A Leader Election Protocol for Fault Recovery in Asynchronous Fully-Connected Networks *

Massimo Franceschetti Jehoshua Bruck

California Institute of Technology
Mail Code 136-93
Pasadena, CA 91125
Email: {massimo, bruck}@paradise.caltech.edu

Abstract

We introduce a new algorithm for consistent failure detection in asynchronous systems. Informally, consistent failure detection requires processes in a distributed system to distinguish between two different populations: a fault free population and a faulty one.

The major contribution of this paper is in combining ideas from *group membership* and *leader election*, in order to have an election protocol for a *fault manager* whose convergence is *delayed* until a new consistent view of the connectivity of the network is established by all processes. In our algorithm a group of processes agrees upon the failed population of the system, and *then* gives to a unique leader, called the *fault manager*, the possibility of executing distributed tasks in a centralized way.

This research and the new perspective that we propose are driven by the study of an actual system, the Caltech RAIN (Reliable Array of Independent Nodes), on which our protocol has been implemented in order to perform *fault recovery* in *distributed checkpointing*. Other potential applications include fault tolerant distributed database services and fault tolerant distributed web servers.

*This work was supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, by an IBM Partnership Award, by DARPA through an agreement with NASA/OSAT and by a fellowship from Telespazio spa.

1 Introduction

In fault tolerant asynchronous distributed systems, it is necessary for processes[1] to be able to monitor one another, keeping track of failures and recoveries of the system's members. Processes might need to take actions based on such failures and recoveries.

As an example, we may consider a distributed fault tolerant server, that works as follows [18]: a request for a particular service is multicast to all processes in the system, the operational process having the smaller id services the request. In order to perform this task, all processes are required to agree on which of them are currently operational and on their ID's. When such knowledge is lost, anomalous situations in which no process responds to a request (because all operational processes believe that there is a failed process with a smaller id that is still operational), or in which more than one process responds (because some operational processes believe that all processes with smaller id's have failed when they have not), may arise. Moreover, when a process changes its status from operational to failed or vice versa, it is necessary for all processes to agree to handle incoming requests before or after such status change. Processes should be able to *consistently* decide on which of them has to respond to incoming requests.

In the example presented above, a need for coordination arises because processes do not perceive changes in the state of the network at the same time. Two possible approaches for achieving such coordination are *leader election* and *group membership.*

The first approach requires running an election protocol each time a failure or recovery is suspected. *Leader election* is one of the fundamental problems in distributed computing [13]. The election of a leader brings a distributed system to a nonsymmetric global state. Once a leader has been determined, distributed tasks such as consensus, resource allocation, load balancing etc., may be solved by a centralized approach.

The second approach is based on *membership services* that provide *process groups* with a consistent membership view [14] [16]. *Membership services* are useful in systems subject to failures, because when a process in a group fails, it may be removed from the group if there is another one that suspects it to be faulty. Changes in the state of a process are reflected in the membership of the group, and are ordered with respect to task management.

The problem with coordination protocols, such as *group membership* and *leader election*, is that they require some form of process agreement in the presence of failures. When developing a distributed protocol in an asynchronous system subject to failures, one should be aware of the impossibility result of Fisher Lynch and Paterson [6], that shows that *consensus* is impossible in an asynchronous system, where one process may fail silently. Chandra et al. [4] adapted the impossibility result to a weakly specified *group membership problem* that allows removal of erroneously suspected processes from the group. However, their proof requires a *liveness* property, namely, if process p_1 or p_2 leaves the group, a new view is eventually installed by at least one process in the system. Neiger [15] showed that weakening the *liveness* property enables a

[1] Words process and processor may be assumed equivalent throughout this paper.

solution of the *group membership problem*. He presented a specification that allows runs in which a membership change occurs, but no process learns the identity of the new group, and the group becomes empty with no process knowing it. However, his new algorithm has a single point of failure, since the crash of a single process (the *current leader*) hangs all processes.

The algorithm we present performs election of a leader each time a failure or recovery is suspected. However, unlike traditional *leader election*, the convergence of our algorithm is *delayed* until a consistent view of the network connectivity is established by all processes. This convergence property is reflected in *real* systems, where processes crashes always lead –after an arbitrary long amount of time– to a scenario in which a connected cluster of processes share a consistent view of the network connectivity.

Unlike *group membership*, the point of failure of our algorithm is not in a single process, but in a *connected component*. Our leader is dynamically elected depending upon failures occurrences, so that if it fails, it is replaced by another process. Only if a view of the network connectivity is established in which all processes suspect all others to have failed, our algorithm blocks and the network needs to be (re)initialized.

Our solution adopts a *logical* approach to failure detection. The algorithm is based on the idea that a change in the state of the system should not be considered, until it is perceived by all processes in the network. Therefore a process, instead of acting directly when it detects a failure (or a recovery), waits for local views of the other processes to conform with its own, designating a unique leader of the group when convergence is established. With this mechanism, at each instant of time every process is either part of a *logical* completely connected component or is in a dead state, thus preventing from performing inconsistent actions.

It is important to note that our algorithm, as those in [9] [14] [16], applies to systems that attempt to maintain a *single* agreed view of the network connectivity. These are known as *primary partition* systems, i.e. systems with no network partitions, or systems in which a *primary component mechanism* ensures that only one network partition is considered as the *logical* connected component.

In particular, we have implemented our algorithm on the Caltech RAIN system, which is based on *non-partitionable*, fault tolerant network topologies [10]. This system has the property that in presence of failures the network topology always consists of a single connected component and a number of isolated nodes. On systems different from RAIN, our agorithm may still be used in conjuction with a simple *primary component mechanism* that allows only a majority (or quorum) of the processes to be *logically* connected and to have a leader. This approach has been used for algorithms implemented on systems described in [1] [9]. Articulate dynamic voting protocols that define quorums adaptively, depending on a dynamically changing set of processes, have also been proposed [8] [12]. Such protocols are always based on underlying group mechanisms, depend from their *liveness* properties and may be implemented on top of our algorithm.

Finally, if our protocol is run on a *partitionable* network, with no mechanism for maintaining a primary component, in the case of a partition the algorithm *splits* and

elects a leader for each subset of connected processes.

In the following, we give a scenario that intuitively explains how our protocol behaves. This scenario is clarified later in the paper, when the actual algorithm is presented.

- Consider a system that initially consists of five processes, numbered 1,2,3,4,5, that are completely connected and are led by process 1.
- The system first loses process 5, i.e. 1,2,3,4 suspect 5 failed.
- Process 5 *spontaneously* enters a *dead* state. Processes 1,2,3,4 are led by process 1.
- The system loses process 1 (*current leader*) and reacts in a similar way: processes 2,3,4 are led by process 2 and process 1 enters a *dead* state.
- Processes 1 and 5 rejoin the network. Since they are *dead* processes, they do not react to any change in connectivity that occurs in their local views and remain *dead*, until the *alive connected component* led by process 2 accepts them.
- Finally, processes 1 and 5 are released and enter again an *alive* state.

Note that at any given instant of time the system is always in a consistent global state, led by a *unique* leader, and that the connected component becomes empty and the protocol blocks only when all processes enter a *dead* state, i.e. every process suspects its neighbors failed.

The rest of the paper is organized as follows. In Section 2 we describe our model. In Section 3 we describe the algorithm in a step-wise refinement fashion. We start by considering an election protocol that does not consider process crashes, then we extend it to handling failures, to handling recoveries, and finally we present our complete algorithm. In section 4 we describe an application of the algorithm as part of a *fault recovery* project. In Section 5 we draw conclusions and discuss future work.

2 The Model

Our system is modeled as a complete graph (N,E), where N represents processors and E represents communication links. We assume that each node has a unique identity defined by a unique *weight* associated with it. The network is considered to be asynchronous, i.e. there is no global clock and message transmission and node processing times are finite but unpredictable. Processors in the network may fail silently and recover. Communication is considered to be point to point and messages *successfully* sent over a link, arrive at destination in a FIFO fashion. Every send operation is considered a blocking operation, in the sense that upon sending a message, a processor waits until either the send action *succeeds* (i.e. is acknowledged by the receiving processor), or it considers the link to the receiving processor in a failed state. We assume a processor to have the ability to query the state of its communication links, using an underlying protocol that maintains a consistent history on the state of a link between a pair of nodes [11]. In particular, such protocol guarantees that:

- Every time processor 1 is unable to communicate with processor 2 (i.e. p_1 sees an *up-down* transition on the link to p_2), then processor 2 will *eventually* be unable to communicate with processor 1.
- Every time processor 1 is able to communicate with processor 2 (i.e. p_1 sees a *down-up* transition on the link to p_2), then processor 2 will *eventually* be able to communicate with processor 1.

Processor failure detection is performed *independently* at each node and corresponds to determining the state of the communication links incident on each node. From our point of view a processor fails when it loses communication links to all other processors in the network, i.e. it becomes isolated. A processor recovers becoming again fully connected to all other not failed processors.

Note that failure detection, i.e. determining the state of a link between two processors, is only equivalent to *suspecting* a crash, since accurate detection of failures (and recoveries) is impossible in an asynchronous environment. Therefore, the underlying protocol [11] that we assume existing on the system, is equivalent to an unreliable failure detector that is *eventually* consistent between a pair of nodes connected to the same link.

3 The Leader Election Algorithm

In this section first we describe a very simple algorithm for leader election, as it was presented in [17]. We call it protocol Γ. Such protocol does not consider any kind of failures, however it solves the problem of leader election in completely connected asynchronous networks. We then modify protocol Γ for handling failures, obtaining protocol Φ. From Φ we derive protocol Ψ that handles failures *and* recoveries. Finally, we introduce protocol Ω, that works in more dynamic, *real world* conditions.

3.1 Protocol Γ

We consider a leader election problem in which an arbitrary subset of nodes, called the *candidates*, wake up spontaneously and start the protocol. All other nodes are *passive* and wake up on receiving a message of the protocol. Initially, a node knows only its own identity. At the termination of the protocol, exactly one node among the candidates is elected, and all nodes know the identity of this node.

The protocol works as follows. A candidate attempts to capture all other nodes. The node that is able to capture all other nodes declares itself the leader. A candidate sends a *candidate* message on all incident edges. When a node j receives a candidate message from node i, it behaves as follows:

- If j is a candidate and $j < i$, then no response is sent.
- Otherwise, j sends an *accept* message to i.

A node that receives an *accept* message on all incident edges, declares itself the leader to all nodes.

3.2 Protocol Φ

We extend the naïve solution of protocol Γ to a system with failures.

Note that in protocol Γ a subset of the nodes spontaneously wakes up and starts the protocol, while in protocol Φ all nodes are potential candidates, and candidacy is performed depending from failure occurrences. Moreover, in protocol Γ a node initially knows only its own identity, and candidates send candidacy messages to all nodes. In protocol Φ, nodes perform the protocol depending on their weight and on the weights of the other nodes they are connected to. If a node is connected to another node, i.e. it is able to communicate with it, it is expected to know its identity.

3.2.1 Protocol Φ (informal)

Weights define an ordering between the nodes. When a node crashes it starts losing links to other nodes. The node with the lower weight in the connected network is expected to become the leader.

A node that experiences a link failure *suspects* the node corresponding to that link to be failing. If it does not have any available link towards a node with a smaller weight, it proposes to be the leader, sending *candidate* messages on all of its output links. If it does not have any available link, it goes into a *dead* state and it stops performing any action.

Nodes send *accept* messages only to the node with the smaller number they are connected to, as soon as it proposes to be a leader, *and* they agree on the state of the links the candidate lost.

The convergence condition for leader election is expressed as follows: each leader candidate, in order to declare itself as leader, needs to collect accept messages from all nodes it sent candidate messages to, if it is still connected to them.

3.2.2 Protocol Φ (formal)

$\forall$ node η
  let η_a = number of available links from η
  let $\mathcal{K} = \{i : weight(i) < weight(\eta)\}$
  let η_k = number of available links from η to nodes $\in \mathcal{K}$
  $\forall$ link loss regarding node j
    if $(\eta_a = 0) \Rightarrow$ enter *dead*
    else if $(\eta_k = 0) \Rightarrow \forall$ available link i send $cand(j)$
  $\forall$ $cand(j)$ received from a node s
    let η_s = number of available links to nodes $p : weight(p) < weight(s)$
    if [(link to node j has failed) $\wedge(\eta_s = 0)$] $\Rightarrow$ send $accept(j)$ to s

Definition 3.1 *The convergence condition for leader election may be formally expressed as follows:*
$\forall$ *node* η
  let r = *number of* accept *messages received by* η
  let q = *number of* cand *messages sent by* η

η *outputs leader* $\Leftrightarrow$ *the following condition holds:*

$$(q \neq 0) \wedge \{(r = q) \vee [(r < q) \wedge (q - r)\ cand(j) to\ node\ i : link\ to\ i\ has\ failed]\} \quad (1)$$

3.2.3 Protocol Φ, Proof of Correctness

Theorem 3.1 *Given a completely connected network of n nodes, for every $k < n-1$ node failures, the convergence condition for leader election (Def. 3.1) eventually holds for exactly a single node in the network, which is the elected leader.*

Proof:

Let $\mathcal{H}$ be the set of $n-k$ nodes that are operational after a k-node failure. Let us introduce an arbitrary long time interval [0,T]. We assume that after time T, k nodes are perceived to be faulty by all nodes in $\mathcal{H}$. In addition, we assume that at time $t \in$[0,T], the k failing nodes are perceived to be faulty only by a subset of $\mathcal{H}$.

The proof is organized as follows. First we show that at time $t \in$[0,T] there is no node for which the convergence condition in Def. 3.1 holds. Then we show that after time T, there is exactly a single node in $\mathcal{H}$ for which the convergence condition holds.

The proof of the first part is by contradiction. Let us assume that there is an elected leader for the k failing nodes at time $t \in$[0,T]. Hence, it follows from Def. 3.1 that the elected leader has collected *accept* messages from all nodes it considers operational, regarding all nodes it perceived to be faulty. This means that there is a set of nodes that agree on the leader view of the faulty nodes. Hence, there is a set of nodes $\mathcal{H}'$ that suspect all nodes not in $\mathcal{H}'$ to be faulty. Each node in $\mathcal{H}'$ cannot suspect all the k failing nodes to be faulty, because, it would imply that $\mathcal{H}' = \mathcal{H}$. This would contradict the fact that we have assumed that at time $t \in$[0,T] k failing nodes are perceived to be faulty only by a subset of $\mathcal{H}$. Neither each node in $\mathcal{H}'$ can suspect less (more) than k nodes, because, it would imply $\mathcal{H} \subset (\supset)\mathcal{H}'$ and a leader election for less (more) than k failing nodes.

We now consider time $t >$T. After time T an arbitrary node is either in $\mathcal{H}$ or is considered to be faulty by all nodes in $\mathcal{H}$. A candidate in $\mathcal{H}$ is the node with the minimal *weight*, that has sent $k(n-k)$ *cand* messages to all nodes in $\mathcal{H}$. After time T, k failing nodes are perceived to be faulty by all nodes in $\mathcal{H}$. Therefore nodes in $\mathcal{H}$ that have received k *cand* messages from the candidate with the minimal weight, have sent k *accept* messages to it. Since there are $n-k$ nodes in $\mathcal{H}$, there is exactly one candidate in $\mathcal{H}$ that received the maximum number of $k(n-k)$ *accept* messages, for which the convergence condition in Def. 3.1 holds. □

3.2.4 Example

Let us consider the complete network of four nodes depicted in Fig. 1. Let us consider a double failure of nodes $N1$ and $N2$, with the following link loss sequence: 1,6,2,4,5. Let us look at table 1 and see at each link loss what is the state of the network, in terms of messages sent and received by the nodes.

Note that node $N3$ is elected leader within the connected component composed of nodes $N3$ and $N4$.

failure: link 1				**failure: link 6**		
node	***state***	***sent***	***received***	***state***	***sent***	***received***
N1	alive	cand(2)→N3 cand(2)→N4		alive	cand(2)→N3 cand(2)→N4 cand(3)→N4	
N2	alive	cand(1)→N3 cand(1)→N4		alive	cand(1)→N3 cand(1)→N4	N3→accept(1)
N3	alive		N1→cand(2) N2→cand(1)	alive	accept(1)→N2	N1→cand(2) N2→cand(1)
N4	alive		N1→cand(2) N2→cand(1)	alive		N1→cand(2) N2→cand(1) N1→cand(3)
failure: link 2				**failure: link 4**		
N1	alive	cand(2)→N3 cand(2)→N4 cand(3)→N4		dead		
N2	alive	cand(1)→N3 cand(1)→N4 cand(3)→N4	N3→accept(1)	alive	cand(1)→N3 cand(1)→N4 cand(3)→N4	N3→accept(1) N4→accept(1)
N3	alive	accept(1)→N2 cand(1)→N4 cand(2)→N4	N1→cand(2) N2→cand(1)	alive	accept(1)→N2 cand(1)→N4 cand(2)→N4	N1→cand(2) N2→cand(1)
N4	alive		N1→cand(2) N2→cand(1) N1→cand(3) N2→cand(3) N3→cand(1) N3→cand(2)	alive	accept(1)→N2	N1→cand(2) N2→cand(1) N1→cand(3) N2→cand(3) N3→cand(1) N3→cand(2)
failure: link 5						
N1	dead					
N2	dead					
N3	alive	accept(1)→N2 cand(1)→N4 cand(2)→N4	N1→cand(2) N2→cand(1) N4→accept(1) N4→accept(2)			
N4	alive	accept(1)→N2 accept(1)→N3 accept(2)→N3	N1→cand(2) N2→cand(1) N1→cand(3) N2→cand(3) N3→cand(1) N3→cand(2)			

Table 1: Node states

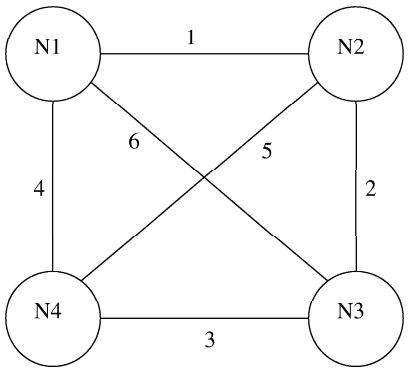

Figure 1: Complete four node network

3.3 Protocol Ψ

A natural extension of protocol Φ, deals with handling node recoveries as well as node failures. A node that recovers "regains" its links towards the other nodes in the network at different times, in a similar fashion than when it lost them when it failed.

Forcing the leader to always be a node in the connected component we use a protocol similar to Φ.

3.3.1 Protocol Ψ (informal)

The protocol forces a leader candidate to always be part of the connected component. Link repairing is treated in the same way as link loss in protocol Φ. We distinguish between four kinds of messages: $cand_{down}(j)$, $cand_{up}(j)$ $accept_{down}(j)$, $accept_{up}(j)$. Protocol Ψ is equivalent to protocol Φ in terms of the convergence condition in Def. 3.1 and response to *cand* messages.

3.3.2 protocol Ψ (formal)

$\forall$ node η
  let η_a = number of available links from η
  let $\mathcal{K} = \{i : weight(i) < weight(\eta)\}$
  let η_k = number of available links from η to nodes $\in \mathcal{K}$
  $\forall$ link loss $\vee$ link repair regarding node j
    if $(\eta_a = 0) \Rightarrow$ enter *dead*
    if $(\eta \notin dead) \wedge (\eta_k = 0) \Rightarrow \forall$ available channel i send $cand(j)_{down} \vee cand(j)_{up}$
  let η_s = available links to nodes $p : weight(p) < weight(s) \wedge p \notin dead$
  $\forall$ $cand(j)_{down}$ received from a node s
    if [(link to node j is failed) $\wedge(\eta_s = 0)] \Rightarrow$ send $accept(j)_{down}$ to s
  $\forall$ $cand(j)_{up}$ received from a node s
    if [(link to node j is available) $\wedge(\eta_s = 0)] \Rightarrow$ send $accept(j)_{up}$ to s

3.3.3 Protocol Ψ, Proof of Correctness

Theorem 3.2 *Given an n-node network, composed of a connected component of size $m < n$, and of $n - m$ isolated nodes, for every $k < n - 1$ node failures and $q < n - m$ node repairs, the convergence condition for leader election (Def. 3.1) eventually holds for exactly a single node in the network, which is the elected leader.*

Proof: The proof follows the one given for Theorem 3.1. The size of the connected component after time T is $m - k + q$. The total number of *accept* messages sent to the final leader after time T is $(k + q)(m - k + q)$, instead of $k(n - k)$. Likewise, the number of *cand* messages of a leader candidate towards *healthy* nodes after time T is $(k + q)(m - k + q)$. The proof proceeds as in Theorem 3.1, substituting n with m and adding the parameter q. □

3.4 Protocol Ω

Protocol Ψ handles both node failures and recoveries. However it is not completely general, and presents some problems in dynamic conditions.

Protocol Ψ considers a node to fail losing all of its communication links to other nodes, and a node to recover, regaining them. In a "real world" environment a node may lose some communication links to other nodes and regain them, without becoming completely isolated from the network. This may happen because of inaccurate failure detection performed by some nodes, or simply because a node that crashes (or recovers), may present transient conditions, during which its state alternatively "bounces" between an *up* and *down* condition, before stabilizing.

Protocol Ω handles inaccurate failure detection and dynamic transient conditions. It introduces the notion of *logical* connected component and *cancel* messages, in order to maintain a consistent global state in the network.

3.4.1 Example

In the following, we give an example that shows how protocol Ψ may fail in the case of inaccurate failure detection and a need for protocol Ω arises.

Let us consider a network composed of a fully connected component composed of nodes 1,2,3,4, and an isolated node (node 5). Imagine that a subset of the nodes in the connected component (for example nodes 1 and 3), perform a wrong failure detection, suspecting node 4 to have failed. According to protocol Ψ, node 1 sends $cand_{down}(4)$ messages to nodes 2 and 3 and node 3 sends an $accept_{down}(4)$ message to node 1. Suppose then, that node 4 becomes again fully connected together with node 5. At this point the network is composed of a fully connected component of five nodes. However, protocol Ψ prevents a leader to be elected because node 1 still waits for an $accept_{down}(4)$ message from node 2.

3.4.2 Protocol Ω (informal)

Initially the network is composed of a fully connected component and a number of isolated nodes. Every node has its own local view of the connectivity that is

consistent with the views of the other nodes. Possible changes in the local connectivity view include node failures and repairs, as well as link repairs to nodes that have not failed, and link failures to nodes that have previously failed and are in the process of being again fully connected to the network. We assume that nodes in the connected component do not consider link repairs to *logically* connected nodes that entered a *dead* state. Therefore, once a node enters a *dead* state, it must exit the connected component (i.e. the leader election process should converge), before being readmitted to it through a new election.

Node failures and repairs are treated as in protocol Ψ and are recognized as link failures to nodes in the connected component and link repairs to isolated nodes. Link failures to *logically* isolated nodes and link repairs to nodes that are already *logically* connected (and are *alive*), are treated as previous wrong detections and *cancel* messages are issued. Nodes update their state flushing memory regarding received messages, upon receiving *cancel* messages, and flushing memory regarding sent messages, upon sending *cancel* messages.

In terms of the convergence condition, protocol Ω slightly differs from protocols Φ and Ψ, introducing the additional requirement that in order to become a leader, a node should not be connected to nodes with a smaller weight. This requirement becomes necessary in a dynamic environment, where nodes may candidate when they are isolated from nodes with a smaller weight and subsequently regain links to them before convergence is established.

3.4.3 Protocol Ω (formal)

The protocol is formally *fully* described in Fig. 2, using *predicate calculus*. A high level representation of the protocol's behaviour in terms of finite state machines is depicted in Fig. 3. The convergence condition for protocol Ω is expressed below.

Definition 3.2 *The convergence condition for leader election for protocol Ω may be formally expressed as follows:*
$\forall$ *node* η
let $\mathcal{K} = \{i : weight(i) < weight(\eta)\}$
let $\eta_k =$ *number of available links from* η *to nodes* $\in \mathcal{K}$
let $l_j \in \{0, 1\}$ *: state of link from* η *to node* j
let $q =$ *number of cand messages sent by* η
η *outputs leader* $\Leftrightarrow$ *the following condition holds:*

$$(\eta_k = 0) \wedge (q \neq 0) \tag{2}$$
$$\wedge \forall s, j \ \{sent(cand_{dn}(j) to\ s) \Rightarrow [received(accept_{dn}(j)\ from\ s) \vee \neg l_j]$$
$$\wedge\ sent(cand_{up}(j) to\ s) \Rightarrow [received(accept_{up}(j)\ from\ s) \vee \neg l_j]\}$$

In Fig. 2 three predicate calculus sentences must hold at all times. The first sentence regards actions to be taken due to connectivity changes (e.g. seding candidacy or cancel messages). The second sentence regards actions to be taken due to receiving of messages (e.g. sending accept messages, flushing memory of previously received

messages upon receiving cancel messages). The third sentence regards checking the convergence condition for leader election expressed in Def. 3.2.

In Fig. 3 a node state is captured by the state of its links and by a *private* state determined by the total number of its available links and by messages sent and received. Three finite state automata are represented. Two of them describe transitions on links l_j to nodes in the connected component and on links l_i to isolated nodes. The third one describes transitions between the node *private* states. Variables *candS* and *candR* represent the total number of candidacy messages sent and received by a node respectively. Such number varies accordingly to *cancel* messages sent and received. The finite state machines evolve concurrently, starting in a state where the network is composed of a fully connected component and a number of isolated nodes and no messages are sent and received. At each machine state, actions are taken according to the predicate calculus sentences in Fig. 2.

3.4.4 Protocol Ω, Proof of Correctness

By its definition it is easy to see that protocol Ω includes protocol Ψ (see Fig. 2 and Fig. 3), i.e. it is correct in the case of a k-node failure and q-node recovery. In the following, we present a theorem that proves the correctness of protocol Ω in the case of wrong failure detection of k nodes (*k-fake* failure). Wrong failure detection is referred to the case that some nodes that are initially considered to be faulty only by a subset of the nodes in the connected component, then become again fully connected to the network. It is easy to extend the theorem to the case of incorrect *recovery* detection.

Definition 3.3 *A k-fake node failure regards k nodes that are temporarily suspected of being faulty by a subset of the nodes in the connected component. Formally:*

let $\mathcal{H} = \{\ i : \forall j \in \mathcal{H},\ i$ *is connected to* $j\ \}$
let $\mathcal{M} \subset \mathcal{H}$

$$k\text{-fake node failure} \Leftrightarrow \exists T > 0 :$$
$$\begin{cases} \text{at time } t \in [0,T] \text{ } k \text{ nodes} \in \mathcal{H} \text{ are perceived faulty only by some nodes in } \mathcal{M} \\ \text{at time } t > T \text{ they are perceived operational by all nodes in } \mathcal{H} \end{cases}$$

Theorem 3.3 *Given a completely connected network composed of a set* $\mathcal{H}$ *of* n *nodes, for every* $k < n$ *fake node failure, there is no node for which the convergence condition for leader election (Def. 3.2) holds, and the state of the network in terms of messages sent and received by the nodes is eventually the same of the initial one.*

Proof:

The proof of the first part of the theorem follows the one given for Theorem 3.1. Reasoning as in Theorem 3.1, we state that since $\mathcal{M} \subset \mathcal{H}$, at time $t \in$[0,T] there is no node for which the convergence condition in Def. 3.2 holds. Note that the stronger expression for the convergence condition in Def. 3.2 is required this time, in order to state that an elected leader has collected *accept* messages from all nodes it considers operational, regarding all nodes it perceived to be faulty.

let *conn* the initial set of fully connected nodes
let *dead* the initial set of isolated nodes
$\forall$ node η
let η_a = number of available links from η
let $l_j \in \{0,1\}$: state of link from η to node j
let $\mathcal{K} = \{i : weight(i) < weight(\eta)\}$
let η_k = number of available links from η to nodes $\in \mathcal{K}$
let $\mathcal{S} = \{p : weight(p) < weight(s) \wedge p \notin dead\}$
let η_s = number of available links from η to nodes $\in \mathcal{S}$
let msg_s : received by η from node s
$\forall i \; \forall j \; \alpha_j(i) = \beta_j(i) = false$

Forever in time:

1. $\forall j$

$(\neg l_j \wedge j \in conn) \Rightarrow$
$[(\eta_a = 0) \Rightarrow$ enter $(\eta, dead)] \vee \{[(\eta \notin dead) \wedge (\eta_k = 0)] \Rightarrow$
$\forall \; i \; [l_i \wedge \neg sent(cand_{dn}(j)$ to i $) \Rightarrow$ send $(cand_{dn}(j)$ to i $) \wedge$ set $(\beta_j(i), true)]\} \vee$

$(l_j \wedge j \notin conn) \Rightarrow$
$[(\eta \notin dead) \wedge (\eta_k = 0)] \Rightarrow$
$\forall \; i \; [l_i \wedge \neg sent(cand_{up}(j)$ to i $) \Rightarrow$ send $(cand_{up}(j)$ to i) $\wedge$ set $(\alpha_j(i), true)] \vee$

$(l_j \wedge j \in conn) \Rightarrow$
$\forall \; i \; [(\beta_j(i) \wedge l_i) \Rightarrow$ send $(cancel_{dn}(j)$ to $i) \wedge$ set $(\beta_j(i), false)] \vee$

$(\neg l_j \wedge j \notin conn) \Rightarrow$
$\forall \; i \; [(\alpha_j(i) \wedge l_i) \Rightarrow$ send $(cancel_{up}(j)$ to $i) \wedge$ set $(\alpha_j(i), false)]$

2. $\forall msg_s$

$(msg_s = cand_{dn}(j)) \Rightarrow$
$[(\neg l_j \wedge (\eta_s = 0) \wedge l_s \wedge \neg sent(accept_{dn}(j)$ to s$)) \Rightarrow [send(accept_{dn}(j)$ to s$) \wedge$
set $(\beta_j(i), true)] \vee$

$(msg_s = cand_{up}(j)) \Rightarrow$
$[(l_j \wedge (\eta_s = 0) \wedge l_s \wedge \neg sent(accept_{up}(j)$ to s$)) \Rightarrow [send(accept_{up}(j)$ to s$) \wedge$
set $(\alpha_j(i), true)] \vee$

$(msg_s = cancel_{dn}(j)) \Rightarrow$
$[(\exists \; msg_s = cand_{dn}(j) \Rightarrow flush(msg_s)) \wedge (\exists \; msg_s = accept_{dn}(j) \Rightarrow flush(msg_s))] \vee$

$(msg_s = cancel_{up}(j)) \Rightarrow$
$[(\exists \; msg_s = cand_{up}(j) \Rightarrow flush(msg_s)) \wedge (\exists \; msg_s = accept_{up}(j) \Rightarrow flush(msg_s))]$

3. *convergence condition*(Def. 3.2) $\Rightarrow out(leader, \eta)$

Figure 2: Protocol Ω

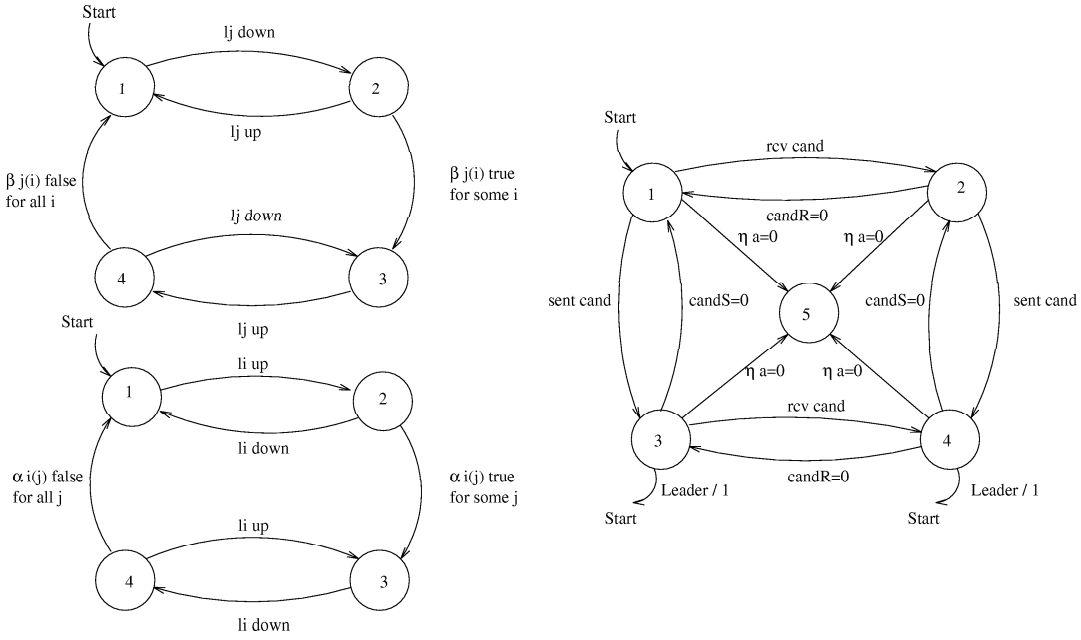

Figure 3: A node executing protocol Ω maintains a finite state automata for every link incident on it and sends messages according to the state of its links and to messages received from other nodes.

If there is no leader elected at time $t \in$[0,T], there cannot be a leader at time $t >$T, because after time T all nodes in $\mathcal{H}$ are perceived to be operational and no more *accept* messages regarding failures are sent.

After time T all nodes in $\mathcal{H}$ share the same connectivity view, which is of a fully connected component of operational processes, identical to the initial one. Therefore, after time T all *cancel* messages have been successfully sent. Hence, after time T no node has memory of sent or received messages regarding failed nodes, otherwise there is at least a node that has not sent cancel messages. This would contradict the fact that after time T the network is fully connected, i.e. all nodes in $\mathcal{H}$ are able to send messages to all other nodes in $\mathcal{H}$. □

4 Applications: the fault recovery project on RAIN

One of the research projects at Caltech is focusing on the construction of highly reliable distributed environments, by leveraging commercially available personal computers, workstations and interconnect technologies.

The Caltech Reliable Array of Independent Nodes (see Fig. 4 for a photo) is a fault tolerant distributed computing environment, built on top of a switched Local Area Network. The system is in continuous evolution, as new features are added.

In particular, the protocol we presented has been implemented on RAIN as part of a *fault recovery* project. Such project presents many innovative aspects [7]:

Figure 4: The RAIN System

- *Distributed checkpointing.* Redundant checkpointing information is distributed on local disks in the network using array codes [2] [3].
- *Redundant connections to switches.* The physical network topology presents features of resistance to partitioning [10].
- *Reliable messaging.* Reliable message exchange is guaranteed by link-monitoring protocols running on the system [11].
- *Fault manager.* The dynamic election of a leader in presence of faults ensures continuous computation.

We have developed a *demo* that shows some of the capabilities we can achieve combining the aspects above. An application that displays a fractal image is run on the system and performs *distributed checkpointing.* Redundancy in the checkpointing information allows reconstruction of the state of the computation from a working subset of all the machines in the system (using an n, k code, we can reconstruct the state when at least k, out of the total n machines, are still working). We are able to disconnect machines from the network, simulating faults. The election protocol designates a *fault manager* that decodes data from some of the remaining machines, recontructs the state of lost applications and performs *fault recovery* assuring that they continue computing on other nodes of the network. Likewise, the disconnected machines enter spontaneously a *dead* state and quit the application. When we reconnect previously disconnected machines a new election is performed and they become

again potential leaders to handle future failures in the group.

5 Conclusions

We have presented a new algorithm for election of a *fault manager* in asynchronous systems. Our algorithm has been implemented as part of a *fault recovery* project and currently runs on the Caltech RAIN system.

We stress out that the algorithm satisfies a *weak liveness* property: it eventually converges only when a connected cluster of processes is found in the system. An infinite sequence of link failures and recoveries may have the protocol run forever, but single or multiple processes crashes cannot force it to halt. This is a very desirable feature in *real world* systems and an important difference with respect to what presented in [15] where a single process crash may halt all processes.

The satisfaction of *weak liveness* is coherent with our *loose* definition of failure detection given in section 2. A failure detection mechanism is usually defined in terms of abstract accuracy and completeness *global* properties [5]. We differ from this approach. We do not use any *global* system properties like the ones introduced in [5] to classify our failure detection mechanism. We define failure detection as a *local* feature of each process in determining the state of its communication links. The link connectivity protocol [11] we assume existing on the system is simple to implement and, ensuring consistency on each link state between every pair of nodes, makes it possible to reason on the network as on a *simple* –one edge per pair of nodes– bidirectional graph. Convergence of our algorithm maps onto a topology property of full connectivity of this graph. In this way, liveness of our algorithm depends from global system state.

We believe that our approach in weakening liveness is an *equivalent* alternative to using formally defined *global* failure detectors. In particular, we conjecture that it is possible to describe our protocol introducing failure detector modules defined in terms of abstract properties as in [5], thus strenghtening the liveness guarantees. While our *loose* definition of *local* failure detectors that operate independently at each node is reflected in the protocol's property of eventual convergence, a definition in terms of abstract *global* properties of the system may transfer liveness into a feature of the failure detection mechanism.

We are currently investigating techniques to reduce the message complexity of the algorithm, as well as the possibility of using our ideas of *delayed* convergence and *local* connectivity monitoring, to solve other problems in distributed computing, formally relating convergence to global system state in terms of network topology properties. Moreover, we are trying to relate algorithms that use weak *liveness* requirements (e.g. algorithms that only eventually converge), to algorithms with strong liveness guarantees based on formally defined *failure detection modules*, since the latter usually define *failure detectors* through *liveness* properties [5]. We believe that the study of this relationship may set a bridge between two different approaches adopted to circumvent well known impossibility results in asynchronous systems subject to failures.

6 Acknowledgments

The authors would like to thank Marc Riedel from Caltech for his useful feedback and the distributed computing group of the Department of Electrical and Computer Engineering of University of Naples for the opportunity to collaborate.

References

[1] K. Birman and R. van Renesse. "Reliable Distributed Computing with the ISIS Toolkit". *IEEE Computer Society Press, 1994.*

[2] M. Blaum, J. Brady, J. Bruck and J. Menon. "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures". *IEEE Transactions on Computers 44-2, February 1995.*

[3] M. Blaum, J. Bruck and A. Vardy. "MDS array codes with independent parity symbols". *IEEE Transactions on Information Theory 42-2, March 1996.*

[4] T. D. Chandra, V. Hadzillacos, S. Toueg and B. Charron-Bost. "On the Impossibility of Group Membership". *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 322-330,May 1996.*

[5] T. D. Chandra and S. Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". *Journal of the ACM 43-2 pp. 225-267, March 1996.*

[6] M. J. Fisher, N. A. Lynch and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". *Journal of the ACM 32-2 pp. 374-382, April 1985.*

[7] M. Franceschetti. "Fault Recovery in a Distributed System". *Laurea Thesis in Computer Engineering, University of Naples, June 1997.*

[8] S. Jajodia and D. Mutchler. "Dynamic Voting for Maintaining the Consistency of a Replicated Database". *ACM Transactions on Database Systems 15-2 pp. 230 - 280, June 1990.*

[9] M.F. Kaashoek and A.S. Tanenbaum. "Group Communication in the Amoeba Distributed Operating System". *Proceedings of the 11th International Conference on Distributed Computing Systems, pp.222-230, May 1991.*

[10] P. LeMahieu, V. Bohossian and J. Bruck. "Fault Tolerant Switched Local Area Networks", *Proceedings of the International Parallel Processing Symposium, Orlando, FL, April 1998.*

[11] P. LeMahieu and J. Bruck. "A Consistent History Link Connectivity Protocol". *Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing, pp. 309-, ACM Press, July 1998.*

[12] E. Y. Lotem, I. Keidar and D. Dolev. "Dynamic Voting for Consistent Primary Components". *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 63-71, August 1997.*

[13] N. Lynch. "Distributed Algorithms". *Morgan Kaufman, 1996.*

[14] L.E. Moser, P.M. Melliar-Smith and V. Agrawala. "Processor Membership in Asynchronous Systems". *IEEE Transactions on Parallel and Distributed Systems 5-5 pp. 459-473,May 1994.*

[15] G. Neiger. "A New Look at Membership Services". *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 331 - 340, May 1996.*

[16] A. M. Ricciardi and K. P. Birman. "Using Process Groups to Implement Failure Detection in Asynchronous Environments". *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 341-352, May 1991.*

[17] G. Singh. "Leader Election in Complete Networks". *SIAM Journal of Computing 26-3 pp. 772-785, June 1997.*

[18] R. van Rennesse, K. Birman, T. von Eicken and K. Marzullo. "New Applications for Group Computing". *Lecture Notes in Computer Science 938 pp . 58-63, 1995.*