Applying Static Network Protocols to Dynamic Networks

Yehuda Afek* Baruch Awerbuch† Eli Gafni‡

Abstract

This paper addresses the problem of how to adapt an algorithm designed for fixed topology networks to produce the intended results, when run in a network whose topology changes dynamically, in spite of encountering topological changes during its execution. We present a simple and unified procedure, called *a reset procedure*, which, when combined with the static algorithm, achieves this adaptation. The communication and time complexities of the reset procedure, per topological change, are independent of the number of topological changes and are linearly bounded by the size of the subset of the network which participates in the algorithm.

1 Introduction

Many distributed algorithms for data communication networks have been developed in recent years [AG85, GHS83, Awe85]. Most of these algorithms are *static*; that is, they operate correctly under the assumption that the network topology remains fixed during the execution of the algorithm. Should the network undergo a topological change in the midst of execution, the results may be meaningless, even if the algorithm terminates. Yet, real networks, such as the ARPANET, do undergo topological changes due to link failures and recoveries.

How do centralized systems cope with failures? A standard procedure is to reset to an initial or checkpoint state and then restart. We would like to employ the same idea in a distributed environment. This raises two problems. A distributed system consists of a number of asynchronous autonomous processors which communicate over links with arbitrary delays. Resetting such a network requires a nontrivial protocol that can handle messages in transit. A compounding problem is that additional failures and recoveries can interfere with the resetting algorithm itself.

In this paper we solve these problems under two reasonable assumptions. Firstly, observe that if all links are always down, or if topological changes are so frequent that no messages are ever transmitted, then no meaningful distributed computation can be done at all. Therefore, we assume that the sequence of topological changes is finite (*eventual-quiescence*). The time after the last topological change models a long stretch of time in which no topological change occurs. Secondly, observe that if failures are undetectable or their detection may be delayed indefinitely, then it is impossible to reset the network [FLP85]. Thus, we make the assumption that in finite time after the last topological change every node is aware of the state of its incident links (*eventual detectability*). Networks satisfying these two assumptions are said to be *dynamic*.

We adapt static algorithms to a dynamic network environment in the following sense. If following the finite sequence of topological changes the final topology is G, then the algorithm terminates, producing a result (states of nodes when quiescence is reached) which is consistent with topology G. Any algorithm which operates correctly in this sense is called *fail-safe*.

The major contribution of this paper is a compiler that transforms static algorithms to fail-safe ones. The heart of the compiler is an efficient *distributed fail-safe reset procedure*, which can be viewed as a fail-safe distributed snapshot algorithm [CL85]. Given the reset procedure we proceed to show how it can be used to transform any static distributed algorithm to a fail-safe one. The transformation is performed as suggested in [Gal76, Fin79, CR84]. The execution of a static algorithm is reset whenever a topological change is detected. A new execution is then started in the hope that the network will be stable for a sufficiently long period of time to allow the new execution

*AT&T Bell Labs., 600 Mountain Ave., Murray Hill, NJ 07974.

†Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171 and NSF contract CCR8611442.

‡Computer Science Department, University of California, Los Angeles, Los Angeles CA 90024. Supported by NSF Presidential Young Investigators Award under grant DCR84-51396 & matching fund from IBM Faculty Development Award under grant D840622.

to run to completion without encountering new topological changes.

Unlike the suggestion in [Gal76, Fin79, CR84], we reset only the subnetwork which was recently traversed by messages of the algorithm. If a topological change occurs in a part of the network in which the algorithm was not running, then no action needs to be taken by the reset procedure.

Given a static algorithm π, the message complexity of our reset procedure, per topological change, is $O(|E_\pi|)$ where E_π is the set of links that may be traversed by messages of π, when it is running in a dynamic network in conjunction with our reset procedure. For many applications this set of links is only a small fraction of all the links in the network. The quiescence time complexity of the reset procedure is $O(n_\pi)$ where quiescence time is the time elapsed from the last topological change until the reset procedure reaches quiescence (i.e., global termination), and n_π is the maximum number of nodes in the network induced by E_π. The space complexity of the procedure is $O(1)$ per incident link at each node.

1.1 Related work

A simple implementation of a reset procedure is to associate a logical time stamp (*version number*) ([Fin79, Lam78]) with each message of the static algorithm. The largest time stamp seen by a node is considered to be the current time stamp of this node. When messages with different time stamps collide, the higher time stamp overruns the other. When a topology change is detected by a node, the node restarts the static algorithm all over with a time stamp greater than its current one. This approach has two drawbacks: (1) the overhead of resetting the static algorithm is a multiplicative factor of the message complexity of the fault-intolerant algorithm, since there is an overhead associated with each message; and (2) the time stamps may increase without bound. The advantage of this approach is that it works even in the absence of FIFO message delivery on links.

To avoid the first drawback in systems with FIFO message delivery on links, the burden of carrying the time stamp can be relegated to a snapshot algorithm [CL85]. Essentially, before a higher version message is transmitted on a link, a token is sent announcing that all subsequent messages on that link assume the new time stamp, until the next token. This solution is a variation on Finn's *unbounded-version-number-algorithm* and was suggested by Cidon and Segall [CS82]. It separates the different versions of the static algorithm by snapshots time-stamped with Lamport clocks.

To remove the second drawback, Finn proposed to pass only differences in time stamps over a link. To bound the magnitude of these differences by $2 \cdot m$, where m is a bound on the total number of links in the network, a recovering link is not added to the active topology until a snapshot has successfully terminated on both ends of the link. However, recently Soloway and Humblet [SH86] gave a counter-example to Finn's bounded solution. Essentially, Finn erred in using a fault-intolerant termination detection algorithm to detect the termination of the snapshot. Soloway and Humblet [SH86], proposed ways to correct Finn's algorithm. Their procedure has $O(n \log n)$ quiescence time complexity, $O(n^2 \cdot |E|)$ message complexity per topological change, and $O(n)$ space complexity, where n and $|E|$ are the total number of nodes and links in the network.

An algorithm similar to our reset procedure was independently discovered and employed by Segall, Gopal and Baratz [SGB87], in the context of database queries in faulty networks.

The rest of the paper is organized as follows. In Section 2 the model is presented. In Section 3 a fail-safe reset procedure and its use in transforming a static algorithm to a fail-safe one, are developed, analyzed and proved correct. Concluding remarks are saved to Section 4.

2 Model

We model a dynamic network by incorporating a few variations to the standard model of static fault-free asynchronous networks.

2.1 Static Networks and Algorithms

For the static network model, we assume the standard asynchronous, point-to-point message-passing model as in [Seg83, Gal82, GHS83]. The network topology is described by an undirected communication graph $G = (V, E)$, where V is the set of nodes, representing processors of the network, and E is the set of edges, representing bidirectional communication lines operating between neighboring nodes. Nodes communicate only by exchanging messages over the links of the network. Messages sent over a link arrive error-free at the other end, after an arbitrary but finite delay. There is no common memory shared by the nodes and there is no global clock.

A distributed algorithm consists of a set of programs, one at each node of the network. The programs (which are identified with nodes) include standard computer instructions and message-sending commands. The algorithm proceeds by exchanging messages between these programs (nodes).

A distributed algorithm is started by an arbitrary set of nodes which are awakened by external signals at arbitrary times and begin running their programs. Other nodes may be awakened by the reception of a message of the algorithm. Reception of a message triggers a local computation and a possible consequent transmission of new messages. Local computation is always terminating, and w.l.o.g. it is assumed to be instantaneous. Thus the algorithm evolves from the event of message reception to the event of another message reception. Global termination of the algorithm, called the *quiescence* of the algorithm, is reached when there are no messages in transit.

The computations performed by a distributed algorithm in a communication network $G = (V, E)$, are specified by an input-output relation with respect to the nodes of G. An *input* (respectively *output*) *configuration* is a map $I : V \rightarrow IN$ (respectively $O : V \rightarrow OUT$) where IN and OUT are specified domains of values. A distributed algorithm π computes an input-output relation R_π, if any execution of π on G which starts with input configuration I terminates with output configuration O, and $(I, O) \in R_\pi$. Moreover, for each pair $(I, O) \in R_\pi$ there is an execution of π which if started with I, terminates with O. In an *off-line* algorithm the input values are given at the nodes before the algorithm is started and the output is produced upon termination. In an *on-line* algorithm input values are provided during the algorithm execution and the algorithm may not terminate. In an on-line algorithm the output values are produced during the execution. In the present paper we are concerned only with the adaptation of static off-line algorithms.

We assume that I is given to the algorithm in the form of initial values to variables in nodes. Furthermore, we assume that "ports" through which messages are sent and received are also given to the algorithm as part of I. Thus for example, after receiving a message on link e, the algorithm may send a message back on link e. Regularly the output O is produced distributively by the algorithm at each node, writing into prespecified part of its memory. A few algorithms are of interest because of their *communication-pattern* rather then written output. Examples of such algorithms are the Synchronizer [Awe85] and the algorithm for Snapshots [CL85]. By requiring each processor to record its complete local communication history on its output memory, the correctness of any terminating asynchronous algorithm can be specified by such an Input Output relation. That is, given the communication logs at every node, it is possible to verify that an intended communication pattern was indeed computed.

In the computation as described above, I is not allowed to change after the algorithm has started. Yet, we may consider such a changes by allowing the algorithm to accept messages to this effect from the "environment". Many algorithms that "sense" their environment and react to it are of this type. An algorithm is *input-safe* if even when I changes while it is in progress, once the changes stop, it will eventually terminate with O that corresponds in R_π to the last I.

Interactive distributed algorithms which run forever and whose inputs arrive during the algorithm execution, (such as end-to-end communication protocols and other distributed operating system functions) can be viewed as a special case of input-safe algorithms. Such algorithms are required to irreversibly "record" part of the final output while additional input-changes transpire. The adaptation of such algorithms to dynamic networks is relegated to a forthcoming paper [?].

2.2 Fail-safe Algorithms

Notice that since computation π is defined with respect to an underlying network G, if we define an input-output relation for all networks, then as with input changes we would like to consider changes in G that occur during the execution of the algorithm. We would like to enlarge the input-safe notion to encompass changes in the underlying communication network. Unlike the input configuration I, the question of how the changes in G are input into the algorithm while it is in progress is not straight-forward. Obviously, these changes should be reflected in the "port" assignment to nodes, but how will this assignment change? and what will be the relation between changes in different nodes? In the next subsection, we describe the model of a *dynamic network*, which precisely answers this questions. An input-safe algorithm where the input notion is enlarged to include changes in the underlying communication network, is called a *fail-safe* algorithm.

Specifically, an algorithm that computes $R_\pi(G)$ when run on a network with topology G, is a fail-safe algorithm, if it is input-safe, and if when run on a dynamic network, once input and topological changes stop with input configuration I' and topology G', it will eventually terminate with output O', such that $(I', O') \in R_\pi(G')$. A fail-safe algorithm is certainly a static algorithm. Therefore, a fail-safe algorithm is called the *fail-safe version* of its static incarnation, in the sense of input-output relation. We will usually employ this notion in the reverse direction, namely, given a static algorithm we will look for a fail-safe algorithm whose static incarnation has the same input-output relation as the given algorithm. The fail-safe version of a static algorithm π is denoted by $FS(\pi)$.

2.3 Dynamic Networks

In the present paper we deal with networks of *dynamic* topology, where the number of times that links may fail and recover is unbounded. Nodes do not fail. Notice, however, that we may model a node failure by failure of all its adjacent links.

Link failure is simply the loss of the connection between two end-points, which may result in a loss of a message. However, once a message is delivered, it is delivered correctly. Link recovery is the recovery of a failed connection. A lower-level protocol, referred to as the *link protocol*, operates on each network link. Its purpose is to maintain communication over the link, and to detect failures and recoveries with respect to that link.

Since we are not concerned here with link protocols, all we need to do is define the communication properties of a network link which are secured by the underlying link protocol [?]. This will enable to rigorously prove the correctness of our protocol. The assumptions that we use are given in the following sub-section. Those essentially consist of the first 4 axioms of [AE86], which formalize the standard assumptions in the literature [Seg83, CR84, SH86].

The Axioms below capture the effects of having an underlying handshake protocol between the two ends; This protocol is invoked by one end when it perceives the link to be operating again following a failure of that link. Axiom(A1) establishes that there is some mechanism by which a node in one end declares the link down (time-out, too many retransmissions are requested, etc.) and up. Axiom(A2) establishes that when a link comes up the two end-nodes are synchronized as to which previous messages were received and which ones were not, i.e. the link comes up "clean". Moreover, together with the messages of the handshake protocol, the events of message reception and transmission form a partial order in a natural way. Axiom(A3) implies the FIFO property, and Axiom(A4) says that eventually both ends agree on the status of the link (if it settles at one end).

A1) At each end point of a link, the link protocol generates an alternating sequence of 'WAKE' and 'FAIL' signals. The time intervals at a node between a 'WAKE' and the following 'FAIL' are said to be the *operating intervals* of the link at that node. Messages can be sent (but are not guaranteed to be received) and received over the link only during operating intervals.

Definition: An interval on one side of the link is a *mate* of an operating interval on the other side if a message sent during one interval was received during the other or vise versa.

A2) An interval can have at most 1 mate. Moreover, for any two distinct intervals at the same side that do have mates, the mate of the earlier interval precedes (at the other side) the mate of the later interval.

A3) Messages received at one end are received in the order they were sent at the other end (FIFO on links).

A4) If an infinite operating interval has a mate, then this mate must be an infinite interval as well. Moreover, every message sent during infinite interval arrives successfully at the other side.

2.4 Complexity Measures

In this subsection we define the complexities of fail-safe algorithms, by which we can evaluate them. Obviously these complexities may be affected by the whole sequence of input and topological changes. All complexities are worst-case complexities. They are measured with respect to worst-case execution, and worst possible finite sequence of input or topological changes in worst possible timing.

Considering the whole sequence as an input, the *amortized message complexity* is the number of elementary messages sent, divided by the length of the input sequence (i.e., the total number of changes), where an elementary message contains at most $O(\log n)$ bits if the algorithm requires that nodes have identifiers, and $O(1)$ otherwise. The *space complexity* is the amount of local memory required. The *quiescence time complexity* is the time elapsed from the last input or topological change until quiescence is reached, given that message transmission delay and inter-message delay are at most one time unit. Finally, the *quiescence message complexity* is the maximum number of messages whose send event is not succeeded (in the partial order of events) by an input or topological change.

As we will show in the next Section our reset procedure by itself is a fail-safe algorithm with respect to its communication pattern. In this paper we are only concerned with the complexity of the reset procedure.

3 Local Reset Procedure

3.1 Execution Subnetwork

A reset procedure resets an execution of a distributed algorithm, which we call *obsolete execution*. It purges all the messages and any other effect of the obsolete execution, resetting the corresponding algorithm to its initial state at each node which has participated in the obsolete execution.

A Reset procedure that resets the whole communication network following each input or topological change is called a *global* reset procedure. A global reset procedure may be wasteful in time and messages, since the algorithm that needs to be restarted may have run only on a small subnetwork. The main algorithm in this paper is a *local* reset procedure; a procedure that will reset only the nodes that need to be reset. It can be trivially modified to a global reset procedure.

Basically, if a node realizes that it ran with obsolete input, its computation and the computation of all nodes that have received a message from it are obsolete. Moreover, the computation of all nodes that it received a message from are obsolete as well, since part of the "effects" they were supposed to produce are lost in an obsolete computation, thus they are obsolete too. Thus, each computation that received a message from an obsolete computation or each computation whose message was received in an obsolete computation is also obsolete.

To retain the information about the obsolete relation between computations, at each point of time in the execution of a distributed algorithm we define and distributively maintain an *execution-subnetwork*, which is a subnetwork of the directed version (two anti-parallel links for each undirected link) of the communication network. Each node maintains a set of incident outgoing links which currently belong to the execution subnetwork. For a message reception at node i from node j, or a message transmission from i to j, node i adds the directed link (i, j) to the execution subnetwork. After node i is reset, it is restarted with no incident link in the execution subnetwork. When node i is aborted, only the computations at nodes that can reach it in the execution-subnetwork need to be aborted.

3.2 Informal Description and Motivation

Suppose that the static algorithm π is in progress at node i, when node i realizes that the input it has used is obsolete. All the computation invested in nodes that were affected by the input change in node i are obsolete too. Had we the capability of "freezing" the whole network in the same instant, we could reset these nodes to their respective initial state, purge all the messages in transit from them, and then "unfreeze" the network with the new input at node i.

Let us first assume that both the underlying communication network and the execution subnetwork are static. Thus w.l.o.g. we can assume that both are the same network. Freezing and restarting a whole static network can be accomplished by the celebrated "Snapshot" algorithm of [CL85].

Let L_i be the set of links incident to node i. When i realizes that it would like the algorithm to restart, it sends Abort messages on L_i, and "freezes" its state. From now on it will receive messages of π, but not process them until it is "unfrozen". Node j that receives an Abort message while unfrozen, freezes and forwards Abort messages on L_j, and acknowledges (Acks) the Abort message it received. All Abort messages received are immediately Acked. A frozen node j becomes unfrozen when it has received Acks to all the Aborts it sent on L_j . When node j becomes unfrozen it purges all the messages of π that were received on an incident link prior to the reception of the Abort on that link. These messages are messages from a previous execution and were sent by nodes which have later restarted, and therefore are obsolete messages. It then re-initializes its begins and start "simulating" the reception of messages which were received but not purged. If no such messages exist, then it can spontaneously restart.

Notice that for the Abort messages and their Acks, the messages of π are invisible. Thus we may speak about the "Abort-algorithm". This algorithm is a variation of the snapshot algorithm [CL85]. To work over dynamic networks, (as well as only over the execution subnetwork that is in a way dynamic), we need a fail-safe version of the Abort-algorithm. Notice that the "output" of the Abort algorithm is a special communication pattern.

To confine the Abort algorithm to the execution subnetwork, each node sends the Abort messages only on its incident links which it designated to be in the execution subnetwork.

The Abort algorithm is not fail-safe. What is more surprising is that if the network is static but runs the abort algorithm only on the execution subnetwork, it might enter a nonterminating cycle of global states while it is in progress. It has two undesirable features. First, the Abort algorithm in conjunction with the ability of unfrozen nodes to join into the execution, may enter into a nonterminating cycle of global states, by which Abort messages are "chasing" messages of algorithm π in a circuit. The Abort messages "clean" the way for the application messages, and the application messages "reintroduce" the links in the circuit into the execution-subnetwork. Second, a less serious problem is the fact that the same node might go through many cycles of being frozen and unfrozen due to the same invocation of the Abort algorithm.

To prevent the cycles, we hold a node frozen until all the nodes which it may have "introduced" into the snapshot algorithm by its invocation of the Abort algorithm, have been frozen. This can be accomplished by delaying the ACK to the the Abort message in each invocation of the algorithm at a node, until it has received all its pending Acks. This standard technique can be viewed as a special case of Dijkstra and Scholten's termination detection procedure [DS80]. When the Abort algorithm propagates it creates a tree rooted at nodes that invoked the algorithm. When a node receives the ACKs to all the Abort messages it sent, it is guaranteed that all the nodes it introduced into the algorithm have been frozen. We can now unfreeze nodes going from leaves inward as they become fully acknowledged, or keep them frozen until another event happens as explained below.

To solve the second problem as well, we do not unfreeze nodes from the leaves inward. This may create a situation where a leaf of the same tree is unfrozen while other branches of the tree are still in "forward-propagation." Rather we hold the nodes of the tree frozen until the root of the tree has received all its pending ACKs. At this moment we know that no branch of the tree is propagating, and we may unfreeze the tree going from the root outward. This may be accomplished by "unfreeze" messages from parents to sons in the tree.

In the context of dynamic network, since a link that recovers does not participate in the execution subnetwork, it is no different than an input change. Thus if the recovery happens when the node is frozen it requires no special treatment, while it triggers the modified Abort algorithm above if it happens at an unfrozen node. The second kind of change, a failure of link e is also treated as input change with the provision that when e fails while the node is frozen, it is considered as if e was not there to start with. That is to say, if an Abort or an ACK were expected on it, they may be considered as if they had been received. Moreover, if e led from the node to its parent in the tree, then the node henceforth considers itself as an originator of the Abort algorithm.

The modified Abort algorithm is a fail-safe version of the Abort algorithm. Following the last input or topological change, eventually the communication pattern of the Abort algorithm is established, followed by quiescence. Since following the last run of the Abort algorithm the network and the input do not change, the last run of π is its static execution on the final topology and inputs.

The formal description of the algorithm consists of two modules: the Reset procedure, and a *Manager* process. The reset procedure is the modified Abort algorithm; the Manager process is a supervisor process, which does not send messages. However, any message of the application algorithm π or the reset procedure, which arrives at a node, is first passed to the Manager. The Manager then decides whether to delay the message, purge it, or pass it on to the intended algorithm, as a function of the state of the reset procedure. It also resets the state of π to its initial state and supplies it with the set of operational links, and supplies the reset procedure with the execution-subnetwork on which it should propagate. Both π and the reset procedure are assumed to treat a message they received from the Manager in the same manner they would treat it when receiving it directly from the link.

In Figure 1 and 2 the formal code for the Reset procedure and the Manager process is given.

The Manager maintains a queue Q of delayed messages of π. Each message in the queue is stored in the form (M,e) where M is the message and e is the link over which it arrived. There are four procedures which operate on the queue Q.

1. **queue(M,e)** which queues (M,e) at the end of Q.
2. **dequeue(Q)** which dequeues (M,e) from the front of Q.
3. **Q:=Q** $-Q|_l$ which removes from the queue all messages (M, e) such that $e = l$.
4. **empty(Q)** is a boolean function which is true iff Q is empty.

We assume that message delivery from the Manager to the reset or π takes no time.

3.3 Properties of the Reset procedure

Theorem 1 *Let π be executed in conjunction with the reset procedure and the Manager algorithm, then following a finite number of input and topological changes, π, the reset procedure, and the Manager algorithm enter quiescence where the state of π is the output of π when executed on a static network with the final topology and input.*

The total number of messages sent by the reset procedure in an execution of *FS(π)* is at most $k \cdot |E_\pi|$ where E_π is the total number of links which were used at least once by messages of π in the execution of *FS(π)*, and k is the total number of changes (inputs or topological). The quiescence time and message complexities of the reset procedure are bounded by $O(n_\pi)$ and O(E_π), respectively, where n_π is defined in accordance to E_π.

The proof of Theorem 1 and the complexities are postponed to Subsection 3.4.

The reset procedure performs its required tasks (as proved in Subsection 3.4), that is, it purges all the effects of executions which encounter topological changes and it terminates a finite time after the last topological change. However, a careful look at the interaction between the reset procedure and algorithm π revels that *FS(π)* can produce intermediate executions which could not have been produced in a run of π on a static network. The execution of *FS(π)* in a dynamic network can give rise to *inconsistencies.*

In between two resets (i.e., in the time intervals in which nodes are in Ready state) we define the topology seen by a node and the topology seen by a message in the execution of $FS(\pi)$ as follows: The topology seen by a node that received no message consists of its incident links. The topology seen by a message is the topology of the node that sent it at the time of transmission. Upon reception of a message by a node program the topology seen by the node is updated to be the union of its previous topology and the topology seen by the message. An inconsistency occurs when a node receives a message whose topology conflicts in at least one link with the topology seen by the node. That is, there is a link, l perceived by the node to be up (down) and in the topology seen by the message l is down (up).

To see that the execution of *FS(π)* could enter an inconsistent state consider the following scenario: The network has 3 nodes, A, B, and C connected in a cycle by three links $(a,b), (b,c)$, and (c,a). Initially all links are up. Node A starts an execution of π by sending a message, $m1$, to B. Upon receiving $m1$,

Messages: **ABORT, ACK, RELEASE** ;
Node Variables:
State $\in$ **Aborting, Ready**
Used: set of links used by π, supplied by Manager ;
AckPend(e) : Boolean $\forall$ e $\in$ Used, initially false ;
Parent: a link leading from child node to parent node ;

```
For Down, Up on link L or any input change:
    if Down on link L then Used := Used - L ;
    if State = ready
    then Parent := nil ;
         Call Procedure Broadcast-Abort ;
    if Down on link L
    then if Parent=L then Parent := nil ;
         if State=aborting & (AckPend(L)|Parent=nil)
         then AckPend(L) := false ;
              Call Converge-Abort ;

Procedure Broadcast-Abort
    state := aborting
    for all e ∈ Used do:
         AckPend(e) := true ;
         send ABORT over e ;
    if Used = ∅
    then Call Procedure Release ;

For ABORT message on link L
    if State = ready
    then Parent := L ;
         Call Broadcast-Abort ;
    else send ACK on L ;

Procedure Converge-Abort
    if for all e ∈ Used not AckPend(e)
    then if Parent ≠ nil
         then send ACK on Parent ;
         else Call Procedure Release ;

For ACK message on link L
    AckPend(L) := false ;
    Call Procedure Converge-Abort ;

Procedure Release
    for all e ∈ Used send RELEASE over e ;
    State := ready ; /* trigger the Manager */

For RELEASE message arriving on link L
    if State=aborting & L = Parent
    then Call Release ;
```

Figure 1: The fail-safe Reset Procedure

```
/* Variables */
initiator : boolean ; /*Initially false, tells whether
                                        received start signal*/
Operational : the set of up links ;   /*Used as input to π

For ACK on link L:
    Deliver (ACK,L) to Reset

For RELEASE on link L:
    Deliver (RELEASE,L) to Reset

For ABORT on link L:
    Q:=Q −Q|_L
    Deliver (ABORT,L) to Reset

For link L going down:
    Q:=Q −Q|_L
    Operational:=Operational−L ;
    Deliver (Down,L) to Reset ;

For link L coming up:
    Operational:=Operational+L ;
    Deliver (Up,L) to Reset ;

For start signal from outside:
    initiator := true ;
    if state=ready
    then deliver start signal to π ;

For message M of π arriving on link L:
    if state=Aborting then queue(M,L) ;
    else
        Used:=Used+L
        Deliver (M,L) to π

For Aborting to Ready state change by Reset:
    Used := ∅ ;
    State π:=initial state ;              /* reset π locally */
    else while not empty(Q) do
        (M,e) := dequeue(Q) ;
        Used := Used + e ;
        Deliver (M,e) to π ;
    if initiator
    then deliver start signal to π ;
```

Figure 2: The code of the Manager process

B sends a message, $m2$, to C. Message $m2$ suffers large delay on link (b,c) (this link could be replaced by a long path). Meanwhile, link (a,b) goes down, and node A resets itself (its Used is now empty) and starts a new execution in which it sends a message, $m3$, to C. Now, node C receives two messages, $m3$ and $m2$, one of which saw link (a,b) up and the other saw it down. Until C receives the ABORT message, which must be on its way to it from B, it is in an inconsistent state.

Since π was designed for static networks there is no guarantee on the actions taken by the node programs while in inconsistent state. Thus the behavior of the intermediate executions and the various complexities of π have to be reconsidered.

For example, consider online static algorithms such as, end-to-end communication protocols, or other operating system functions, where at various nodes outputs are written in an irreversible way during the algorithm execution (e.g. writing to a terminal, or performing other external action). If we would run such an algorithm in a dynamic network, in conjunction with the reset procedure as described above, then nodes in inconsistent states could irreversibly write wrong outputs. Being irreversible, this outputs cannot be corrected when the ABORT message arrives. Thus in general, our local reset procedure cannot be used to adapt static online algorithms to dynamic networks.

To avoid the problem indicated above, one should either replace the local reset procedure with a global-reset procedure, which does not give rise to inconsistencies [?], or verify before taking any irreversible step that it is in a consistent state. Such a verification is done by scanning the execution-subnetwork for ABORT messages by a fail-safe broadcast and echo algorithm.

An efficient global-reset procedure can be constructed using our local reset procedure as follows: First we take the optimal Spanning Tree (ST) algorithm presented in [Awe87] and convert it into a fail-safe spanning tree algorithm *FS(ST)* by running it in conjunction with our efficient local-reset procedure and Manager process. When the *FS(ST)* algorithm terminates the root of the tree starts a fail-safe broadcast with echo on the tree which purges all effects of old executions, sweeps all incident links and reset the programs at all the nodes in the network. When this phase terminates the application algorithm may be restarted, knowing that all its previous effects were purged.

3.4 Correctness Proof

To prove Theorem 1 we first prove that in response to a finite sequence of input or topological changes the reset procedure independently of algorithm π will enter quiescence with each node in Ready State. We then proceed to show that the interaction of Manager with π is such that following the quiescence of the reset algorithm, π will enter quiescence in the intended global state.

Lemma and Corollaries 2-9 concern only the reset procedure.

Lemma 2, and 3 establish properties on the communication over a link in the reset procedure, which can be handled in relative isolation from the "guts" of the algorithm.

Lemma 2 *At a global state of the system, if at node i AckPend(e=(i,j)) = true, and at node j state=Aborting and e$\neq$Parent(j), then eventually at i either AckPend(e) is false, or link e goes down at i.*

Sketch-of-Proof: Notice that if AckPend(e)= true at i then there are three possibilities: (1) it will stay that way forever, (2) eventually AckPend(e)= false as a result of a reception of an ACK, (3) link e will go down at i. Consider the beginning of the last Aborting interval at i. An ABORT to j was then sent on e from i to j, AckPend(e)= was set true at i, and it necessarily stayed so till now, since AckPend is set only once in a Aborting interval. If the ABORT does not arrive at j then by Link Axiom(A4) i must be notified that link e went down. Thus assume the ABORT did arrive. At this event j either generates an ACK, or Parent(j)= e. Suppose Parent(j)$\neq$ e, then either the ACK will arrive, or again by Link Axiom(A4), e must eventually go down at i. Thus the remaining case is that the ABORT has arrived at j and Parent(j)=e. Then a precondition to change Parent(j) from e is either to send ACK on it or that the link will fail at j, in both cases either the ACK is received, or by Link Axiom(A4), i is notified of failure of e. Since the reception of an ACK sets AckPend(e)= false at i, the lemma is proved. ∎

Lemma 3 *At node i, an ACK received at i on link e=(i,j) in event a, was sent at j after the reception of the last Abort message on e generated at i prior to a.*

Sketch-of-Proof: It can be seen from Link Axiom(A2) and Lemma 2 that there are no "stragglers" ACKs. Each ACK is generated in an injective correspondence to an ABORT. To establish the lemma it suffices to prove that if an ACK is not received when expected in response to an ABORT, then the corresponding ACK is not received. An ABORT (which sets AckPend(e=(i,j))) can stay unacknowledged forever in two ways. Either AckPend(e)=true forever, which implies that the ACK is not received, or e goes down at i before the ACK arrives. In the later case, by Axiom(A4) eventually e will also go down in j. If an ACK was sent at j before e goes down at node j, then axioms (A2) and (A4) guarantees that the ACK will be lost. Otherwise, if the link went down at j before the ACK was generated then the ACK that corresponds to this ABORT will not be generated, and as a result will not be received. ∎

Let *interval* at a node be a maximal period of time in which its *state* does not change. Obviously each node alternates between *Aborting* and *Ready* intervals. An interval is called *final* if it stretches to infinity (in the direction that time progresses).

The next two lemmas and corollary establish that an Aborting interval cannot be a final interval. That is, state Aborting is a transient state.

Lemma 4 *In a global state of the system, a path of nodes $i_1, i_2, ..., i_k$ such that for $j = 1,..,k-1$, e=$(i_j, i_{j+1}) \in$ Parent at i_j, is a simple path.*

Sketch-of-Proof: Notice that in each aborting interval, the Parent node is determined only in the transition to the aborting interval. Also, Parent at node i is not nil only in an aborting state. Thus all the nodes in the sequence are in aborting state in the global state. We claim that the current aborting interval at node i_j was created by the reception of ABORT message generated in the current aborting interval at i_{j+1}. Obviously it was created by an ABORT from i_{j+1} because of the existence of Parent. If the ABORT message was created by the previous aborting interval at $i_[j = 1$, then it means that the link was all the time up at i_j since the reception of that ABORT message till now. Since i_{j+1} has since already switched to ready it must have received, by lemma 3, the intended ACK on the link and sent RELEASE on it. Since the ACK was sent by i_j all its Used link were ready and the RELEASE from i_{j+1} should have terminated the aborting interval in i_j, contradiction. Thus the aborting interval in i_j started strictly after the aborting interval in i_{j+1} which establishes the lemma. ∎

Lemma 5 *If at i there exist e$\in$Used and such that AckPend(e)=true then eventually for all e$\in$Used, AckPend(e)=false.*

Sketch-of-Proof: Since it must be an Aborting interval, and in such interval Used may only shrink, the other possibility is that AckPend(e)=true forever and as a result its aborting interval stretches forever. Follow that link to the next node. The same argument applies there, either it eventually sends the required ACK, or by Lemma 2, the only other possibility is that it is also waiting on a link indefinitely and therefore this link is up forever and is a Parent in the other node forever. By lemma 4, since we cannot close a cycle, we must reach a node that for all its links in Used AckPend is false, which contradicts the fact that at its predecessor the link which leads to it has AckPend=true forever. ∎

Corollary 6 *If at i state=aborting, then eventually state=Ready.*

Sketch-of-Proof: By Lemma 5 eventually all Used links are Ready. By lemma 4 tracing a path back on Parents we must end up at a root node (node without a Parent). By lemma 5, this node must eventually start a broadcast of RELEASE messages which either reaches every node in the path, or some link on the path fails. In the latter case, one of the endpoints of this link becomes the root of a disconnected sub-tree and starts broadcasting RELEASE messages over that sub-tree. By induction on the depth of the tree, every node will eventually execute the Release procedure. ∎

The next lemma and two corollaries establish that there exist a final interval in each node, and moreover this interval is a Ready interval.

Associate each aborting interval and each RELEASE message with an input and or a topological change as follows: If aborting interval A terminates when Parent=$\emptyset$, then associate the interval with the last input or incident topological change that occurred at i prior to the termination of A. A RELEASE message sent upon termination of an aborting interval B is associated with the same change B is associated with. An aborting interval B that terminates as a result of receiving RELEASE message on Parent is associated with the same change that the RELEASE message is associated with.

Lemma 7 *Distinct aborting intervals in the same node are associated with distinct topological or input changes.*

Sketch-of-Proof: First notice that there are no "stragglers" RELEASE messages, which release a node when the are not supposed to. This follows similar arguments as in Lemma 3, with the additional observation that all prior RELEASE messages on a link which becomes Parent as a result of the reception of the ABORT are guaranteed by the FIFO (Axiom(A3)) to be "swept" by the ABORT. Obviously, all aborting intervals that terminate not as a result of receiving RELEASE message (root interval) are associated with distinct changes (a failure or recovery is perceived as two distinct changes, one in each incident link). Consider the global state when a root node generates a RELEASE. By Lemma 4 the set of Parents on which the RELEASE may propagate is currently a tree, and this set may only shrink later as a result of failures. Thus since RELEASE messages from (root interval) propagate on a tree, the result is established. ∎

Corollary 8 *Given a finite sequence of input and topological changes, eventually the reset procedure reaches quiescence.*

Corollary 9 *There exists a final interval in each node. Moreover this interval is a Ready interval.*

We now turn to Algorithm π. We are concerned only with the execution of π in the final intervals. At the beginning of these interval the state of π in each node is the initial state and the final input and incident topology. Lemma 10 below establishes that all messages received in final intervals were sent in final intervals. Moreover, by the interaction of the Manager with π all messages which are sent in final intervals and therefore not followed by ABORT messages and are not purged. Again by Lemma 10, they must be received in final intervals. Thus the execution of π in the final intervals is its static execution on a static network with the final input and topology, which establishes Theorem 1.

To prove Lemma 10 we say that a Ready interval A *affects* Ready interval B, if a message of π sent during A was received during B. Notice that received in this context means processed by π since messages may be delayed by the Manager. Let the relation *affect** be the reflexive commutative transitive closure of affect. Then the affect* relation induces equivalence classes on the Ready intervals.

Lemma 10 *Let A and A' be two Ready intervals in the same equivalence class, then if A is not final then A' is not final.*

Proof: It suffices to prove the lemma just for the reflexive commutative closure of affect. The extension to affect* will follow by induction on a path of intervals connecting A and A'. Thus we have to consider the case that A affected A' or vise versa. Since we do not have a construct for a node to send a message to itself, A and A' happened at distinct node. Assume A occurred in i and A' occurred in j. If A affected A', then the link e=(i, j) was put on Used in i. Since A is not final it was followed by aborting interval. If e was still in Used at the beginning of the aborting interval, then ABORT was forwarded on it. If not, e must have failed in i during the Ready interval. As in Lemma 2 both cases force an aborting interval in j. Following the reception of the message from i, and the FIFO guarantees that this aborting interval succeeds A'. Similarly, if A' affected A, then the moment the message from j is received at i, e is put on Used in i, and the previous argument carries verbatim, only that instead of FIFO the causality relation between messages insure that the aborting interval in j succeeds A'. ▮

3.5 Complexity

By Lemma 7 the number of aborting intervals in each node is bounded by the length of the sequence of changes. The size of this sequence is k, the total number of input changes. Since in each aborting interval a node sends a constant number of messages on each link, the message complexity of the Reset procedure is at most $k|E_\pi|$ where E_π is the union of all the links which messages of π have traversed during the execution of π in conjunction with the reset procedure and the manager in the dynamic network. Obviously $|E_\pi|$ is bounded by $|E|$, the maximum size of the communication network. Yet for many instances of algorithms when analyzed, one can establish a much better bound (e.g. an algorithm that wakes up and counts the number of incident links of one of its neighbors. In such an algorithm E_π is zero). We do not bound E_π by the size of the network which π may traverse in a static network, since π when run in conjunction with the Reset procedure in a dynamic network may temporarily give rise to a state which may not occur in a "static-run", as shown in Section 3.3.

Similarly it can be shown that the quiescence time and message complexity of the reset procedure are bounded by n_π and O(E_π), respectively, where n_π is defined in accordance to E_π.

We mention in passing that if one is concerned about the number of messages sent by algorithm π during the intermediate ready intervals, one may control π; Stop the execution of π at various times and check if it entered an undesirable state, if it did, delay the algorithm until the Reset procedure will abort it. It can be shown that using the controller of [AAPS87] we can bound the total number of messages sent by π by $k \cdot (C_\pi + V_\pi \cdot \log^2 V_\pi \cdot \log C_\pi)$, where C_π and V_π are the message and node complexities of π in the static setting.

4 Conclusions

We have presented a mechanism to adapt any static application algorithm to run in a network whose topology dynamically changes. The mechanism works by restarting the application following each topological change. The restart process runs only on the part of the network traversed by the application algorithm. This has the advantage that following short stability period, if the application did not make much progress following its restart, correspondingly the duration of the new restart process will be short. More importantly, many application algorithms involve only a fraction of the whole network. Correspondingly, not only will the restart happen fast, but it will also not interfere with other applications that are unaffected by the topological change, and are running in other parts of the network.

The approach we took to the adaptation, namely restarting, may be wasteful. It assumes that all the "work" invested by node prior to the topological change is obsolete. In many applications in which we have some "understanding" of what the application algorithm is doing, we may try to replace only what locally looks like the obsolete information, saving the rest. With "slowly" changing topology this approach may be superior to ours. For instance, for the fail-safe topology maintenance algorithm we achieve $O(n \cdot |E|)$ amortized and quiescence worst case message complexities and $O(|E|)$ quiescence time complexity. Applying our techniques to the spanning tree of [Awe87] result in a fail-safe spanning tree algorithm with $O(|E|)$, $O(n)$, and $O(|E| + n \log n)$, amortized message, quiescence message and quiescence time complexities, respectively. However, for the *average case* complexity, the special purpose designed algorithms are usually more efficient. E.g., in the topology maintenance algorithms of [?, ?] following a "small" change in the topology only a little computation is invested. While our algorithm recomputes the whole topology after each topological change, their algorithms preserves the unchanged topology information and recomputes only the changes.

The quiescence time of our procedure is $O(n_\pi)$ time. $\Omega(n_\pi)$ is clearly a lower bound on the time complexity of any reset procedure, however, one could envision situations in which the static application algorithm could be restarted $O(1)$ time after the last topological change. In this case the messages of the reset procedure would run ahead of the messages of the new version of the static algorithm. We conjecture that in general, at least $\Omega(n_\pi)$ time is required from the last topological change to the restart of the static application algorithm.

Acknowledgments

The authors wish to express their thanks to Paul Beame, Ken Berman, Danny Dolev, Bob Gallager, Oded Goldreich, Nancy Lynch, Michael Merritt, Serge Plotkin, David Shmoys, and Mark Tuttle for useful discussions.

id AA29892; Thu, 6 Aug 87 07:21:59 PDT

References

[AAPS87] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks. Efficient fault tolerant election algorithms with different termination conditions. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, 1987. Work in progress.

[AE86] B. Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *NETWORKS*, 16:381–396, 1986.

[AG85] B. Awerbuch and R. Gallager. Distributed bfs algorithms. In *Proceedings of IEEE conference on Foundations of Computer Science*, October 1985.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[Awe87] B. Awerbuch. Linear time distributed algorithms for minimum spanning trees, leader election, counting and related problems. In *Proc. of the ACM Symp. on Theory of Computing*, May 1987.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, January 1985.

[CM85] K. M. Chandy and J. Misra. *A Paradigm for Detecting Quiescent Properties in Distributed Computations.* Technical Report TR-85-02, Department of Computer Science, University of Texas at Austin, January 1985.

[CR84] I. Cidon and R. Rom. *Computer Networks with Changing Topologies Part I: Protocol Extension.* Technical Report, Technion, Haifa, February 1984.

[CS82] I. Cidon and A. Segall. Network resynch protocols. 1982. unpubliched manuscript.

[DS80] W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11-1:1–4, August 1980.

[Fin79] S. G. Finn. Resynch procedures and fail-safe network protocol. *IEEE Tranc. on Comm.*, COM-27:840–845, June 1979.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[Gal76] R. G. Gallager. A shortest path routing algorithm with automatic resynch. March 1976. Unpublished note.

[Gal82] R. G. Gallager. *Distributed Minimum Hop Algorithms.* Technical Report LIDS-P-1175, M.I.T. Lab for Information and Decision Systems, January 1982.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[Seg83] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1), January 1983.

[SGB87] A. Segall, I. S. Gopal, and A. E. Baratz. Query algorithms in computer networks. In *Proceedings of 2nd International Workshop on Distributed Algorithms*, July 1987.

[SH86] F. R. Soloway and P. A. Humblet. *On distributed network protocols for changing topologies.* Technical Report, MIT Lab.

for Info. and Dec. Sys., May 1986. Report
LIDC-P-1564.