nodes and links. In many situations, it is important to reconstruct the MST after each topological change. The main hurdle to reconstruct MST arises due to the asynchronous nature of the system. Moreover, A node only knows local information. If topological changes occur, it must be propagated to each node via message communication. It is also possible that some part of the network gets the latest knowledge, whereas some portion does not. Algorithm should address this issue as well.

Several algorithms for constructing MST in distributed systems were proposed in last three decades. Most of these MST construction algorithms are applicable in a static topology. In this paper, we have proposed an algorithm based on message passing to reconstruct an MST that works seamlessly even in a dynamic topology. Our algorithm considers a single link failure and assumes that no further topological changes occur during the execution of the algorithm. It is also shown that the total number of messages required to reconstruct the spanning tree is significantly less in comparison to other existing algorithms.

The rest of the paper is organized as follows: Section II describes the related work and overview of our result. Section III describes description of distributed algorithm, analysis and proof of correctness. In section IV, we provide a result of simulation; section V concludes the overall algorithm and finally in section VI, we point out the further research areas we are working on.

II. Related Work

In their pioneer paper [2], Gallager, Humblet and Spira proposed one of the first distributed protocols to construct MST in year 1983. The protocol of [2] is further improved in the protocol [3], [4], [5], [6], [7] and [8]. In [9], some flaws of [5] are rectified. All these protocols of constructing MST are developed for static topology. Some of them address message efficiency and some of them address time efficiency, as the performance measures for the algorithm.

However, the distributed systems are dynamic as described in the previous section. Researchers are working on protocols, which are resilient in nature to adopt the topological changes. Few of such algorithms are given in [1], [12] and [14]. In their paper [10] B. Das and M. C. Loui provide a serial and a parallel algorithm to address the similar problem and later improved by Nardelli, Proietti and Widmayer in their paper [11]. These algorithms do not address the distributed version of the problem. In paper [13], P. Flocchini, T. M. Enriquez, L. Pagli, G. Prencipe, and N. Santoro provided a distributed version of the same problem. In [13] authors provided with

the precomputed node replacement scheme. In our paper we provide the improved version of the distributed algorithm of [1] for a single link failure. In the following section, we will describe the response of a link failure by [1].

A. Basic Algorithm of [1]

In the algorithm of [1], C. Chang, I. A. Cimett and S.P.R. Kumar proposed a resilient algorithm which reconstructs the MST after link failure and recovery. Complexity of the algorithm for a single link failure is $O(e)$, where e is the number of links in the network.

A link failure is a process of fragment expansion i.e. A link(which is a part of the MST) failure breaks the MST into two different fragments. The failed link initiates the process of recovery in the adjacent nodes. The initiator node generates new fragment identity. In algorithm [1], authors suggest two approaches to generate new fragment identity such that no conflict occurs between subsequent topological changes. First approach, is to include identity of all nodes of the fragment into the fragment identity. Second approach is to maintain a counter for each link. This counter counts the link failure and includes the value along with weight of the node for generating new fragment identity. In case of first approach, the size of the fragment identity of a large fragment becomes very large. So, the second approach is efficient in terms of message size.

If a link $w(u, v)$, is broken in certain time and u be the parent of v. In response to the link failure, u will generate the new identity for the fragment like $w(u, v), u, c$ where c is the counter of topological changes of the link $w(u, v)$. Then u will forward the fragment identity to the root of the fragment. However, in case of v, after generating the fragment identity like $w(u, v), v, c$, it marks itself as root of that fragment. After getting the new fragment identity, root of these two fragments changes their fragment identity and starts broadcasting *REIDEN<id>* using the tree links and wait for the acknowledgment. Any intermediate nodes, upon getting the *REIDEN<id>* message changes its fragment identity to the id and sends the same message to its child node. A leaf node, after getting the *REIDEN<id>* changes its fragment identity and sends *REIDEN_ACK<id>* to its parent. Intermediate node, sends *REIDEN_ACK<id>* to its parent only after getting *REIDEN_ACK<id>* from all of its children. Receiving of the *REIDEN_ACK<id>* message indicates that all nodes of the fragment aware about the new identity value. Root now changes its state to find and initiate the find minimum outgoing edge (hereafter referred to as *MOE*) phase by sending *FINDMOE* message over the children. When a node receives *FINDMOE* message, a node changes its state to find. In the find state each node starts to send *TEST<id>* message via each non tree link. A *TEST<id>* message is responded by either *ACCEPT<id>* or *REJECT<id>*. An *ACCEPT<id>* message, indicates that the edge is outgoing, leading to another fragment. An important thing to remember here is that the *ACCEPT* or *REJECT* message should return the identity number of the test message. This will help to determine whether the message is for the current failure or previous one. After identifying the *MOE* a node propagates the *FINDMOE_ACK<w(MOE)>* to upward. Where *w(MOE)* is the locally known best outgoing weight, either its own *MOE* or *MOE* received from its children (Whichever is minimum). After receiving *FINDMOE_ACK<w(MOE)>* root sends *CHANGE_ROOT<id>* through the same path it receives the *MOE*. The *CHANGE_ROOT<id>* reaches to the node where the *MOE* of the fragment incident. The node marks itself the new root of the fragment and sends *CONNECT* message over the *MOE*. Connect subroutine works same as the algorithm [2] and merge two fragments sharing the same *MOE* and starts the next iteration.

B. Overview of Our Results

When considering the single link failure, our approach provides a significant improvement on the total number of messages required during reconstruction over the algorithm of [1]. Also, the message size for some control message is improved slightly.

After link failure, the fragment which contains the root node of the MST is referred as *root fragment* in this paper. If the root fragment contains E' number of edges, then our algorithm requires $2 \times E'$ fewer messages to reconstruct the MST. Our approach here is to use the previously known fragment identity (say it as a historical data) for the root fragment. However, how the algorithm evolves if another link failure occurs during the execution is still under observation.

III. Algorithm to Reconstructing MST after Link failure

We closely followed the response of the algorithm [1] for a single link failure and found some improvement areas. In the following subsections, we will describe the network model, our observation regarding the algorithm [1], our contribution to improving the algorithm, description of the modified distributed algorithm, analysis of the outcome and proof of correctness.

A. Network Model

The communication model for the algorithm is modeled as an asynchronous network represented by an undirected weighted graph of N nodes. The graph is represented by $G(V, E)$, where V is the set of nodes and $E \subset V \times V$ is the set of links. Each node is a computing unit consisting of a processor, a local memory and also an input and output queue. The input (output) queue is an unlimited sized buffer to send and receive messages. A unique identification number is associated with each node (Node i represent the node with the identification number i).

Each link (u, v), assigned with a fixed weight $w(u, v)$ is a bidirectional communicational line between the node u and the node v. Each node has only the local information i.e. each node aware about its identification number and the weight of the incident links. After construction of the MST, each node will be aware of two additional information; first, the adjacent edge leading to the parent node in the MST and secondly the adjacent edges, those leading to the child nodes in the MST.

Nodes can communicate only via messages. The messages may be lost due to link failure during transmission. However, if the link is functioning, the messages can be sent from either end and can be received by the other end within a finite, undeterminable time, without error and in sequence.

Also, if a link failure occurs, then the failure event triggers the recovery process for each end of the failed link and the recovery process initiated by the same node.

B. Observation

In the algorithm of [1] the reconstruction process works in two phases. In *Phase-I*, root node informs each node of the fragment with the new fragment identity, and in *Phase-II*, each node finds its own *MOE* and forward the *MOE* to the root. Root then identifies the *MOE* of the fragment. Finally, the fragment sends *CONNECT* message via the *MOE*.

After failure each fragment changes its fragment identity to new one. Also, each message passes to the neighbor contains the fragment identity along with the control information. This is used because if overlapping of link failure occurs then the message response could be avoided depending upon the fragment identity such a way, that only the current failure will be processed during the execution.

C. Our Contribution

Our contribution to the algorithm is that we can use the historical data like previously known fragment identity for one fragment. When we use the previously known fragment identity than the fragment with the older identity enter its *Phase-II* without executing *Phase-I*. For our algorithm, we use the previously known fragment identity for the root fragment. Also, the *FAILURE* message propagating from the failure link to the root of the root fragment no longer requires to carry newly generated fragment identity. That means the *FAILURE* message size is also reduced. The difficulty with this approach is, when a *TEST<id>* message is received it may be possible that fragment identity of the node is not updated yet. It may be part of the same fragment or other fragments still in *Phase-I* (Propagating new fragment identity is not completed yet). However, if a node receives a *TEST* message in *Phase-II*, then its fragment identity is correct. So, to avoid the conflicting response to a *TEST<id>* message, the response is delayed until the node enters into *Phase-II*.

Also, it is assumed that no further failures occur during the execution. So, it is possible to reduce message size for control messages. For example, the *ACCEPT* and *REJECT* message do not require to send the fragment identity back to the sender.

D. Description of the protocol

In the beginning, each node maintains a collection of adjacent edge. This adjacent edge collection is sorted by the cost of the link. During the life time, an adjacent link can have one of the following status

1) *Basic* - the link is yet to processed
2) *Parent* - the link lead to the parent
3) *Child* - the link lead to child

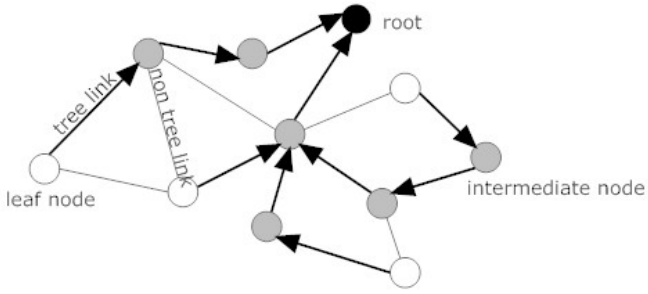

Fig. 1. MST of a random network

4) *Rejected* - the link is leading to the node included in the same fragment
5) *Down* - the link in not working

It is assumed that, the MST is already constructed using some distributed protocol. A link failure, triggers the recovery process in both end of the link. Let us take, failure occurs for the link $e = (u, v, w, c)$ where u and v is the node connecting the link. w is the weight of the link and c is the status change count for the link. If the link is not included in the MST, i.e. it is either in *Basic* or in *Rejected* state then u and v simply changes the link status to *Down* and do nothing.

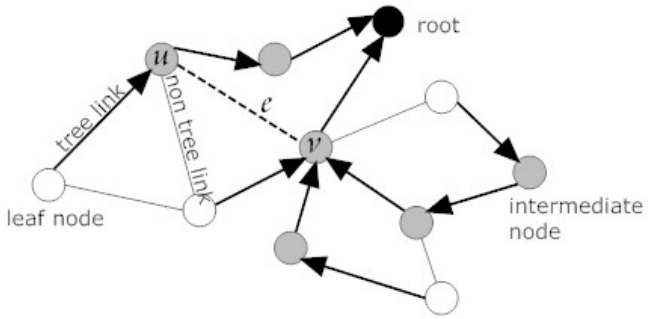

Fig. 2. A non MST link failure

Otherwise, the nodes marks the link *Down* and respond in the following manner -

- When previous status of the link is *Parent* - the link marks itself as root of the newly generated fragment. It then marks all *Rejected* nodes to *Basic*. This is necessary because the link may lead to the other fragments due to the topological change. The node generates new fragment identity for the fragment, reset its own fragment identity and enters into the *Phase-I* by sending the *INIT<fid>* message to its children. Here *fid* is the new fragment identity as described by the algorithm [1], i.e. it includes the weight and count of failure along with the identity of the node. If u is the parent of v in the example edge e then after failure v marks itself the root and changes its fragment identity to $fid = w(u, v), v, c$ then initiate *Phase-I* by sending this *fid* along with the *INIT* message. After receiving the *INIT<fid>* message node changes its

fragment identity to *fid* and marks all *Rejected* link to *Basic*. Then it forwards the message to its children. If the node is leaf node then it returns a *FINISH* message to its parent. Each intermediate node waits for receiving *FINISH* message from all its children and then sends the *FINISH* message to the parent. A *FINISH* message received by v (i.e. the root) indicates that all node of the fragment knows the current fragment identity. Then v starts *Phase-II* by sending the *FINDMOE* message.

- When previous status of the link is *Child* - the link forward the *FAILURE* message to upward. Note that the node did not generate new fragment identity to forward along with *FAILURE* message. When the root node detects the failure, it initiates the *Phase-II* directly by sending *FINDMOE* message to its children.

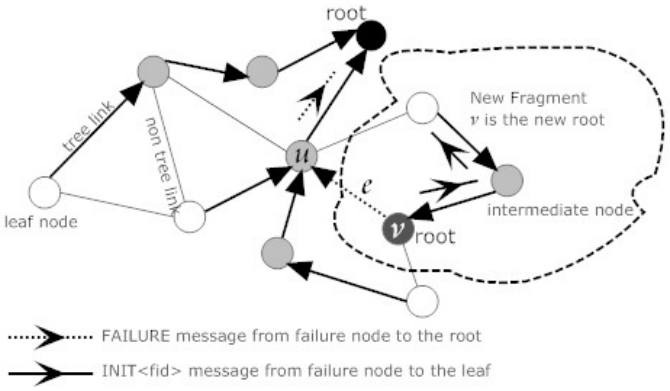

Fig. 3. MST link failure and response of u and v

- A node receiving *FINDMOE* message, immediately enters to finding state. In finding state each node finds its local *MOE*. To find the local *MOE* a node picks the minimum weighted adjacent edge which is in *Basic* state and send *TEST*<*fid*>.

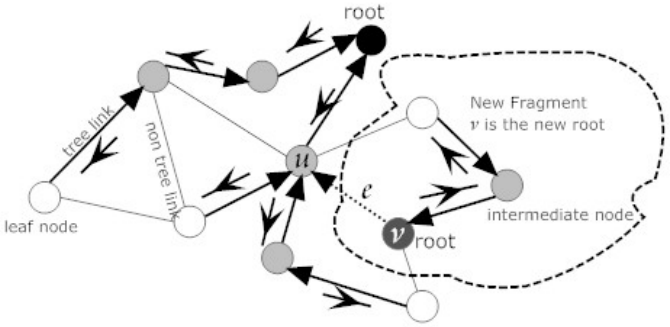

Fig. 4. *Phase-II* initiated by root of the fragment with *FINDMOE* message

- A node received a *TEST*<*fid*> message. The following two cases to consider -
  - Node executing in *Phase-I*: Then the response is delayed until the node itself enters into *Phase-II* by receiving *FINDMOE* from its parent.

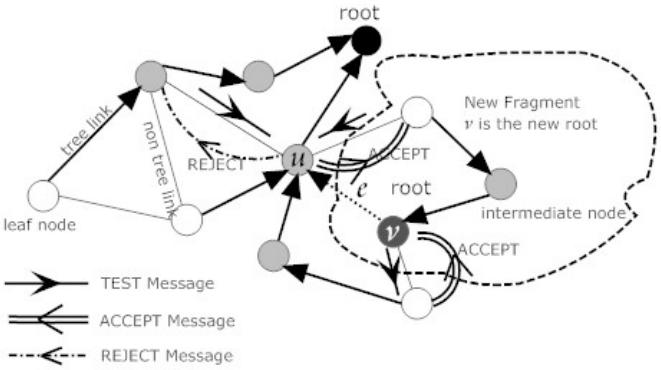

Fig. 5. TEST message and response

  - Node executing in *Phase-II*: In this scenario a *TEST*<*fid*> message is replied by either *ACCEPT* if its fragment identity is different than of *fid* or by *REJECT* if its fragment identity is same as *fid*.

- Upon receiving the *REJECT* message the node picks the next best edge in *Basic* state and sends *TEST*<*fid*> to test the edge. However, if it gets *ACCEPT* message, which indicates it found its local *MOE*, then the node waits for its children's response. After finding the local *MOE*, leaf nodes propagate the best weight to its parent via *REPORT*<*wt*> message.
- When a *REPORT*<*wt*> message is received, the intermediate nodes compare their local *MOE* with the wt received from children and change the *MOE* accordingly. After getting *REPORT*<*wt*> message from all of its children it sends the *REPORT*<*wt*> to its parent with the best weight known by the node. Thus the best weight is propagated to the root node of the fragment. At this point the root sends the *CHANGE_ROOT* message to the same path that leads to the *MOE*.
- A node with the *MOE* of the fragment receives *CHANGE_ROOT* and marks itself as a root of the fragment. Then it sends the *CONNECT* message over the *MOE* and merge with the fragment sharing same *MOE*.

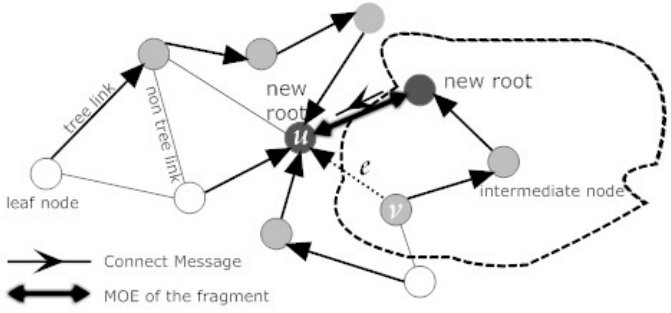

Fig. 6. Root of the fragment is changed and CONNECT message send for merge the fragment

E. Analysis

Compare with the algorithm [1], in our approach the root fragment directly enters into *Phase-II*. So, the *INIT<fid>* and *FINISH* messages (*REIDEN<id>* and *REIDEN_ACK<id>* in algorithm [1]) of phase one is not required for root fragment. Let us take the root fragment has E' number of edges after failure. Then to executing *Phase-I* it required to send E' number of *INIT<fid>* message over E' links. Also, it required to send E' number of *FINISH* message over E' links. That means the reconstruction process of our algorithm require $E' + E' = 2E'$ fewer messages then the protocol described in [1].

When compare our approach with the protocol of [1] in terms of message size, some control message contains very few bits with respect to the protocol of [1]. For example, the *FAILURE* message in the root fragment only contains the control information indicating the failure occurrence. No fragment identity is sends along with it. As we consider no further failure occur during execution of the protocol *ACCEPT* and *REJECT* message also contains control message only; no fragment identity returns with it.

1) Complexity: Let us consider the network contains N nodes and E edges. The initial MST contains N nodes and e edges before the failure. Also, consider the root fragment contains N' number of nodes and E' number of edges. If the height of the root fragment be h', then to propagate the failure message to the root of the root fragment require $O(h')$ number of message. Propagate the new fragment identity for the other fragment, requires $O(N - N')$ messages because this information will be propagated through the tree links. Similarly, to send Find *MOE* request and merging two fragments require $O(N)$ messages since these messages also be sent through the tree links. However, finding *MOE* of the fragments require to send messages through $O(E)$ links of the network. So, the message complexity for the algorithm on link failure is $O(E)$.

F. Proof of Correctness

At first, we will proof several Lemmas, which are used in the distributed algorithm, and then we will show that the algorithm generates the minimum weighted spanning tree after completion of the algorithm.

Lemma III-F.1. *Before topological changes, each node of the network has information about the tree links incident to it.*

Proof: It is assumed that the MST is initially constructed using some distributed protocol. In our simulation, we use the algorithm [7] to construct MST. Each node maintains a list of incident edges and their status as described in section 3.4 i.e. nodes are aware about the links leading to its parent and links leading to its children. These parent and child links denote the tree links which are included in MST. Hence, each node is aware about the tree links, incident to it before any topological changes. ■

Lemma III-F.2. *Root of the fragments receives failure notification within a finite time.*

Proof: A failed link broke down the MST into two fragments and notifies the failure to either end of the link. The node attached with the root fragment propagate the *FAILURE* message toward the root element. The other node mark itself root of the new fragment and generates the new fragment identity. Each message is assumed to be reach at the destination in finite time and in a sequence manner whenever the link is working. Also, it is assumed that no further failure occurs during the execution. Hence, the *FAILURE* message is correctly received by the root of the both fragments in finite time. ■

Lemma III-F.3. *On each node, fragment identity is updated before the start of Phase-II according to the latest link failure.*

Proof: Root fragment uses the previously known fragment identity of existing MST. So the root fragment does not require to update the fragment identity. It then directly enters into the *Phase-II*, all nodes belongs to the root fragment already aware about the fragment identity. However, for the other fragment, newly generated identity is propagated to the child nodes by *INIT<fid>* message. Leaf node returns the *FINISH* message to its parent after updating their fragment identity. Only after receiving *FINISH* message from its entire child a intermediate node sends *FINISH* message to its parent. As the messages are assumed to be reach at destination sequential manner in a finite time then *FINISH* message received by the root of the fragment implies each node already updated its fragment identity. Then the root node initiates *Phase-II* for that fragment. Thus, when a node is in *Phase-II*, its fragment identity is always updated with the latest link failure. ■

Lemma III-F.4. *Each fragment starts its Phase-II execution in finite time.*

Proof: As their is no link failure occur during the execution of the protocol, all messages of *Phase-I* will be properly responded by the nodes in a finite time and finally terminated when *FINISH* message received by root of a fragments. Root node then initiate the *Phase-II* by sending *FINDMOE* message. Hence, each fragment starts its *Phase-II* in a finite time. ■

Lemma III-F.5. *Fragments find its MOE within a finite time.*

Proof: After getting *FINDMOE* message each node sends *TEST* message to the minimum weighted non tree link (Which is not in *Down* status) to test whether the link is leading to another fragment or not. Then the node waits for the response from the other end. The *TEST* message is correctly reached to the other end because the link is not marked as down and message lost is not a valid property according to the assumption. *TEST* message response is delayed until the node start executing *Phase-II*. From the previous lemma, we found that *TEST* message is responded within a finite time because the responder node will enter *Phase-II* within a finite time. Also, the response is known to be correct because in *Phase-II* the fragment identity is already updated with the new fragment identity. Now, if E_l be the local minimum outgoing

edge and E_c be the minimum outgoing edge forwarded by all of its children. Then, an intermediate node forward the $MOE = min(E_l, E_c)$ to its parents. Thus, at the root node MOE of the fragment is calculated by $MOE = min(E_l, E_c)$ within a finite time. ■

Theorem III-F.1. *The algorithm reconstructs spanning tree in finite time and the spanning tree is the minimum weight spanning tree of the network.*

Proof: When the minimum weighted outgoing edge *MOE* is determined by root, the fragment changes its root to the new node where the *MOE* is incident. Then it sends the connect message through *MOE*. If the *MOE* found by one fragment is also *MOE* of the other fragment then two fragments merge and create a merged fragment. If there is no other fragment remains then it is the desire spanning tree. As merging only possible if both fragments agreed that the *MOE* is common, it is obvious that algorithm does not produce any cyclic path.

Considering the case, where only one link failure occurs we can easily derive the proposition that the *MOE* of one fragment is also the *MOE* of other fragment(Since network links are uniquely weighted). Hence, the fragments merge at *MOE* and produce a spanning tree.

Now, there is no possibility of any other outgoing edge with lesser weight then *MOE*, because in the process only minimum weight edge is filtered and forwarded from the leaf node to the root node($MOE = min(E_l, E_c)$ from last lemma). Finally, root node determines *MOE* of the fragment. So, the merging occurs at the minimum possible weighted edge of two fragments. Hence, merged spanning tree has the minimum collective weight in the system.

Thus, upon terminating the algorithm reconstruct the spanning tree which has the minimum weight i.e. *MST* of the network. ■

IV. Experiment and Result

To compare the output, we simulate few network graphs using 'Network Simulator v2 (NS2 2.29)' [15]. We constructed the initial *MST* using the algorithm [7]. The pictures below shows the results of one experiment.

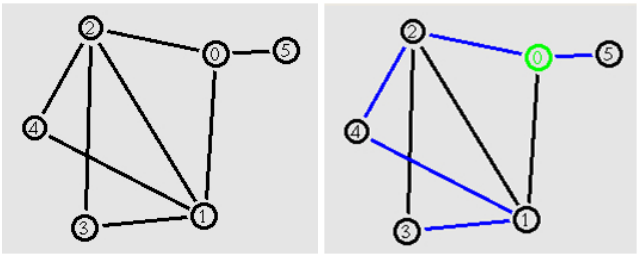

Fig. 7. Initial Graph and Constructed MST

The example graph contains the vertex set V, edge set E and corresponding weight set W as below -

Vertex:

$$V = \{0, 1, 2, 3, 4, 5\}$$

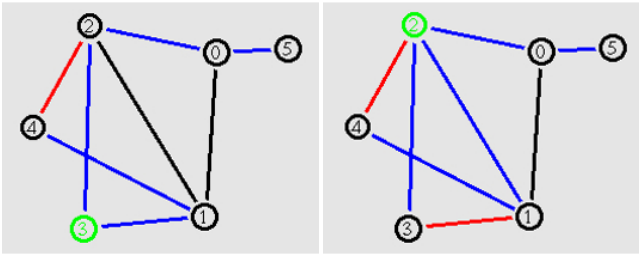

Fig. 8. Recovered MST after failure(in red)

Edges:

$$E = \{e1 = (0,5), e2 = (0,2), e3 = (0,1), e4 = (1,2), e5 = (1,4), e6 = (1,3), e7 = (2,4), e8 = (2,3)\}$$

Weight:

$$W = \{w(e1) = 4, w(e2) = 8, w(e3) = 15, w(e4) = 10, w(e5) = 3, w(e6) = 5, w(e7) = 6, w(e8) = 7\}$$

Message required to terminate the execution of the algorithm is tabularized in *Table* 1. *Scenario* 1, $e7$ link fails and initiates the recovery process. If we use the construction algorithm again (i.e. algorithm [7]) after failure it requires 114 messages, where as the performance is improved if we use the reconstruction algorithm (i.e. using the algorithm [1]). In our modified algorithm, it takes fewer messages then existing reconstruction algorithm of [1].

TABLE I
Link Failure vs Require Message Chart

Broken Link	Message Require For: MST Construction	Algorithm of [1]	Modified Algorithm
Scenario 1: Single link failure			
$e7$	114	59	55
Scenario 2: Single link failure			
$e6$	114	54	46
Scenario 3: Link failure one after another			
$e7$	114	59	55
$e6$		50	50

V. Conclusion

In this paper, we have presented a distributed algorithm to reconstructing a Minimum Spanning Tree after deletion of a link. The problem can also be solved using the protocol [1]. Here we showed that for single link deletion scenario our protocol can reconstruct the MST with $2E'$ fewer messages than protocol [1]; where E' represents the number of links in the root fragment. If we consider MST with large depth and the link failure occur to very close to leaf node (i.e. the E' is much greater than $E - E'$) then our algorithm performs much better

way. However, when E' is zero i.e. the link failure occurs at root node, the algorithm completed without any improvement of the total number of messages. We can refer to *Scenario* 3, where for next link failure there is no improvement in total number of messages.

VI. Further Works

When a *TEST*<*fid*> message is delayed until the node enters its finding state then we can use the same *TEST*<*fid*> to determine whether the adjacent edge is Rejected or Accepted and used it instead of sending another *TEST*<*fid*> message over the same edge. However, this approach may lead to some other difficulties due to the asynchronous nature, and we are currently working on the same.

Whenever the failure occurs at the root or very close to the root, improvement is close to zero. We are working on the algorithm so that it can use historical data such a way, which produces improvement for other scenarios too.

Also, currently we are working how we can modify the algorithm so that it accepts topological changes during execution of the algorithm.

References

[1] C. Cheng, I.A. Cimett, and S. P.R. Kumar, *"A protocol to maintain a minimum spanning tree in a dynamic topology.*" Computer Communications Review 18, no. 4 (Aug. 1988): 330-338

[2] R. Gallager, P. Humblet and P. Spira, "*A distributed algorithm for minimum-weight spanning trees.*" ACM Transaction on Programming Languages and Systems, 5(1):66-77, January 1983

[3] Chin F., Ting H. "*An almost linear time and O(n log n+e) messages distributed algorithm for minimum-weight spanning trees.*" Proceedings of 26th IEEE Symp. Foundations of Computer Science, p.257-266, 1985

[4] Gafni E., "*Improvement in the time complexities of two message optimal protocols.*" Proceedings of the ACM Symp. on Principles of Distributed Computing, 1985

[5] B. Awerbuch, "*Optimal Distributed Algorithm for Minimum Weight Spanning Tree, Counting, Leader Election, and Related Problems,*" Symp. Theory of Comp., pp. 230-240, May 1987.

[6] J. Garay, S. Kutten and D. Peleg, "*A Sub-Linear Time Distributed Algorithm for Minimum-Weight Spanning Trees.*" 34th IEEE Symp. on Foundations of Computer Science, pp. 659-668, November 1993.

[7] Gurdip Singh , Arthur J. Bernstein, "*A highly asynchronous minimum spanning tree protocol.*" Distributed Computing, v.8 n.3, p.151-161, March 1995

[8] Elkin M., "*A faster distributed protocol for constructing minimum spanning tree.*" Proceedings of the ACM-SIAM Symp. on Discrete Algorithms, p.352-361, 2004

[9] Michalis Faloutsos, Mart Molle, "*Optimal Distributed Algorithm for Minimum Spanning Trees Revisited*" Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 231-237, 1995.

[10] B. Das and M.C. Loui, "*Reconstructing a minimum spanning tree after deletion of any node.*" Algorithmica, 31. pp. 530-547, 2001.

[11] E. Nardelli, G. Proietti, and P. Widmayer "*Nearly linear time minimum spanning tree maintenance for transient node failures*" Algoritmica, 40. pp. 119-132, 2004

[12] Hichem Megharbi and Hamamache Kheddouci "*Distributed algorithms for Constructing and Maintaining a Spanning Tree in a Mobile Ad hoc Network*" First International Workshop on Managing Context Information in Mobile and Pervasive Environments, 2005.

[13] P. Flocchini, L. Pagli, G. Prencipe, and N. Santoro "*Distributed computation of all node replacements of a minimum spanning tree*" Euro-Par, volume 4641 of LNCS, pages 598-607. Springer, 2007

[14] Awerbuch, B., Cidon I., and Kuten, "*Optimal maintenance of a spanning tree.*" J. J. ACM 55, 4, Article 18 (September 2008), 45 pages.

[15] Network Simulator version $2(NS2)$, URL: http://www.isi.edu/nsnam/ns/