A Highly Available Local Leader Election Service

Christof Fetzer, Flaviu Cristian

Abstract— **We define the highly available local leader election problem, a generalization of the leader election problem for partitionable systems. We propose a protocol that solves the problem efficiently and give some performance measurements of our implementation. The local leader election service has been proven useful in the design and implementation of several fail-aware services for partitionable systems.**

Keywords— **Local leader election, partitionable systems, timed asynchronous systems, global leader election.**

I. Introduction

The leader election problem [1] requires that a unique leader be elected from a given set of processes. The problem has been widely studied in the research community [2], [3], [4], [5], [6]. One reason for this wide interest is that many distributed protocols need an election protocol as a sub-protocol. For example, in an atomic broadcast protocol the processes could elect a leader that orders the broadcasts so that all correct processes deliver broadcast messages in the same order. The *highly available leader election problem* was defined in [7] as follows: (S) at any point in time there exists at most one leader, and (T) when there is no leader at time s, then within at most κ time units a new leader is elected.

The highly available leader election service was first defined for synchronous systems in which all correct processes are *connected*, that is, can communicate with each other in a timely manner. Recently, the research in fault-tolerant systems has been investigating asynchronous partitionable systems [8], [9], i.e. distributed systems in which the set of processes can split in disjoint subsets due to network failures or excessive performance failures (i.e. processes or messages are not timely ; see Section III for details). Like many other authors do, we call each such subset a *partition*. For example, processes that run in different LANs can become partitioned when the bridge or the network that connects the LANs fails or is "too slow" (see Figure 4). One reason for the research in partitionable systems is that the "primary partition" approaches [10] allow only the processes in one partition to make progress. To increase the availability of services, one often wants services to make progress in all partitions.

Our recent design of a membership [11] and a clock synchronization service for partitionable systems [12] has indicated that we need a leader election service with different properties for partitionable systems than for synchronous systems. The first problem that we encountered is how to specify the requirements of such a *local leader election service*. Ideally, such a service should elect exactly one local leader in each partition. However, it is not always possible to elect a leader in each partition. For example, when the processes in a partition suffer excessive performance failures, one cannot enforce that there exists exactly one local leader in that partition. To approach this problem, we have to define in what partitions local leaders have to be elected: we introduce therefore the notion of a *stable partition*. Informally, all processes in a stable partition are connected to each other, i.e. any two processes in a stable partition can communicate with each other in a timely manner. The processes in a stable partition are required to elect a local leader within a bounded amount of time. An election service might be able to elect a local leader in an *unstable partition*, i.e. a partition that is not stable, but it is not guaranteed that there will be a local leader in each unstable partition. We call a process "unstable" when it is part of an unstable partition.

In each stable partition, a local leader election service has to elect exactly one local leader. In an unstable partition the service might not be able to elect exactly one local leader. It can be advantageous to split an unstable partition into two or more "logical partitions" with one local leader each if that enables the processes in each of these logical partitions to communicate with each other in a timely manner (see Figure 1). To explain this, note that our definition of a "stable partition" will require that all processes in such a partition be connected to each other. This implies that when the connected relation in a partition is not transitive, that partition is unstable. For example, the connected relation can become non-transitive for three processes $\{p, q, r\}$ if the network link between p and r fails or is overloaded while the links between p and q and q and r stay correct (see Figure 2).

In specific circumstances, our local leader service splits an unstable partition into two or more logical partitions with one leader in each. The service makes sure that a timely communication between any two processes in a logical partition is possible. However, sometimes this communication has to go via the local leader in case two processes p and r in a logical partition are only connected through the local leader q (see Figure 2.b). Informally, a logical partition created by our local leader service is a set of processes such that the local leader of this logical partition can communicate in a timely fashion with all processes in the logical partition.

The scenario depicted by Figure 1 can be one such situation, where two logical partitions with one leader in each are created. However, when logical partitions are created,

Department of Computer Science & Engineering, University of California, San Diego, La Jolla, CA 92093−0114. e-mail: cfetzer@cs.ucsd.edu, flaviu@cs.ucsd.edu, http://www.cs.ucsd.edu/~cfetzer. This research was supported by grants F49620-93 and F49620-96 from the Air Force Office of Scientific Research. An earlier version of this paper appeared in the Proceedings of the Sixth IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-6). For more information see http://www.cs.ucsd.edu/~cfetzer/HALL.

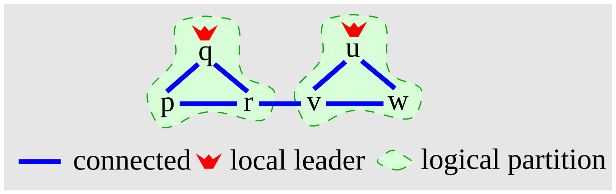

Fig. 1. A local leader election service is permitted to split an unstable partition $\{p,q,r,u,v,w\}$ into two logical partitions $\{p,q,r\}$ and $\{u,v,w\}$.

it is not done trivially; in particular, we prohibit that case where a local leader service simply elects all unstable processes as local leaders. For example, in an "unstable" trio that consists of two processes p and r that are connected to a third process q but are not connected to each other, only one of the three processes is permitted to become local leader (see Figure 2). Note that we do not want to have two local leaders p and r even when these two processes are only indirectly connected through q (see Figure 2.g). The intuition behind this restriction is that the election of a local leader l has to be "supported" by all processes connected to l. Since p and q cannot both get the support of p, at most one of the two processes is allowed to become leader.

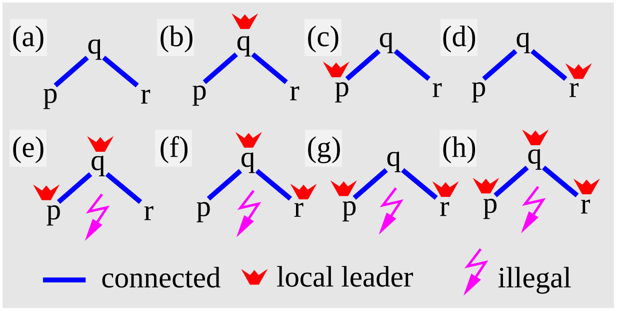

Fig. 2. The trio p, q, r is unstable since p and r are not connected. A local leader election service must elect at most one process in the trio as local leader.

We derive in this paper a formal specification for the highly available local leader election service. The specification implies that a local leader service creates logical partitions so that (1) logical partitions never overlap, (2) stable partitions are subsets of logical partitions, (3) two local leaders are always in two separate logical partitions, and (4) logical partitions are such that processes in one partition are not connected to the local leader of any other logical partition.

In this paper we propose an efficient protocol that implements a highly available local leader election service. We use this protocol in our fail-aware group membership service [11] and fail-aware clock synchronization service for partitionable systems [11]. We give performance measurements of our implementation on a network of workstations.

II. Related Work

There are many publications about solutions to the leader election problem for synchronous and asynchronous systems [1], [13], [2], [3], [4], [5], [6]. The election problem was first defined and solved by [1]. Many election algorithms are based on the "message extinction" principle that was first introduced by [13]: a process r rejects a message that requests that a process q should become leader whenever r knows of a process p that wants to become leader and p has a lower id than q. Many papers about leader election do not address the masking of failures during an election or the election of a new leader when the previous one fails. We are not aware of any other specification for a local leader election service for partitionable systems.

Our implementation of an election service contains some novel aspects. First, instead of using message extinction, we use an independent assessment protocol [14], [11] that approximates the set of processes in a stable partition. Typically, this ensures that only one process l in a stable partition requests to become leader and all other processes in l's stable partition support l's election. A local leader has to renew its leadership in a round-based fashion. This ensures that the crash of a local leader in a stable partition results in its replacement within a bounded amount of time. In a stable partition of N processes the protocol sends one broadcast message and $N-1$ unicast messages per round. Second, we use communication by time to ensure that logical partitions never overlap: we use a mechanism similar to that of a lease [15] to make sure that a processes is at any point in time in at most one logical partition.

A protocol option allows us to use the same protocol to elect either local leaders or one global leader: the protocol can be forced to create only logical partitions that contain a majority of the processes. Since logical partitions never overlap, at any time there can exist at most one majority partition and thus, at most one global leader in the system. While a local leader can be used to maintain consistency amongst the processes in a partition, a global leader can be used to maintain consistency amongst all processes. For example, a local leader can be used to ensure mutual exclusion between the processes in one partition while a global leader can be used to ensure mutual exclusion between all processes.

Some group membership services for partitionable systems [16], [17], [18] can be used to elect local leaders. For example, the strong membership protocol of [16] or the three round protocol of [18] can be used to elect local leaders such that each local leader is in a disjoint partition. However, a local leader service can be said to be more "basic" than a membership service in the sense that (1) a membership service for partitionable systems typically elects local leaders that create new groups (e.g. see [16]), and (2) an implementation of a local leader service does not need the stronger properties provided by a group membership service such as an agreement on the history of groups.

III. Timed Asynchronous System Model

The *timed asynchronous system model* [19] is an abstraction of the properties of most distributed systems encountered in practice, built out of a set of workstations connected by a LAN or WAN. The timed model makes very few assumptions about a system and hence, almost all practical distributed systems can be described as timed asynchronous systems. Since it makes such weak assumptions, any solution to a problem in the timed model can be used to solve the same problem in a practical distributed system. The timed model is however sufficiently strong to solve many practically relevant problems, such as clock synchronization, highly available leadership, membership, atomic broadcast and availability management [19].

The timed model describes a distributed system as a finite set of processes $\mathcal{P}$ linked by an asynchronous datagram service. The datagram service provides primitives to transmit unicast and broadcast messages. A one-way time-out delay δ is defined for the transmission delays of messages: although there is no guarantee that a message will be delivered within δ time units, this one-way timeout is chosen so as to make the *likelihood* of a message being delivered within δ timeouts suitably high [20]. We say that a process receives a message m in a *timely manner* iff the transmission delay of m is at most δ. When the transmission delay of m is greater than δ, we say that m has suffered a performance failure or that m is *late* [20].

We assume that there exists a constant δ_{min} that denotes the minimum message transmission delay: any message sent between two *remote* processes has a transmission delay of at least δ_{min} time units. By "remote" we mean that the message is sent via a network.

The asynchronous datagram service has an omission/performance failure semantics [20]: it can drop a message or it can fail to deliver a message in a timely manner, but the probability that it delivers corrupted messages is negligible. Broadcast messages allow asymmetric performance/omission failures: a process might receive a broadcast message m in a timely manner, while another process might receive m late or not at all.

The *asynchronous datagram service* satisfies the following requirements:

- *Validity*: when a process p receives a message m from q at some time t, then indeed there exists some earlier time $s < t$ such that q sent m to p at s.
- *No-duplication*: a process receives a message m at most once, i.e. when message m is delivered to process q at time s, then there exists no other time $t \neq s$ such that the datagram service delivers m to q at t too.

The process management service defines a scheduling time-out delay σ, meaning that a process is likely to react to any trigger event within σ time units (see [19]). If p takes more than σ time units to react to a trigger event, it suffers a performance failure. We say that p is *timely* in an interval $[s,t]$ iff at no point in $[s,t]$ p is crashed and p does not suffer any performance failure in $[s,t]$. We assume that processes have crash/performance failure semantics [20]: they can only suffer crash and performance failures. Processes can recover from crashes.

Two processes are said to be *connected* [18] in $[s,t]$ iff they are timely in $[s,t]$ and each message sent between them in $[s,t-\delta]$ is delivered in a timely manner (see Figure 3). We denote that p and q are connected in $[s,t]$ by using the predicate *connected(p,q,s,t)*.

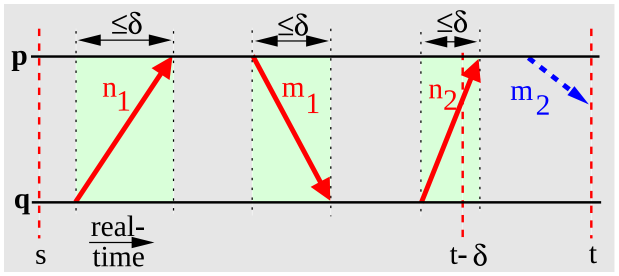

Fig. 3. Timely processes p, q are connected in $[s,t]$ iff all messages sent between them in $[s,t-\delta]$ are delivered within δ time units.

Processes have access to local hardware clocks with a bounded drift rate. Correct hardware clocks display strictly monotonically increasing values. We denote the local hardware clock of a process p by H_p. For simplicity, we assume in this paper that we can neglect the granularity of a hardware clock: e.g. a clock has a resolution of $1\mu s$ or smaller. Hardware clocks are proceeding within a linear envelope of real-time: the drift rate of a correct hardware clock H_p is bounded by an a priori given constant ρ so that, for any interval $[s,t]$:

$$(t-s)(1-\rho) \leq H_p(t) - H_p(s) \leq (t-s)(1+\rho).$$

An important assumption is that the hardware clock of any non-crashed process is correct. Informally, we require that we can neglect the probability that the drift rate of a hardware clock of a non-crashed is not within $[-\rho,\rho]$. Whether some failure probability is negligible depends on the stochastic requirements of an application [20], [21]. For non-critical applications, the use of a simple counter connected to a quartz oscillator and an appropriately chosen ρ provide a sufficiently close approximation of a crash failure semantics, i.e. one can neglect the probability that any clock failure except clock crash failures occur. For safety critical applications, such an implementation might not be sufficient. However, one can use multiple oscillators and counters to make sure that the probability of any clock failure except a clock crash failure becomes negligible [22].

For simplicity, we assume that a hardware clock does not recover from a crash. Hardware clocks do not have to be synchronized: the deviation $H_p(t) - H_q(t)$ between correct hardware clocks H_p and H_q is *not* assumed to be bounded.

For most quartz clocks available in modern computers, the maximum hardware clock drift rate ρ is in the order of 10^{-4} to 10^{-6}. Since ρ is such a small quantity, in what follows we neglect terms in the order of ρ^2 or higher. In particular, we will equate $(1+\rho)^{-1} \approx (1-\rho)$ and $(1-\rho)^{-1} \approx (1+\rho)$. When a process measures the length of an interval $[s,t]$ by $T - S \triangleq H_p(t) - H_p(s)$, the error of this measurement is within $[-\rho(T-S), +\rho(T-S)]$ since

$$(T-S)(1-\rho) \leq t-s \leq (T-S)(1+\rho).$$

IV. Communication Partitions

The timeliness requirement of our specification of the local leader problem will be based on the notion of a *stable partition*. However, there are many possible and reason-able definitions of a *stable partition* because one can put different constraints on the kind of communication possible between the processes in two stable partitions. The strongest definition would require that no communication be possible between two stable partitions (see the definition of the predicate *stable* in [18]) while the weakest definition would not place any constraints on the communication between partitions. The timeliness requirement of the local leader problem will demand that a local leader be elected in any stable partition within a bounded amount of time. Therefore, the weaker the definition of a "stable partition" is, the stronger will be the timeliness requirement since a protocol is required to elect a leader under harder conditions. In this paper , we use a formalization of the notion of a "stable partition" which is in between the above two extremes: a Δ-partition (see below). The protocol we propose is based on Δ-partitions.

The definition of the local leader problem is largely independent of the actual definition of a stable partition. However, we assume that all processes in a stable partition are connected. We introduce a generic predicate *stablePartition* that denotes the definition that is assumed by an implementation of a local leader service: *stablePartition(SP,s,t)* is true iff the set of processes SP is a stable partition in interval $[s,t]$.

In this section, we introduce one possible definition of a stable partition. Let us motivate it by two LANs connected by a network (see Figure 4): the processes that run in one LAN can communicate with the processes in the other LAN via a network. When the network provides a fast communication between the two LANs, we want to have one leader for both LANs. Since the network can become overloaded, processes in the two LANs can become logically disconnected in the sense that the communication between them is too slow for having only one leader for the two LANs. In that case, we want to have a local leader in each of the two LANs.

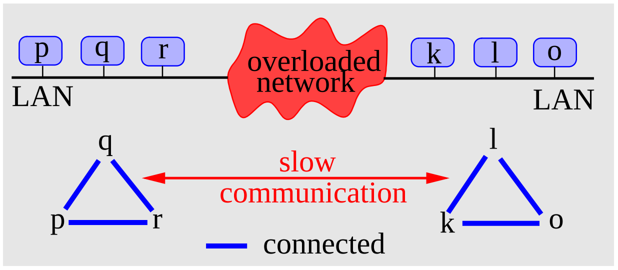

Fig. 4. Two LANs are connected by a network. The processes in the two LANs become partitioned when the network fails or is "too slow".

Two processes p and q are *disconnected* [18] in an interval $[s,t]$ iff no message sent between these two processes arrives during $[s,t]$ at its destination. In this paper , we introduce a weaker predicate that we call "Δ-disconnected". The intuition behind this predicate is that we can use a fail-aware datagram service [23] to classify messages as either "fast" or "slow". The service calculates an upper bound on the transmission delay of each message it delivers. If this bound is greater than some given Δ, the message is classified as "slow" and otherwise, the message is classified as "fast". Constant Δ is chosen such that the calculated upper bound for messages sent between two connected processes, i.e. these messages have a transmission delay of at most δ, is at most Δ. One has to choose $\Delta > \delta$ since one can only determine an upper bound and not the exact transmission delay of a message.

To be able to calculate an upper bound on the transmission delay of each message it delivers, the fail-aware datagram service [23] maintains for each process q an array TS such that TS contains for each process p the receive and send time stamps of some message n that q has received from p (see [23] for details). The fail-aware datagram service piggy-backs on each unicast message m it sends from q to p the time stamps of n and the send stamp of m. The computation of the upper bound for m uses the time stamps of the round-trip (n,m): the transmission delay of m is not greater than the duration between p sending n and p receiving m since q had received n before it sent m. One can use several techniques to improve this upper bound such that it becomes close to the real-transmission delay of m. The upper bound calculation is similar to the reading error computation in probabilistic clock reading [24].

A process p is Δ-*disconnected* from a process q in a given time interval $[s,t]$ iff all messages that p receives in $[s,t]$ from q have a transmission delay of more than Δ time units (see Figure 5). We use the predicate Δ-*disconnected(p,q,s,t)* to denote that p is Δ-disconnected from q in $[s,t]$.

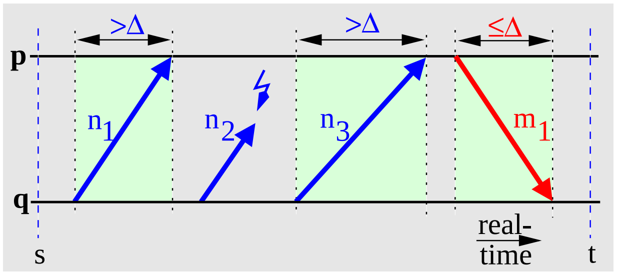

Fig. 5. Process p is Δ-disconnected from q in $[s,t]$ because all messages that p receives from q in $[s,t]$ have a transmission delay of more than Δ. Note that q might receive messages from p with a delay of less than Δ.

We say that a non-empty set of processes SP is a Δ-*partition* in an interval $[s,t]$ iff all processes in SP are mutually connected in $[s,t]$ and the processes in SP are Δ-disconnected from all other processes (see Figure 6):

$$\begin{aligned}&\Delta\text{-}partition(SP,\ s,\ t) \triangleq SP \neq \emptyset\\&\quad \wedge \forall p,q \in SP : connected(p,q,s,t)\\&\quad \wedge \forall p \in SP, \forall r \in \mathcal{P} - SP\text{: } \Delta\text{-}disconnected(p,r,s,t).\end{aligned}$$

V. Specification

In this section, we derive a formal specification for the local leader problem. The main goal of a local leader service is to elect one local leader per stable partition. However, the specification has also to constrain the behavior of

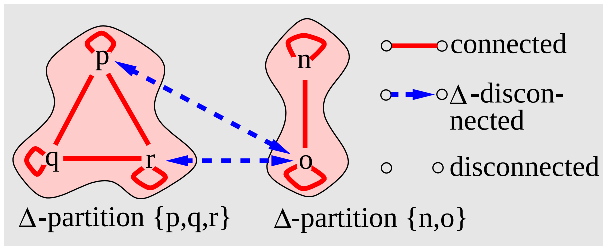

Fig. 6. All processes in a Δ-partition can communicate with each other in a timely manner. All messages from outside the partition have a transmission delay of more than Δ.

processes that are not in stable partitions. Our approach to this problem is that we require that a local leader service creates logical partitions such that (1) in each logical partition there is at most one leader, and (2) each stable partition is included in a logical partition.

The local leader election problem is defined using three predicates and one constant all of which have to be instantiated by each implementation of a local leader service: *stablePartition, Leader, supports* and constant κ. The predicate Δ-partition(SP,s,t) defined in Section IV is one possible definition of a *stablePartition*. The predicate $Leader_p(t)$ is true iff process p is a local leader at time t. Our specification is based on the idea that a process p has to collect some support (e.g. votes) before p can become leader. A vote of a process q for process p can have a restricted lifetime, i.e. q can say that its vote for p is only valid for a certain amount of time. Our specification of the local leader problem is independent of the actual way a protocol implements the voting. We achieve this by introducing the notion of a process q *supporting* some process p. By "q supports p at time t" we mean that q has voted for p's election as a local leader and this vote is still valid at time t. Formally, this is expressed by the predicate $supports^t(p,q)$, defined to be true iff p supports q's election as local leader at time t.

We will require that at any point in time a process support *at most* one process and that a local leader l be supported by all processes that are connected to l. In particular, a leader p in a stable partition SP must be supported by all processes in SP, since all processes in a stable partition are by definition connected. We will define the predicates *Leader* and *supports* associated with the proposed local leader election protocol in Section IX.

The specification of the local leader problem consists of four requirements: (T, SO, BI, LS). The timeliness requirement (T) requires that in any stable partition a local leader be elected after at most κ time units (see Figure 7). To allow a local leader service to implement a rotating leader schema (see Figure 8), after a first local leader is elected in a stable partition, we do not require that this process stays a local leader as long as the partition remains stable. Instead, we require that in any interval of length κ in which a set of processes SP is a stable partition, there exist at least one point in time at which there exists a local leader in SP.

The timeliness requirement (T) can formally be expressed as follows.

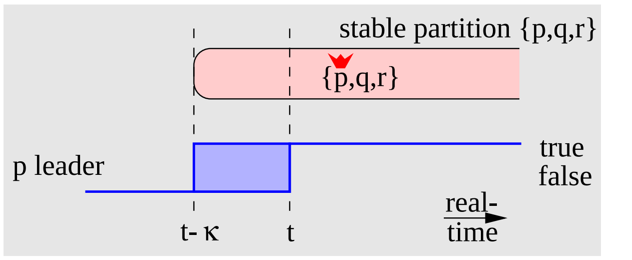

Fig. 7. When a stable partition $\{p, q, r\}$ forms at time $t - \kappa$, it is guaranteed that a local leader p is elected within κ time units.

(T) When a set of processes SP is a stable partition in an interval $[t-\kappa, t]$*, then there exists a process* $p \in SP$ *and a time* $s \in [t-\kappa, t]$ *so that p is local leader at time s:*

$$\forall SP \subseteq \mathcal{P}, \forall t : stablePartition(SP, t-\kappa, t)$$
$$\Rightarrow \exists p \in SP, \exists s \in [t-\kappa, t] : Leader_p(s).$$

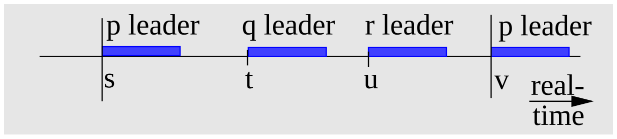

Fig. 8. A rotating leader schema transfers the leadership periodically between the processes. This schema has been proven useful in the implementation of atomic broadcast services.

We require that at any point in time a process r support at most one process. Formally, we state the "Support at most One" requirement as follows:

(SO) For any time t, if a process r would support a process p at t and a process q at t, then p and q are the same process:

$$\forall t, \forall p, q, r \in \mathcal{P} : supports^t(r,p) \land supports^t(r,q) \Rightarrow p = q.$$

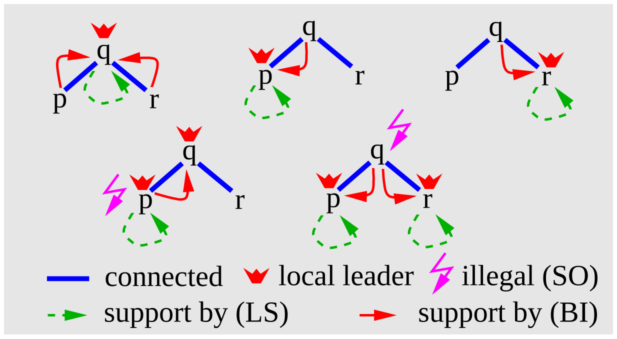

Fig. 9. Requirement (SO) requires that a process support at most one process at a time and (LS) that a leader supports itself. Thus, in the trio $\{p, q, r\}$ there can exist at most one leader whenever (SO,LS,BI) is satisfied.

We already mentioned that when there is a trio of three processes p, q, and r such that p, q and q, r are connected (see Figure 2), we want that at most one of the three processes be a local leader. We therefore introduce two more requirements: the "leader self support" requirement (LS) and the "bounded inconsistency" requirement (BI). Requirement (BI) requires that when a local leader p is connected to a process q for at least κ time units, q must support p. In other terms, a process can be connected to two leaders for at most κ time units (bounded inconsistency). For example, when two stable partitions merge into a new stable partition, after κ time units there is at most one local leader in the new stable partition.

(BI) When a process p is local leader at time t and p has been connected for at least κ time units to a process q, then q supports p at t:

$$\forall p, \forall q, \forall t\colon connected(p, q, t\text{-}\kappa, t) \wedge Leader_p(t) \Rightarrow supports^t(q, p).$$

A timely process is always connected to itself. Hence, requirement (BI) implies that a timely local leader has to support itself within κ time units of becoming local leader. We strengthen this special case by requiring that any local leader have always to support itself; in particular, a local leader l has to support itself as soon as it becomes local leader and even when l is slow. We show in Section VI how requirements (LS) and (SO) ensure that there is at most one local leader in each logical partition.

(LS) A local leader always supports itself:

$$\forall p, \forall t : Leader_p(t) \Rightarrow supports^t(p, p).$$

Let us explain why the three requirements (SO,LS,BI) imply that in a trio $\{p, q, r\}$ with p,q and q, r are connected for at least κ time units, there must be at most one leader (see Figure 9). If p and q were leaders at the same time, p would have to support itself (LS) and p would have to support the local leader q because p and q are connected (BI). However, p is only allowed to support one process at a time (SO). Thus, p and q cannot be leaders at the same point in time t. If p and r would be leaders at the same time, q would be required to support p and r (BI). This would again violate requirement (SO). Therefore, at most one process in $\{p, q, r\}$ can be leader.

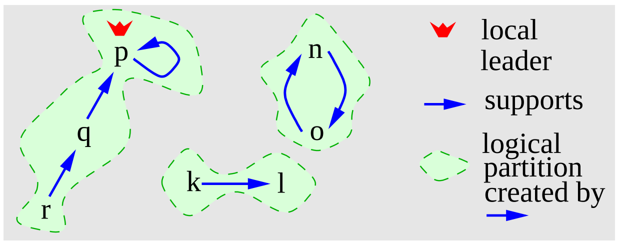

Fig. 10. The supports predicate partitions the set of processes.

VI. Logical Partitions

We now show how the supports predicate creates logical partitions and that each of these partitions contains at most one leader. Furthermore, each leader in a stable partition SP is in a logical partition LP that contains SP, i.e. $SP \subseteq LP$. Intuitively, a logical partition LP that contains a process p contains each process q for which there exists a finite, *undirected* path in the supports-graph between p and q (see Figure 10). By *undirected* we mean that the path ignores the "direction" of the *supports* predicate.

Formally, we define logical partitions with the relation $SUPPORT^t$ that is the reflexive, symmetric, and transitive closure of $supports^t$ (see Figure 11). By definition, $SUPPORT^t$ is an equivalence relation that partitions the set of processes in completely connected subgraphs. We say that two processes p and q are in the same logical partition at time t iff $SUPPORT^t(p, q)$ is true. Since $SUPPORT^t$ is reflexive, each process is in a logical partition. Two logical partitions LP_1 and LP_2 are either non overlapping or they are equal because of the symmetry and transitivity of $SUPPORT^t$.

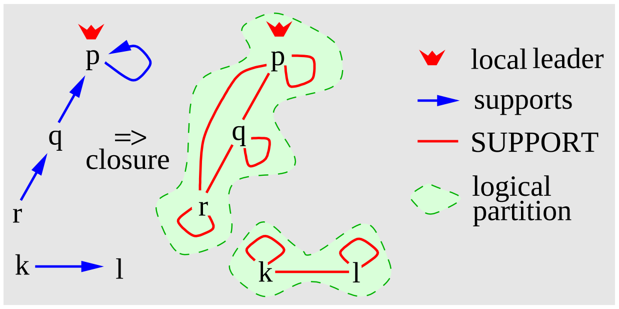

Fig. 11. The $SUPPORT^t$-relation is the reflexive, symmetric, and transitive closure of supports^t. The closure creates fully connected subgraphs that are isolated from each other.

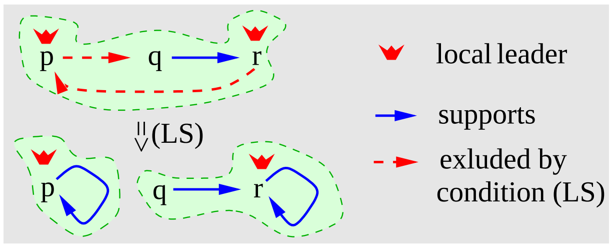

Fig. 12. By requiring that a leader supports itself, we guarantee that there is no undirected path in the supports-graph that contains more than one local leader.

Let us show that in each logical partition LP there is at most one local leader. The intuition is that by requiring that a local leader supports itself, we split any path or cycle with two local leaders (that could exist in a supports-graph that does not satisfy (LS)) into two paths with one local leader each (see Figure 12). More precisely, we can prove by contradiction that there exists at most one leader per logical partition. To do so, let us assume that there would exist a time t and a logical partition LP that contains two local leaders p and q at t (see Figure 13). Since $p, q \in LP$, by definition $SUPPORT^t(p, q)$ holds. Therefore, there has to exist a finite, undirected path UP in the supports-graph between p and q. Since at any time a process supports at most one process (SO) and a leader has to support itself (LS), p and q are at the two ends of UP such that there exist two processes k and l in UP that support p and q, respectively. Processes k and l have to be supported themselves by two other processes because of (SO). This argument can be applied recursively to show that the path UP would have to have an infinite length. Hence, there cannot exist a finite path between two leaders and therefore a logical partition contains at most one local leader.

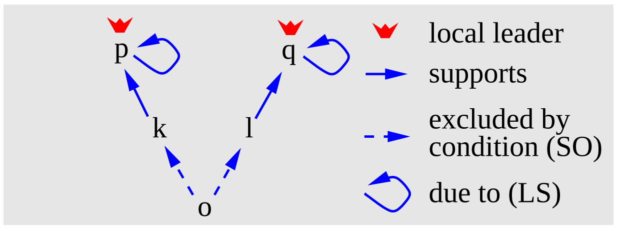

Fig. 13. When two leaders p and q would be in the same logical partition there would exist a finite undirected path between p and q. Since p and q have to be at the two ends of that path by (LS) and (SO), there would have to exist a process o that supports two processes.

Since all processes in a stable partition are mutually connected, (BI) implies that within κ time units after a stable partition SP has formed, any local leader in SP is supported by all processes in SP (see Figure 14). Thus, after κ time units there is at most one leader per stable partition because at any point in time a process supports at most one process (SO).

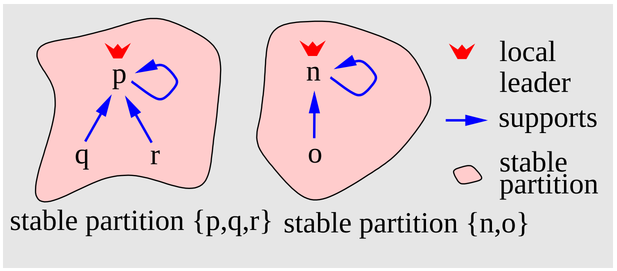

Fig. 14. A leader in a stable partition that has formed at least κ time units ago is supported by all processes in its partition.

When there exists a stable partition $SP = \{p, q, r\}$ that has formed no later than time $t - \kappa$ and p is local leader in SP at time t, all processes in SP support p (see Figure 15). Hence, between any two processes u, $v \in SP$ there exists an undirected path (u, p, v). This implies that u and v are in the same logical partition. Note that a logical partition LP can be a strict superset of a stable partition SP: it is allowed for a process n outside of SP to support a process in SP and hence, n can be in the logical partition LP that contains SP.

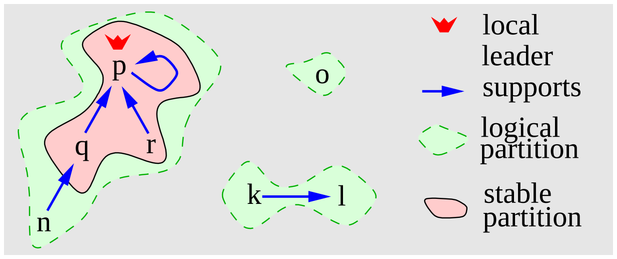

Fig. 15. The logical partition $\{p, q, r, n\}$ is a strict superset of the stable partition $\{p, q, r\}$.

When a stable partition SP forms at time $t-\kappa$, then for κ time units there could exist more than one local leader because leaders from previous partitions have to be demoted first (see Figure 16). Note that even though there might exist more than one local leader in a stable partition for a bounded amount of time, each of these local leaders is in a logical partition with no other local leader.

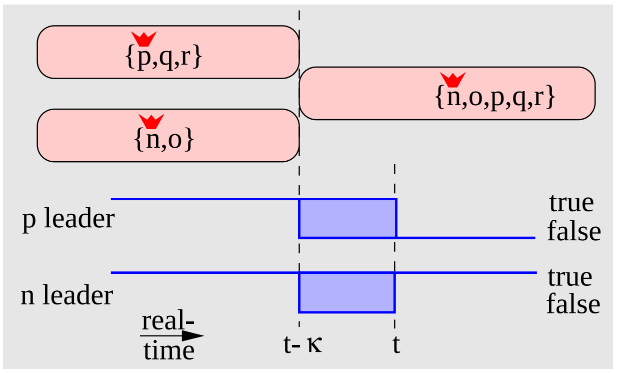

Fig. 16. When two stable partitions merge into a new partition, there might exist multiple local leaders in the same stable partition. The duration of such an inconsistent behavior is however bounded by κ.

Requirement BI demands that a local leader p be supported by all processes that are connected to p for at least for κ time units. This requirement ensures that two parallel local leaders p and q are only elected when there is some good reason for that (see Figure 17): the supporters of p cannot be connected to q (for longer than κ) and the supporters of q are not connected to p (for longer than κ).

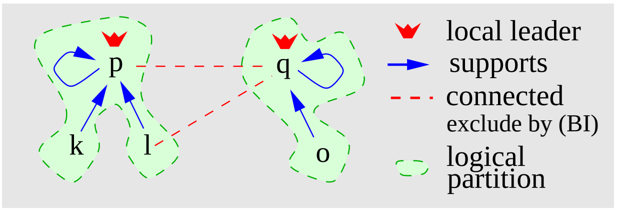

Fig. 17. Requirement (BI) prohibits that a supporter of a local leader p has been connected to another local leader q for more than κ time units.

VII. Protocol Overview

The idea of the proposed protocol for a local leader election service is the following. The processes send all messages with a fail-aware datagram service [23] that classifies all messages it delivers as either “fast” or “slow”. Each process p uses an independent assessment protocol [11] to approximate the set of processes in its Δ-partition by a set that we call $aliveSet_p$: this set contains the processes from which p has “recently” received a fast message. This independent assessment protocol does not send any messages. It uses the messages sent by other services like the local leader election service. To update the alive-sets, it stores for each process pair p and q the receive time stamp of the most recent fast message that q has received from p.

A process p that has an id smaller than the ids of all other processes in its alive-set broadcasts by sending periodic "Election"-messages to request that the other processes support its election as local leader. A process q only supports the election requests of a process p if p is the process with the smallest id in q's $aliveSet_q$. When a process q is in a Δ-partition SP for a "sufficiently long" time, its alive-set contains at most processes from SP because all messages it receives from processes outside of SP are slow and hence, are not sufficient to keep these processes in q's alive-set. In particular, the process $p = min(SP)$ will start broadcasting election messages since p's id will be smaller than the id of any other process in its alive-set. Since each process $q \in SP$ is connected to p, q receives p's election messages as "fast" messages and includes p in its alive-set. Process p's id is smaller than any other id in q's alive-set, i.e.

$$min(aliveSet_q) = min(SP) = p,$$

because q's alive-set contains at most processes from SP and it does contain p due to p's periodic election messages. Thus, all processes in SP implicitly agree to support the election of p and they all reply to p's election messages. Thus, p includes all processes in SP in its alive-set, i.e.

$$aliveSet_p = SP.$$

When p gets fast "supportive replies" by all processes in its $aliveSet_p$, it becomes leader. Since all processes in SP support p's election, process p succeeds in becoming leader.

When a new Δ-partition SP forms at time t, all processes in SP have to support the same process $p = min(SP)$ within κ time units. A process q supports the election of any process only for a bounded amount of time: this enables q to support another process within a bounded amount of time. For example, when q supports a process r at t, it will support another process p only after its support for r expires. When a process r becomes leader with the support of q, we have to ensure that r is demoted before the support by q expires. The protocol achieves this timely demotion even when r is slow: r is only leader as long as its hardware clock shows a value smaller than some value *expirationTime* that is calculated during r's election. Note that q can support a different process p after its support for r expired without any further exchange of messages between q and r. This is an important property because q does not have to learn that r has been demoted or has crashed (which would be impossible to decide in an asynchronous system!) before it supports a different process.

Our protocol does not require the connected relation to be transitive for the successful election of a local leader (see Section X). The *aliveSet* of a process p approximates the set of processes connected to p. If process p gets the support of all processes in its *aliveSet*, p can become leader even when no two other processes in its *aliveSet* are connected.

VIII. Protocol

The pseudo-code of the protocol for a local leader election service is given in Figures 30 and 31. All messages are sent and delivered by a fail-aware datagram service [23]. This allows a receiver of a message m to detect when the transmission delay of m was "fast" or "slow", i.e. at most Δ time units or more than Δ time units, respectively. The fail-aware datagram service provides a primitive to send unicast messages (denoted by *fa-Unicast*) and one to broadcast messages (denoted by *fa-Broadcast*). To deliver a message m at clock time $recTime$ that was sent by process p to q, the service generates an event denoted by *fa-Deliver(m,p,fast,recTime)* at the destination process q. The boolean flag *fast* is set to true when m was fast, otherwise, it is false.

Each process p maintains a set of processes (called $aliveSet_p$) from which p has received a "fast" message in the last, say, *expires* clock time units. We will determine the value of the constant *expires* later on. The alive-set of a process is maintained by two procedures: *UpdateAliveSet* and *PurgeAliveSet*. The procedure *UpdateAliveSet* inserts the sender of a fast message m into *aliveSet* and stores the receive time stamp of m in an array called *lastMsg*. The procedure *PurgeAliveSet* uses array *lastMsg* to remove a process r from $aliveSet_{myid}$ of the executing process $myid$, if process $myid$ has not received a fast message from r for more than *expires* clock time units.

Let us assume that the ids of processes are totally ordered. Function *PurgeAliveSet* also returns the first point in clock time (given with respect to the local hardware clock) when $myid$ could be smaller than the id of any other process in *aliveSet*. When the process does not receive any more messages from processes with smaller ids, the returned bound is tight. The calculated bound is used to optimize the broadcasts of Election-messages: a process p starts broadcasting periodic Election-messages as soon as its id becomes smaller than the id of any other process in its alive-set. When p is timely, it broadcasts at least every EP clock time units an Election-message, where EP (election period) is an a priori given constant. The scheduling of the broadcasts is performed with the help of an *alarm clock* that we call the $aliveAC$. A process p can set an alarm clock with the method *SetAlarm*: *SetAlarm(T)* requests that a "WakeUp" event be generated just after p's hardware clock shows value T. A timely process p is awakened within σ time units of its hardware clock H_p showing T. The scheduling timeout delay σ is an a priori defined constant. A process can cancel an alarm by operation "Cancel".

An Election-message sent by process p contains

- a unique time stamp (denoted *request*) to detect replies that correspond to this request of p to become leader, and
- the alive-set of p at the time when p's hardware clock showed value *request*.

When a process p sends an Election-message, it stores the *request* time stamp in a variable called *lastRequest* to be able to identify replies for the message. Process p also stores its current alive-set in a variable *targetSet* and resets the variable *replySet* to the empty set: p will insert into its *replySet* the ids of all processes from which p gets a fast reply that supports p's election.

A process q replies to all fast Election-messages and ignores all slow messages. A "Reply"-message m identifies the *request* that q is replying to and also contains a flag that indicates if q is supporting p's election. When the flag is true, we say that m is a "supportive" reply and by sending m, q guarantees not to send another supportive reply in the next *lockTime* clock time units. Process q achieves this guarantee by storing p's id and the *request* id in a variable *LockedTo* and the sum of the receive time of m plus *lockTime* in a variable *LockedUntil*. When other Election-messages arrive, q can use the variables *LockedTo* and *LockedUntil* to determine if it is still supporting p (i.e. q is still "locked" to p).

We will determine the value of the constant *lockTime* in Section IX. We say that a process q "is locked to process p (at time t)" to denote that q cannot send a supportive reply to any other process than p (at t). When q sends a supportive reply to p (at t), we say that "q locks to p (at t)".

Let us consider that process q receives an Election-message m from p containing the time stamp *request* and p's alive-set (denoted by *alive*). Process q only locks to p if

- m is a fast message: q has to ignore all election requests from outside of its Δ-partition,
- q is not locked to any other process: this ensures that at any point in time a process is locked to at most one process,
- p's id is smaller than q's id, and
- p is the process with the minimum id in q's alive-set: when two processes r and q are in the same Δ-partition SP, they implicitly agree on which process to support since – as we explain below – the following condition holds:

$$min(aliveSet_r)=min(aliveSet_q)=min(SP).$$

When process p receives a fast supportive reply from a process q to its last Election-message, p inserts q into its *replySet*. After 2Δ real-time units (which could be up to $2\Delta(1+\rho)$ clock time units due to the drift of p's hardware clock) p checks if it has become leader or has renewed its leadership (by calling function *CheckIfLeader*). In case p has already been leader, p tests if it has been able to renew its leadership as soon as p has received replies from all processes in its *targetSet*, i.e. as soon as *replySet*=*targetSet*. A process p only becomes leader (see function *CheckIfLeader*) iff

- p has been in its own alive-set at the time p has sent the Election-message (see proof of requirement (BI) in Section IX),
- p has received a fast supportive reply from all processes in its current alive-set: this is necessary to make sure that p is supported by all processes that p is connected to, and
- p has the minimum id in $replySet_p$: this makes sure that p supports itself.

When p becomes leader, it sets its local variable *imLeader* to true and calls a procedure *newLeader* to notify its client that it is leader until p's hardware clock shows some clock value *expirationTime* (to be determined in Section IX). Process p schedules the broadcast of its next Election-message at time *expirationTime*-$2\Delta(1+\rho)$ so that it can renew its election before its leadership expires. The protocol provides a Boolean function *Leader?* that checks if the calling process p is a local leader: when the function *Leader?* reads its hardware clock at time t, it is leader at t iff *expirationTime* $> H_p(t)$ and the flag *imLeader* is true.

In case p does not succeed to become local leader and p has received at least one supportive reply, p broadcasts a "Release"-message to let all processes that are locked to p know that they can lock to another process. When a process q receives a Release-message from p for the last election message that q has locked to, q releases its lock.

IX. Correctness

We show in this section that the proposed local leader protocol satisfies the requirements (T,SO,BI,LS). To do this, we first have to define the predicates *stablePartition*, $Leader_p$ and $supports^t$. Since our protocol is designed for Δ-partitions, we define

$$stablePartition(SP,s,t) \triangleq \Delta\text{-}partition(SP,s,t).$$

Let us denote the value of process p's variable *varname* at time t by $varname_p^t$. The predicate $Leader_p$ is defined by,

$$Leader_p(t) \triangleq H_p(t) < expirationTime_p^t \wedge imLeader_p^t.$$

Before a process becomes leader, it sets its variable *supportSet* to the set of processes that have sent a supportive reply. A process q *supports* a process p at time t iff p is leader at time t and q is in p's support set:

$$supports^t(q,p) \triangleq Leader_p(t) \wedge q \in supportSet_p^t.$$

Formally, we express the property that process q is locked to process p at t by a predicate $locked_q^t(p)$ that is defined as follows (see Section VIII and Figure 31 for an explanation of variables *LockedTo* and *LockedUntil*):

$$locked_q^t(p) \triangleq \exists T{:}LockedTo_q^t=(p,T) \wedge H_q(t) \leq LockedUntil_q^t.$$

As long as q does not crash, it is locked to at most one process at a time because q always checks that its last lock has expired before it locks to a new process. Recall that the timed asynchronous system model allows crashed processes to recover. We assume that when a process q crashes, it stays down for at least *lockTime* clock time units. This enables q to lock to a process p immediately after q recovers from a crash without waiting first for at least *lockTime* clock time units to make sure that any lock q had issued before it crashed has expired. Note that this initial waiting time would otherwise (i.e. without the above assumption) be required to make sure that a process is locked to at most one process.

To simplify our exposition, we use the phrase "process p does action a at clock time T .." to denote that "process p does action a at some point in real-time t when p's hardware clock H_p shows value T, i.e. $H_p(t) = T$, ..".

A. Supports At Most One (SO)

To see that requirement (SO) is satisfied, i.e. that a process q supports at most one process p at a time, let us consider the scenario shown in Figure 18. Process p sends an Election-message m at clock time $S = lastRequest_p$, q receives m at T, and q sends a supportive reply at U. Process q is locked to p for at least *lockTime* clock units, i.e. until its hardware clock shows a value $Y \geq T + lockTime$. When process p becomes leader at time W and q is in p's support set, it assigns its variable $expirationTime_p$ the value $S + lockTime(1-2\rho)$. Since p's and q's hardware clocks can drift apart by at most 2ρ (and p's hardware clock shows S before q's hardware clock shows T due to the positive transmission delay of m), p's hardware clock shows value $expirationTime_p$ before q's hardware clock shows $T + lockTime$ (see Figure 18). Hence, whenever a process q supports p, q is locked to p. Since q is locked to at most one process at a time, it follows that q supports at most one process at a time.

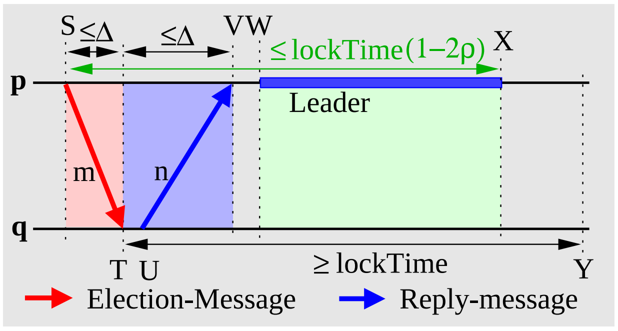

Fig. 18. A process q that sends a supportive reply n to p's Election-message m is locked to p for *lockTime* clock units, while p is demoted before q releases its lock to p.

B. Leader Self Support (LS)

Before becoming leader, a process p checks that it has received a supportive reply from itself (see condition *myid = replySet.Min()* in procedure *CheckIfLeader*). Thus, a leader always supports itself and the protocol therefore satisfies requirement (LS).

C. Timeliness (T)

For the timeliness condition (T) to hold, we have to make sure that the protocol constants κ, *expires* and *lockTime* are well chosen. Constant *expires* states for how long a process p stays in the in the *aliveSet* of a process without the arrival of a new fast message from p. Hence, we have to derive a lower bound for *expires* to make sure that processes in a Δ-partition do not remove their local leader from their *aliveSet*.

To derive a lower bound for constant *expires*, let us consider the situation in which a timely process p tries to become leader by sending periodic Election-messages (see Figure 19). The goal is that p stays in the alive-set of each process q that is connected to p and q stays in the alive-set of p. Therefore, the constant *expires* has to be chosen such that for any two successive Election messages m_1 and m_2 sent by p and that q receives at clock times S and T, respectively, condition $T - S \le expires$ holds. Similarly, when p receives q's replies n_1 and n_2 at times U and V, respectively, the distance between these two receive events should be at most *expires*: $V - U \le expires$. We therefore derive upper bounds for $T - S$ and $V - U$ under the assumption that p and q are connected. The duration between two successive broadcasts is at most EP clock time units and the real-time duration is thus at most $EP(1+\rho)$. The difference in the transmission delays of m_1 and m_2 is at most $\Delta - \delta_{min}$. Thus, $T - S$ is bounded by,

$$T\text{-}S \le (1+\rho)(EP(1+\rho)+\Delta\text{-}\delta_{min}) \le expires.$$

Since the clock time between transmitting m_1 and m_2 is at most EP, and the maximum difference between the round-trip times (m_2, n_2) and (m_1, n_1) are $2\Delta - 2\delta_{min}$, *expires* has to be bounded by,

$$V\text{-}U \le EP+(1+\rho)2(\Delta\text{-}\delta_{min}) \le expires$$

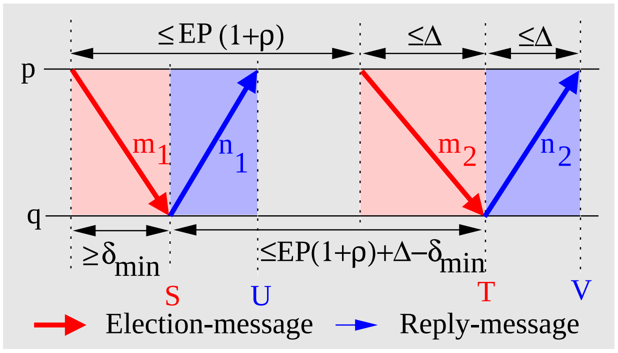

Fig. 19. Constant *expires* is chosen so that p stays in q's alive-set between S and T and q stays in p's alive-set between U and V.

Typically one will set the constant *expires* to a small multiple of the lower bound so that one late election message or an omission failure of an Election-message does not remove the sender from the alive-set of the other processes. In our implementation we defined the constant as about four times the lower bound and achieved excellent stability in the sense that during our measurements the minimum process was not removed from the alive-sets unless we crashed the process or partitioned the system.

C.1 Constraining κ

We will now show that all processes in a Δ-partition SP will implicitly agree on the process with the minimum id in SP. Let us consider that SP forms at time s and stays stable at least until time $y > s$, and that $p = min(SP)$ (see Figure 20). Any message m that a process $q \in SP$ receives from a process r outside of SP, i.e. $r \in \mathcal{P} - SP$, has a transmission delay of more than Δ. Hence, the fail-aware datagram service delivers m as a slow message and q will therefore not update its array *lastMsg* and its *aliveSet* for r. After *expires* clock time units, that is, after time $s + expires(1+\rho)$, all processes outside of SP are removed from the alive-sets of the processes in SP. Thus, after time $s + expires(1+\rho)$, p's id is smaller than the id of any other process in its alive-set. Process p's *aliveTimer* will generate a "timeout" event no later than time $t \triangleq s + (expires + \sigma)(1+\rho)$, since a process sets its *aliveTimer* so that it generates a timeout event within σ time units of p's id becoming smaller than any other id in its alive-set. Hence, p will broadcast its first Election-messages m_1 after s no later than time t. The first election request can fail because p's target set does not necessarily contain p's own id. All processes in SP receive a fast m_1 no later than $u \triangleq t + \Delta$. Hence, they all will include p in their alive-sets no later than time u. After time u no process in SP will lock to any other process than p since p has the minimum id in their alive-sets. All processes in SP will reply to p and p

will get these replies as fast messages since p is connected to all processes in SP by our hypothesis. Process p will therefore include all processes in SP in its alive-set: after time $u+\Delta$, p's alive-set consists of all the processes in SP, i.e. $aliveSet_p = SP$.

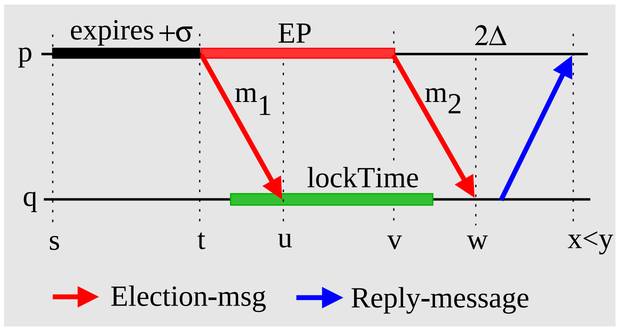

Fig. 20. The process $p = min(SP)$ in a Δ-partition SP that forms at s takes up to *expires*+σ clock time units to send its first Election-message m_1 and it will send the second at most EP later. We show that p succeeds to become leader with m_2 even when a process q would be locked to another process at u.

After time t, p will broadcast at least every EP clock units an Election-message. Note that p is timely because p is by our hypothesis in a Δ-partition. Hence, p sends its next Election-message m_2 no later than time $v \triangleq t + EP(1+\rho)$ (see Figure 20). Below we will constrain $lockTime$ so that even when q would be locked to another process at u, q will have released that lock before it receives m_2 at w. All processes in SP will send supportive replies for m_2 since:

- no process in SP will be locked to another process anymore,
- p has the minimum id in the alive-sets of all processes in SP, and
- all processes in SP receive m_2 as a timely message.

Process p will become leader no later than time $v+2\Delta$. For our protocol to satisfy requirement (T), we have therefore to constrain κ as follows:

$$\kappa \geq (expires + \sigma + EP)(1+\rho) + 2\Delta.$$

C.2 Constraining *lockTime*

When a timely process p broadcasts an Election-message at time s, then p receives the replies of connected processes within 2Δ time units. The *releaseTimer* is therefore set to $2\Delta(1+\rho)$, where the factor of $(1+\rho)$ takes care of the drift of p's hardware clock. To ensure that a process becomes leader for a positive time quantum, the *lockTime* has to be chosen such that p's leadership does not expire before it actually starts. Since (1) p stays leader for at most $lockTime(1-2\rho)$ after sending its election message m (see Figure 18), and (2) it takes up to $2\Delta(1+\rho)+\sigma(1+\rho)$ clock time units until p's *releaseTimer* timeouts after sending m, we have to make sure that

$$lockTime(1-2\rho) > 2\Delta(1+\rho) + \sigma(1+\rho).$$

Thus, we assume the following lower bound for *lockTime* :

$$lockTime > (2\Delta + \sigma)(1+3\rho).$$

We now derive an upper bound for constant *lockTime*. The goal is that when process q has locked to a process $r > p$ just before receiving m_1, q should release its lock before q receives the next message m_2 from p (see Figure 21). When p does not become leader with m_1, it schedules its next request in at most EP clock time units. Due to the scheduling imprecision, a timely p will actually send m_2 within $[EP-\sigma, EP]$ clock time units. The drift rate of p's hardware clock is within $[-\rho, +\rho]$ and thus, the real-time duration between the two send events is at least $(EP-\sigma)(1-\rho)$. The real-time duration between the reception of m_1 and m_2 by q is at least $(EP-\sigma)(1-\rho)-\Delta+\delta_{min}$ because the difference in the transmission delays of m_1 and m_2 is at most $\Delta - \delta_{min}$. Therefore, constant *lockTime* should be at most,

$$lockTime \leq (1\text{-}\rho)\ [(EP\text{-}\sigma)(1\text{-}\rho)\text{-}\Delta\text{+}\delta_{min}]$$

In our protocol we set *lockTime* to this upper bound, i.e. we choose *lockTime* as big as possible for a given EP.

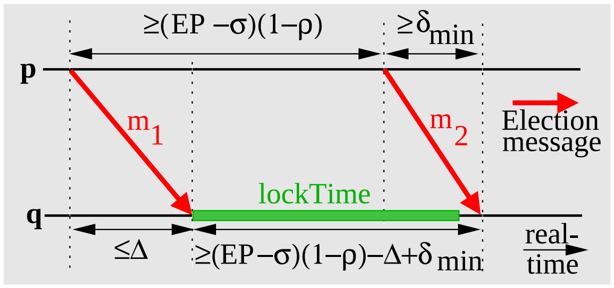

Fig. 21. Process q locks just before receiving m_1 to a process $r > p$: q should release its lock before it receives the next Election-message m_2 from p.

D. Bounded Inconsistency (BI)

To show that the protocol satisfies requirement (BI), we consider two processes p and q that become connected at some time s and stay connected until some time $y \geq s+\kappa$ (see Figure 22). Let p have the smaller id of the two processes, i.e. $p < q$. We have to show that for any time x such that $s+\kappa \leq x \leq y$ and p is leader at x that q supports p at x. Let us consider that m_0 is the last Election-message that p has sent before s, and m_1 is the first that p sends after the two processes become connected at s. A process p includes itself in its alive-set only when p receives a timely election or reply message from itself. The broadcast of m_0 could result in a timely reply message n_0 to itself. As a consequence of m_0, p could include itself in its alive-set after s with a receive time stamp of up to $H_p(s+2\Delta)$. When p does not send another Election-message m_1 for more than *expires* clock time units, p will remove itself from its alive-set. Note that after p has been removed from its own alive-set, p is not in the target set and the first election request of p will fail because p needs to be in its target set to become leader (see function *CheckIfLeader*). Hence, when p sends the first Election-message $2\Delta + expires(1+\rho)$ time units after s, this election request will fail. However, q's reply to m_1 will force p to get a supportive reply for all successive election requests. In other words, after p receives n_1, it can only become leader with the support of q (as long as p and q stay connected). When p sends its first Election-message within $2\Delta + expires(1+\rho)$ time units, p's election can be successful even without the support of q because p checks that it has become leader as soon as it has received a reply from all processes in its target set. However, p's leadership will expire within *lockTime(1-2ρ)* clock time units. Due to n_1, p cannot renew its election without q's support. Thus, the requirement (BI) is satisfied for

$$\kappa \geq 2\Delta + (1+\rho)(\text{expires} + \text{lockTime}(1\text{-}2\rho)).$$

Note that this bound for κ is smaller than the previously derived bound to satisfy the timeliness condition (see Section IX-C)

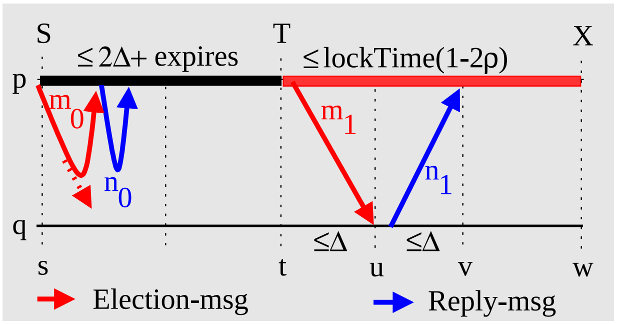

Fig. 22. Processes p and q become connected at s and p will include q in its alive-set at time v. Thus, any Election-message p sends after v requires a supportive reply from q to allow p to become leader.

X. Protocol Properties

The proposed protocol elects within κ time units in any Δ-stable partition SP the process with the minimum id as local leader. Process $p = min(SP)$ will stay leader as long as SP stays stable: after p becomes leader, it reduces its time-outs for broadcasting its Election-messages from EP to *lockTime(1-2ρ)-2Δ*$(1+\rho)$. This makes sure that as long as SP stays stable p can renew its election before its leadership expires in *lockTime(1-2ρ)*.

A local leader p always knows its logical partition LP: $LP = supportSet_p$. Note that the definition of *supports* that we give for our protocol (see Section IX) states that a process r only supports another process q if q is leader and r is in q's support set. Hence, for our protocol a process r cannot support a process q that is not leader or a local leader that does not know of r's support. In this way, in our protocol we actually exclude situations like that depicted in Figure 10 in which a logical partition contains processes that support a process other than the local leader.

Since logical partitions do not overlap, at no point in time do the support sets of local leaders overlap. The support set is the basis for our implementation of a membership protocol [11]. Process p piggy-backs its support set on its next election message to provide all processes in its logical partition with the current members in their logical partition.

The proposed protocol guarantees a stronger timeliness requirement than the one required by the specification. In

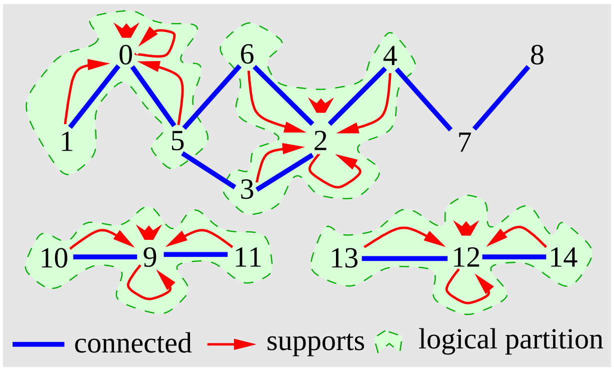

Fig. 23. The protocol elects in each maximum connected set at least the process with the minimum id as local leader.

particular, the connected relation does not have to be transitive to guarantee that a local leader is elected within κ. We say that a set of processes CS is a *maximum connected set* iff

- any two processes p and q in CS are either connected or Δ-disconnected,
- for any two processes p and q in CS there exists a path in the connected-graph between p and q, and
- any process in CS is Δ-disconnected from all processes outside of CS.

The protocol succeeds to elect the minimum process in any maximum connected set within κ time units (see Figure 23). The behavior of the protocol for any maximum connected set CS can be described by a graph algorithm (see Figure 24). Initially, the set S contains all processes in CS, no process in CS is marked (i.e. the set MS is empty), and the set of local leaders LS is empty. The algorithms iteratively computes the minimum process l in set S. When l and all processes connected to l are not marked (i.e. they are not in the set MS), l is included in the set of local leaders LS. The intuition of a process being 'marked' is that a marked process has already locked to another process. All processes that are connected with l are marked by being included in the set MS. The algorithm terminates when S becomes empty. After the algorithm has terminated, LS contains exactly the set of local leaders that the proposed local leader election protocol will elect in the maximum connected set CS.

The protocol can be configured so that it guarantees that there exists at most one leader at a time, i.e. the safety property (S) of the conventional leader election problem is satisfied. Let N denote the maximum number of processes that are participating in the election protocol, i.e. $N = |\mathcal{P}|$. By setting constant *minNumSupporters* to $\lceil \frac{N+1}{2} \rceil$ a process has to get the support of more than half of the processes to become leader (see function *CheckIfLeader* in Figure 31). Thus, any local leader is in a logical partition with more than $N/2$ processes and because logical partitions do not overlap, there can be at most one leader at a time. The modified protocol satisfies all requirements (T,SO,LS,BI). However, we have to define predicate *stablePartition* in the following way:

```
S $\leftarrow$ CS;
LS $\leftarrow \emptyset$;
MS $\leftarrow \emptyset$;
while S $\neq \emptyset$ do
    l $\leftarrow$ min S;
    S $\leftarrow$ S$-$\{l\};
    for all p $\neq$ l:
        if connected(l,p) then
            if p $\in$ MS then
                MS $\leftarrow$ MS $\cup$ \{l\};
            MS $\leftarrow$ MS $\cup$ \{p\};
        endif
    if l $\notin$ MS then
        LS $\leftarrow$ LS $\cup$ \{l\};
end
```

Fig. 24. This algorithm computes the set of local leaders *LS* that the proposed local leader election protocol elects in a maximum connected set *CS*.

$$stablePartition(SP,s,t) \overset{\Delta}{=} \Delta\text{-}partition(SP,s,t) \wedge |SP| \geq \lceil \tfrac{N+1}{2} \rceil.$$

XI. Performance

We measured the performance of the local leader election protocol in our Dependable Systems Lab at UCSD, on 8 SUN IPX workstations connected by a 10 Mbit/s Ethernet. All messages are send by a fail-aware datagram service [23] that classifies any message it delivers as either "fast" or "slow". The service calculates an a posteriori upper bound on the transmission delay of a message m and if this calculated bound is not greater than some given threshold Δ, m is classified as "fast" and otherwise, m is classified as "slow". The fail-aware datagram service guarantees that the Validity and the Non-duplication requirements hold (see Section III).

The threshold Δ for fast messages was $\Delta = 15ms$, the timeout for the scheduling delay σ was $30ms$, and the election period EP was $50ms$. The typical behavior of the protocol is that exactly one process p is periodically broadcasting election messages and all other processes are replying with unicast messages to p. The measured election times, i.e. the time between transmitting an election message by a process p and the time at which p becomes leader, reflects this typical behavior (see Figure 25). These measurements were based on 100000 successful elections. The election time increases linearly with the number of processes participating in the election: the average election time and the 99% election time, i.e. a process succeeds with a 99% probability to become leader within that time, are shown in Figure 26.

We also measured the time it takes to elect a new leader when the system splits from one into two partitions (see Figure 27). The graph is based on about 12000 measurements and was performed using the leader election protocol as part of a membership protocol [11]. A process p removes a process q from its alive-set when p has not received a fast message from q for more than *expires=230ms*. In other words, when the system splits up, it takes the processes in one partition up to $230ms$ to remove all processes

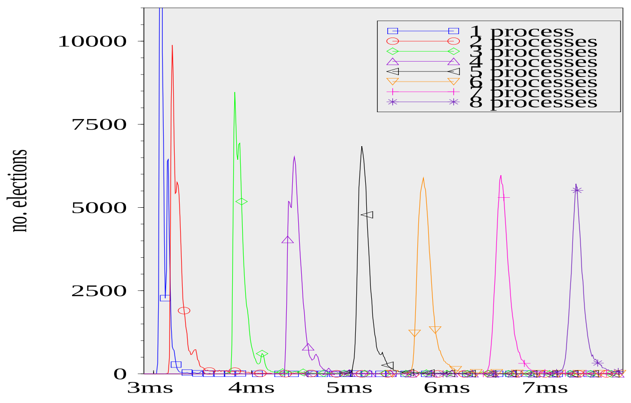

Fig. 25. Distribution of the election time for 1 to 8 processes participating in the election. The smaller difference between 1 and 2 processes is due to the fact that no local replies are sent when there is more than 1 process.

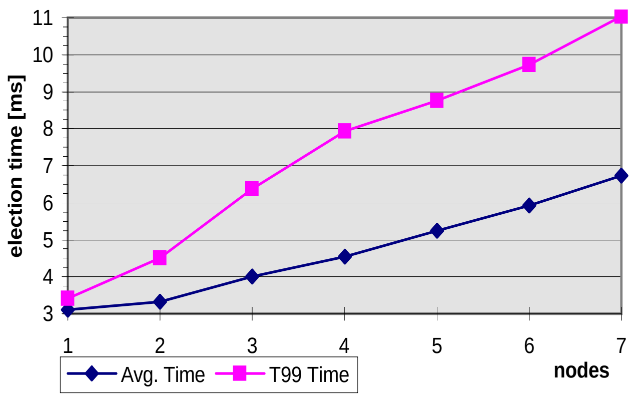

Fig. 26. The average election time and 99% election time for 1 to 7 participating processes.

from the other partition from their alive-sets. However, *expires* could be reduced to about $50ms$ (see Section IX-C). However, we use the larger value of $230ms$ to minimize the number of parallel local leaders in case of temporary instabilities. The first election attempt after the system becomes partitioned typically fails because the alive-sets of the processes in each of the two newly formed partitions are not up-to-date yet. The second election attempt however does in general succeed to elect a new local leader.

The linear increase of the election time with the number of participating processes is mainly due to the fact that a process receives more Reply-messages to its election message and less to the fact that the Ethernet is overloaded. One possible enhancement of the protocol would be to use the alive-set provided in an election message to build an n-ary tree and use this tree to collate replies (see Figure 28): (1) a process on the leaves of the tree replies to its parent node, (2) a process in an inner node waits for the replies of its children before it replies to its parent node, and (3) the root becomes leader when it has received replies from all its children. While reducing the election time, such an enhancement would complicate the protocol since it has to handle the case that a process q in the tree crashes or is too slow and hence, the root would not get any message from any of the children of q.

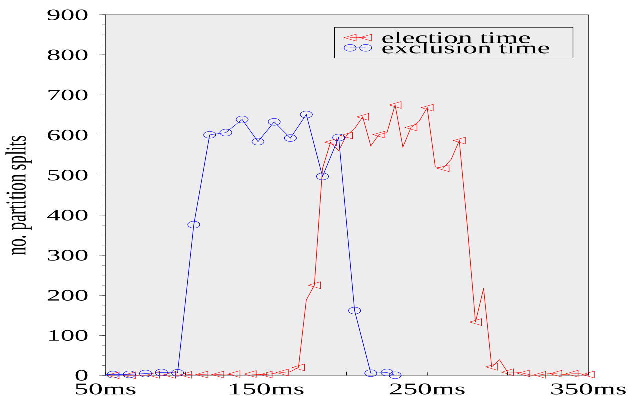

Fig. 27. The left graph shows the measured time l takes until it removes all processes from the other partition from its alive-set. This is the time l waits until it attempts to become local leader. The right graph shows the measurement of the time needed to elect a new leader l after a partition split.

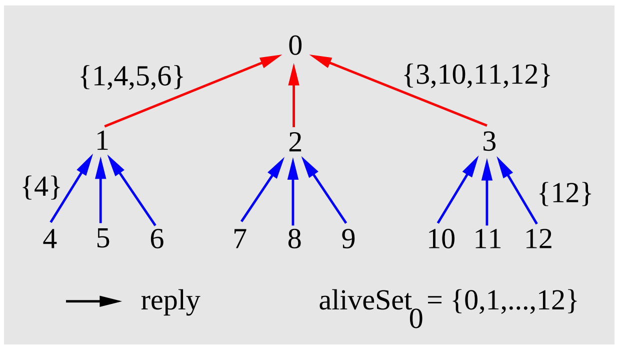

Fig. 28. The alive-set of the root process 0 is used to define a 3-ary tree which is used to collate replies of the processes in $aliveSet_0$. Processes outside $aliveSet_0$ reply directly to 0.

We implemented the tree based collation of replies to compare its effect on the election time. For a system with eight processes, three processes directly replied to the new leader l, while 3 other processes had to reply to another process q before q replied to l. The improvement for 8 processes was not sufficient to justify the increased complexity (see Figure 29). However, for systems with more processes this enhancement could decrease the election time significantly.

XII. Conclusion

The ideal goal of a local leader service is to elect exactly one local leader in each communication partition. This goal is however not always achievable because the failure frequency in a communication partition might be too high for the processes in this partition to be able to elect a local leader. In this paper, we derive a specification which approximates this ideal goal of having exaclty one local leader per communication partition. We also show that this specification is implementable in timed asynchronous systems, i.e. distributed systems without upper bounds on the transmission or scheduling delay but in which processes have access to a local hardware clock with a bounded drift rate.

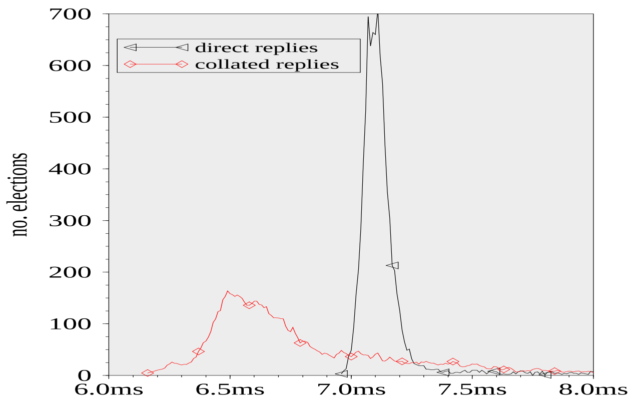

Fig. 29. Comparison of tree based collation of replies and direct replies to the leader. The two graphs are based on 10000 elections each.

The local leader problem requires that an implementation creates non-overlapping *logical partitions* and elects one leader per logical partition. A logical partition created by our local leader service is a set of processes such that the local leader of this logical partition can communicate in a timely fashion with all processes in the logical partition. If the connected-relation changes, e.g. a process becomes disconnected from the leader, the local leader service has to adapt the logical partition to the new connected-relation. A stable partition is a communication partition in which all processes are connected with each other. Therefore, a local leader service has to create for each stable partition SP a logical partition that includes SP and has to elect a leader l in SP within a bounded amount of time after SP has formed.

The specification of a local leader service can efficiently be implemented in timed asynchronous systems. We introduce in this paper a round-base local leader election protocol: a leader is only elected for some maximum duration and has to update its leadership during the next round. This periodic update is necessary to be able to adapt the logical partitions to changes of the connected-relation. For example, if the current local leader l crashes or is disconnected, the remaining processes can elect a new leader within a bounded amount of time since l is demoted with some known amount of time. In a stable partition of N processes, the protocol sends one broadcast datagram and $N-1$ unicast datagrams per round. A local leader service has been proven useful in the design and implementation of a fail-aware membership service and a fail-aware clock synchronization service for asynchronous partitionable systems.

```
import
    const    myid : P;
    const    $\Delta$ : time;
    const    $\delta_{min}$ : time;
    const    $\sigma$ : time;
    const    EP : time;
    const    $\rho$ : time;
    const    expires : time;
    function  H : time;
    function  newLeader(demotedAt: time);
    procedure  fa−send(m : msg, destination : P, [sendTime: time]);
    procedure  fa−broadcast(m : msg);

const
    lockTime : time init $(1-\rho)(EP(1-\rho)-\Delta+\delta_{min})$;
    minNumSupporters: integer init 1;

var
    aliveAC : alarm clock;
    releaseAC: alarm clock;
    imLeader: boolean init false;
    expirationTime: time init 0;
    supportSet: set init $\emptyset$;
    aliveSet: set init $\emptyset$;
    replySet: set init $\emptyset$;
    targetSet: set init $\emptyset$;
    lastMsg: P $\rightarrow$ time init $\lambda$.0;
    lockedTo : P x time;
    lockedUntil: time init 0;
    lastRequest: time init 0;

msg
    (“Election”, request : time, alive : set);
    (“Reply”, request : time, support : boolean);
    (“Release”, request : time);

function  PurgeAliveSet(now: time) : time
    noMinBefore : time init now;
    for all  p $\in$ aliveSet:
        if  now $\geq$ lastMsg[p]+expires  then
                aliveSet.Remove(p);
        else  if  myid $>$ p $\wedge$ lastMsg[p]+expires $>$ noMinBefore  then
                noMinBefore = lastMsg[p]+expires;
    return  noMinBefore;
end

procedure  UpdateAliveSet(sender: P, fast: boolean, recTime: time)
    if  fast  then
        aliveSet.Insert(sender);
        lastMsg[sender] $\leftarrow$ recTime;
    endif
    PurgeAliveSet(recTime);
end

procedure  CheckIfLeader ()
    if  myid $\in$ targetSet $\wedge$ replySet = aliveSet
        $\wedge$ myid = replySet.Min()
        $\wedge$ replySet.Size() $\geq$ minNumSupporters  then
        supportSet $\leftarrow$ replySet;
        imLeader $\leftarrow$ true;
        expirationTime $\leftarrow$ lastRequest + lockTime$(1-2\rho)$;
        aliveAC.SetAlarm(expirationTime$-2\Delta(1+\rho)-\sigma$);
        newLeader(expirationTime);
    else
        aliveAC.SetAlarm(lastRequest+EP$-\sigma$);
        if  replySet $\neq \emptyset$  then
            fa−broadcast(“Release”, lastRequest);
    endif
    lastRequest = 0;
end
```

Fig. 30. Part 1 of the pseudo-code for a local leader service.

```
function  Leader?() : boolean
    return  H() $<$ expirationTime $\wedge$ imLeader;
end

task  LeaderElection
import
    fa−deliver(m : msg, sender : P, fast : boolean,
                    recTime : time);

begin
    aliveAC.SetAlarm(H());
    loop
        select event
            when fa−deliver((“Election”, request, alive),
                                        sender, fast, recTime):
                UpdateAliveSet(sender, fast, recTime);
                support $\leftarrow$ (lockedUntil $<$ H() $\vee$ sender = lockedTo.P)
                            $\wedge$ sender = aliveSet.Min()
                            $\wedge$ sender $\leq$ myid $\wedge$ fast;
                if  support  then
                    (lockedTo,lockedUntil) $\leftarrow$ ((sender, request),
                                            recTime+lockTime);
                endif
                if  fast  then
                    if  sender $\neq$ myid $\vee$ alive.Size() $\leq$ 1  then
                        fa−send((“Reply”, request, support),
                                        sender, recTime);
                    else  if  support $\wedge$ request = lastRequest  then
                        replySet.Insert(myid);

            when fa−deliver((“Release”, request), sender,
                                        fast, recTime):
                if  lockedTo = (sender, request)  then
                    lockedUntil = 0;

            when fa−deliver((“Reply”, request, support),
                                        sender, fast, recTime):
                UpdateAliveSet(sender, fast, recTime);
                if  fast $\wedge$ request = lastRequest $\wedge$ support  then
                    replySet.Insert(sender);
                    if  replySet = targetSet $\wedge$ Leader?()  then
                        CheckIfLeader ();
                        releaseAC.Cancel();
                    endif
                endif

            when aliveAC.WakeUp(T):
                lastRequest = H();
                nextTimeout : time init PurgeAliveSet(lastRequest);
                replySet $\leftarrow \emptyset$;
                targetSet $\leftarrow$ aliveSet;
                if  aliveSet = $\emptyset$ $\vee$ myid $\leq$ aliveSet.Min()  then
                    fa−broadcast((“Election”, lastRequest,
                            aliveSet), lastRequest);
                    releaseAC.SetAlarm(lastRequest+$2\Delta(1+\rho)$);
                else
                    aliveAC.SetAlarm(nextTimeout);

            when releaseAC.WakeUp(T):
                CheckIfLeader ();

        end select
    end loop
end
```

Fig. 31. Part 2 of the pseudo-code for a local leader service.

XIII. Appendix

Sym.	Sec.	Meaning
$aliveSet_p$	VII	independent assessment view of p
(BI)	V	bounded inconsistency requirement
$connected$	III	timely message exchange possible
δ	III	one-way time-out delay
δ_{min}	III	minimum message transmission delay
Δ	IV	threshold used to classify messages
Δ-disconnected	IV	a weak form of being disconnected
Δ-partition	IV	a stable partition
EP	VIII	election period
$expires$	VIII	expiration time of alive-set entries
$fast$	IV	upper bound of a fast message $\leq \Delta$
H_p	III	hardware clock of process p
κ	V	maximum leader election time
$Leader_p(t)$	V	p is leader at t
$lockTime$	VIII	support time for leader
(LS)	V	"leader self support" requirement
$\mathcal{P}$	III	set of processes
p,q,r		processes
ρ	III	maximum drift rate of a hardware clock
s,t,u,v		real-time values
S,T,U,V		clock time values
σ	III	scheduling time-out delay
$slow$	IV	upper bound of a slow message $> \Delta$
(SO)	V	"Support at most One" requirement
$stablePartition$	V	formalization of a stable partition
$supports$	V	supports-relation
$SUPPORT$	VI	closure of supports-relation
(T)	V	timeliness requirement

References

[1] G. LeLann, "Distributed systems - towards a formal approach," in *Information Processing 77*, B. Gilchrist, Ed. North-Holland, 1977.

[2] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 49–59, Jan 1982.

[3] H. Abu-Amara and J. Lokre, "Election in asynchronous complete networks with intermittent link failures.," *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 778–788, 1994.

[4] H.M. Sayeed, M. Abu-Amara, and H. Abu-Amara, "Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links.," *Distributed Computing*, vol. 9, no. 3, pp. 147–156, 1995.

[5] J. Brunekreef, J.-P. Katoen, R. Koymans, and S. Mauw, "Design and analysis of dynamic leader election protocols in broadcast networks," *Distributed Computing*, vol. 9, no. 4, pp. 157–171, 1996.

[6] G. Singh, "Leader election in the presence of link failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 3, pp. 231–236, March 1996.

[7] F. Cristian, "Reaching agreement on processor-group membership in synchronous distributed systems," *Distributed Computing*, vol. 4, pp. 175–187, 1991.

[8] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Communications of the ACM*, vol. 39, no. 4, pp. 64–70, Apr 1996.

[9] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 54–63, Apr 1996.

[10] K. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, Dec 1993.

[11] C. Fetzer and F. Cristian, "A fail-aware membership service," in *Proceedings of the 16th Symposium on Reliable Distributed Systems*, Oct 1997, pp. 157–164, http://www.cs.ucsd.edu/~cfetzer/FAMS.

[12] C. Fetzer, "Fail-aware clock synchronization," Tech. Rep. Time-Services, Dagstuhl-Seminar-Report; 138, Mar 1996, http://www.cs.ucsd.edu/~cfetzer/FACS.

[13] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes.," *Communications of the ACM*, vol. 22, no. 5, pp. 281–283, May 1979.

[14] R. Carr, "The Tandem global update protocol," *Tandem Systems Review*, Jun 1985.

[15] Cary G. Gray and David R. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Dec 1989, pp. 202–210.

[16] F. Jahanian, S. Fakhouri, and R. Rajkumar, "Processor group membership protocols: Specification, design and implementation," in *Proceedings of the 12th Symposium on Reliable Distributed Systems*, Princeton, NJ, Oct 1993, pp. 2–11.

[17] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala, "Processor membership in asynchronous distributed systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 5, pp. 459–473, May 1994.

[18] F. Cristian and F. Schmuck, "Agreeing on processor-group membership in asynchronous distributed systems," Tech. Rep. CSE95-428, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.

[19] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Transactions on Parallel and Distributed Systems*, To appear in 1999. http://www.cs.ucsd.edu/~cfetzer/MODEL.

[20] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of ACM*, vol. 34, no. 2, pp. 56–78, Feb 1991.

[21] D Powell, "Failure mode assumptions and assumption coverage," in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing Systems*, 1992, pp. 386–395.

[22] C. Fetzer and F. Cristian, "Building fault-tolerant hardware clocks," in *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, USA, Jan 1999, pp. 59–78, http://www.cs.ucsd.edu/~cfetzer/HWC.

[23] C. Fetzer and F. Cristian, "A fail-aware datagram service," in *Fault-Tolerant Parallel And Distributed Systems*, D.R. Avaresky and D.R. Kaeli, Eds., chapter 3, pp. 55–69. Kluwer Academic Publishers, 1998, http://www.cs.ucsd.edu/~cfetzer/FADS.

[24] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, pp. 146–158, 1989.

Christof Fetzer received his diploma in computer science from the University of Kaiserlautern, Germany (12/92) and his Ph.D. from UC San Diego (3/97). He is currently a research scientist at UC San Diego and he will join AT&T Labs in 1999. He received a two-year scholarship from the DAAD and two best student paper awards. He was a finalist of the 1998 Council of Graduate Schools/UMI distinguished dissertation award. Dr. Fetzer has published over 25 research papers in the field of distributed systems.

Flaviu Cristian is Professor of Computer Science at the UC San Diego. He received his PhD from the University of Grenoble, France, in 1979. He joined IBM Research in 1982. While at IBM, he worked in the area of fault-tolerant distributed systems and protocols. After joining UCSD in 1991, he and his collaborators have been designing and building support services for providing high availability in distributed systems. Dr.Cristian has published over 100 papers in international journals and conferences in the field of dependable systems.