

May 1987

LIDS-P-1700

DISTRIBUTED NETWORK PROTOCOLS FOR CHANGING TOPOLOGIES: A COUNTEREXAMPLE

Stuart R. Soloway
Pierre A. Humblet

ABSTRACT

A number of distributed network protocols for reliable data transmission, connectivity test, shortest path and topology broadcast have been proposed with claims that they operate correctly in the face of changing topology, without need for unbounded numbers to identify different runs of the algorithms. This paper shows that they do not possess all the claimed properties and that they may not terminate.

Stuart Soloway performed this research while he was with Codex Corporation; he is now with Digital Equipment Corporation. Pierre Humblet is with the Laboratory for Information and Decision Systems, Rm 35-203, Massachusetts Institute of Technology, Cambridge, MA 02139. His work on this research was supported in part by Codex Corporation and in part by the National Science Foundation under contract ECS 8310698.

1 INTRODUCTION

A remarkable protocol has been introduced [Fin79] to guarantee reliable end to end data transmission in a network in the presence of arbitrary link and intermediate node failures while not requiring unbounded numbers to identify messages; it also provided a network connectivity test. The basic idea has also been used in [Seg83] to construct other protocols for connectivity test, shortest path and path updating with similar properties. These works relied on techniques set forth in [Gal76].

This article shows that although they contain valuable ideas the previous papers share a basic flaw and that the algorithms do not always operate correctly. This will be demonstrated in the case of [Fin79] in the following section. It is possible to modify some of the algorithms to insure the bounded sequence number property, but unfortunately at an increase in running time and communication cost compared to the previous (incorrect) versions. Such a modified algorithm appears in a companion paper [Hum87].

Before proceeding with Finn's algorithm we outline our model. We have a finite network of unreliable links and nodes. Nodes have distinct identities; to simplify the notation we assume that there is at most one link between two nodes, so that a link can be identified by the identities of its end points. Nodes execute distributed algorithms consisting of exchanging messages over links, receiving an external "GO signal" and processing. Message passing is the only way for the nodes to communicate. They have no access to a shared memory or to a global clock.

Regarding the transmission of messages over unreliable links, we assume the existence of a link protocol that interfaces with the processes that execute the algorithms and that has the properties similar to HDLC, i.e. after a link goes UP it transmits messages correctly and in sequence, until it eventually goes down. Links may not go Up (or Down) simultaneously at both ends. A more formal definition of the links behavior appears in [Hum87].

Similarly nodes can be Up or Down. A node operates without errors while it is Up but loses all its memory when going Down. When a node goes Down, all its links go Down within a finite time, and they cannot go back Up while the node is Down. Initially all nodes are Down.

2 FINN'S ALGORITHM

This section outlines the basic mechanism of Finn's algorithm and shows the problem that can appear in presence of link or node failures. Our goal in sketching the algorithm is to allow interested readers to go back to [Fin79] or [Seg83] and check that some algorithms there do not work, and to provide other readers with some intuition about the nature of the problem.

We view the algorithm as only providing for a connectivity test, i.e. when it halts at a node after a finite number of link or node failures, the node is aware of what other nodes are in the same connected network component.

Each node I maintains a vector $D(I)$ with an entry $D(I)(J)$ for each node J in the network. $D(I)(J)$ can take the values 0, 1 and 2 with the following meanings. A value of 0 indicates that it is not known at node

I if J is in the same connected component. The value 1 indicates that J is in the same component, but that the identities of all its connected neighbors might not be known as no message has been received from all of them. The value 2 indicates not only that J is in the same component, but also that a message has been received from all its connected neighbors (thus $D(J)(K)$ is 1 or 2 for all connected neighbors K of J).

Initially $D(I)$ is set to all 0, except $D(I)(I)$ which is set to 1 (at all nodes I). Nodes exchange their identities and $D(.)$ vectors with their neighbors; when a vector $D(K)$ is received at node I , $D(I)(J)$ is set to $\text{MAX}(D(I)(J), D(K)(J))$ for all J and if $D(I)(L)$ is equal to 1 or 2 for all neighbors L of I then $D(I)(I)$ is set to 2. If this update causes any change in $D(I)$ the updated value of $D(I)$ is communicated to all neighbors of I , where similar updates take place.

It is easy to see that in case of a "cold start" in absence of topological change the algorithm will terminate a node I with the entries of $D(I)$ set to 0 or 2, the later values corresponding to nodes in the same component as I .

The case of changing topology can be handled quite naturally by restarting the algorithm every time a topological change is noticed. To distinguish algorithm cycles it is enough to use restart numbers, keeping in memory the largest number already seen, and choosing a larger number at each restart.

By including the restart number in each message one can insure that all nodes in a connected component participate in the latest restart, discarding messages from previous ones (this method is different from that used in the ARPANET [McQ77] where there is a separate sequence

number for each node; it works correctly even if many nodes independently originate new restarts with identical sequence numbers).

The problem with this approach is that restart numbers increase monotonically. To avoid this difficulty [Fin79] has suggested that a node transmit only the difference between its current restart number and the previous one, and that each node maintain "link counters" to track the differences between the numbers of the restarts taking place at its neighbors. Transmitting and tracking differences solves the problem of monotonic increasing sequence numbers, but poses a problem when a link comes Up: with respect to what should the difference be interpreted? To solve this last problem [Fin79] delayed the processing of a link coming Up until both ends have terminated the algorithm and all "link counters" are zero; the "link counter" of a link coming Up in these conditions is initialized to zero. (We refer the reader to [Fin79] for the details).

To see that delaying links coming Up does not suffice, consider the following example where there are 4 nodes.

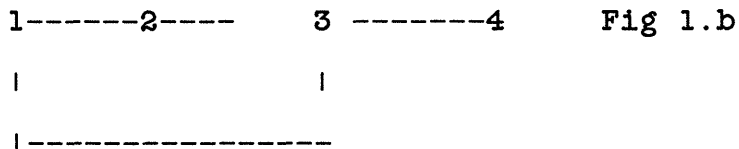
1 -----2-----3 4 Fig 1.a

Initially (Fig. 1.a) links (1,2) and (2,3) are Up, no node has started the algorithm and all link counters are 0. Node 3 starts its first restart and transmits $D(3) = (0,0,1,0)$ to 2. In answer node 2 transmits $D(2) = (0,1,1,0)$ to 1 and 3. Node 3 replies by sending $(0,1,2,0)$, that message arrives at 2 and it is forwarded to 1. While it is in transit, the three following sets of events take place:

a) The link between 2 and 3 fails, but it takes at very long time for the failure to be noticed at 2.

b) During that time node 3 terminates the algorithm then connects with node 4 and both run the algorithm until completion.

c) When this is done the link between 1 and 3 comes Up (it can, as node 1 has not yet joined any restart). The network is then as in Fig. 1.b.

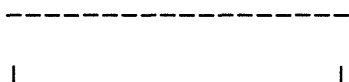


Both 1 and 3 start the algorithm by sending (1,0,0,0) and (0,0,1,0) to their respective neighbors 2,3 and 1,4; assume that the message from 3 to 4 suffers a long delay.

Now node 1 receives (0,1,1,0) from 2 and (0,0,1,0) from node 3. It sends (2,1,1,0) to its neighbors 2 and 3. After receiving this message node 2 has a vector (2,2,2,0), it sends it to 1 and terminates the algorithm, even though it does not know about 4! (in fact node 4 has not even started the algorithm in its current network component).

At this point the algorithm has halted at 2 without fulfilling its promise, but one might hope that this is not disastrous: node 2 will eventually receive notification that its link to 3 has failed and will restart and, in absence of topological changes, correctly terminate. However another event with catastrophic consequences can also occur.

It is now acceptable for link (4,2) to come Up, as none of its extremities are involved in the algorithm. The situation is then as in Fig. 1.c.



1-----2----- 3-----4 Fig. 1.c

Nodes 2 and 4 restart (the second time for 2, but only the first for 4 in the current network component) indicating a restart number increment of 1. The restart from 2 will be interpreted by 1 as being the SECOND one; node 1 will immediately also restart, answer to 2 and notify 3. The restart from 4 will be interpreted by 3 as being the FIRST one, and when notice of a second restart arrives from 1 node 3 dutifully relays it to 4, where it will arrive after the first restart from 1, thus triggering a message to 2 that a new restart is to take place. Node 2 then notifies 1 that a new restart (the THIRD one !) is to occur and the reader realizes that the algorithm is now chasing its tail, never terminating. That node 2 is eventually notified that its link to 3 has failed does not help.

A similar counterexample can be constructed for the algorithm EMH-Version B in [Seg83]. The proof of theorem EMH-B-1 has a flaw in the second column of page 32.

The counter example just outlined requires a peculiar timing of message arrivals and links going Up and Down. Those are issues that arise in many networks. To avoid timing problems of a similar nature, [Per83] reports that in the Arpanet nodes are delayed coming back Up following a failure. Although not elegant, this technique can be quite effective. However as networks grow in size and complexity the delay would have to grow. It is thus interesting to develop protocols that work correctly under all timing conditions. This is done in [Hum87], [Awe87] and

[Gaf87].

REFERENCES

- B. Awerbuch, "Fail-Safe Compilation of Protocols on Dynamic Communication Networks", MIT-LCS, 1987
- S.G. Finn, "Resynch Procedures and a Fail-Safe Network Protocol", IEEE Trans. Commun., vol. COM-27, pp. 840-845, June 1979.
- E. Gafni and , "", 1987.
- R.G. Gallager, "A Shortest Path Algorithm With Automatic Resync", unpublished note, March 1976.
- P.A. Humblet and S. Soloway, "A Fail-Safe Layer for Distributed Network Algorithms and Changing Topologies", this issue.
- J.M. McQuillan and D.C. Walden, "The ARPANET design decisions", Comput. Networks, vol 1, Aug. 1977.
- R. Perlman, "Fault-Tolerant Broadcast of Routing Information", Proc. IEEE Infocom '83, San Diego, 1983.
- A. Segall, "Distributed Network Protocols", IEEE Trans. on Info. Theory, Vol. IT-29, no. 1, Jan. 1983.