Fast Leader (Full) Recovery despite Dynamic Faults

Ajoy K. Datta, Stéphane Devismes, Lawrence L. Larmore, Sébastien Tixeuil

Verimag Research Report n^o TR-2012-18

October 12, 2012

Fast Leader (Full) Recovery despite Dynamic Faults

Ajoy K. Datta, Stéphane Devismes, Lawrence L. Larmore, Sébastien Tixeuil

October 12, 2012

Abstract

In this paper, we give a leader recovery protocol, LE(k), that recovers a legitimate configuration where a single leader exists, after at most k memory corruption faults hit the system in an arbitrary manner. That is, if a leader ℓ is elected before state corruption, the *same* leader is elected after recovery. Our protocol works for an anonymous bidirectional, yet oriented, ring of size n, and does *not* require that processes know n, although the knowledge of k is assumed. If $n \geq 18k + 1$, our protocol recovers the leader in $O(k^2)$ rounds using $O(\log k)$ bits per process, assuming unfair scheduling. Our protocol handles *dynamic* faults in the sense that the k memory corruption faults may occur while the network has started recovering the leader.

Keywords: self-stabilization, k-stabilization, leader election, leader recovery

How to cite this report:

```
@techreport {TR-2012-18,
  title = {Fast Leader (Full) Recovery despite Dynamic Faults},
  author = { Ajoy K. Datta, Stéphane Devismes, Lawrence L. Larmore, Sébastien Tixeuil },
  institution = {{Verimag} Research Report},
  number = {TR-2012-18},
  year = {2012}
}
```

1 Introduction

Self-stabilization [9, 10, 18] is often regarded as a strong *forward recovery* mechanism that recovers from any *transient* failure. Informally, a self-stabilizing protocol is able to recover correct behavior in finite time after arbitrary faults and attacks placed the system in some arbitrary initial state. Its generality comes at a price: extra memory could be needed in order to crosscheck inconsistencies; symmetries occurring in the initial state could cause a given problem (*e.g.* leader election or mutual exclusion) to be impossible to solve deterministically, and when few faults hit the network, "classic" self-stabilization does not generally guarantee a smaller recovery time.

Related work.

The intuition that when few faults hit the system, it should be possible to impose more stringent constraints on the recovery than just a basic "eventual" correctness has proven to be a fertile area in recent research [4, 5, 11, 12, 1, 6, 7, 16, 17]. Defining the number of faults hitting a network using some kind of Hamming distance (the minimal number of processes whose state must be changed in order to recover a correct configuration), variants of the self-stabilization paradigm have been given. The notion of k-stabilization [4, 5, 11, 12] guarantees that the system recovers when the initial configuration is at distance at most k from a legitimate configuration. This notion is weaker than self-stabilization, as this latter permits recovering from any configuration. In the literature, weakened forms of self-stabilization were used both for (1) circumventing impossibility results in self-stabilization (*e.g.* deterministic leader election or recovery in anonymous networks) and (2) obtaining recovering times that only depend on the number of faults k (as opposed to n or D, the network size or diameter). In this paper we also deal with these two motivations, as we propose a k-stabilizing leader recovery protocol, whose recovery time is $O(k^2)$ rounds.

The concept of only-k-dependant recovery time has been refined under the name of *time adaptivity* (or fault locality) [1, 6, 7, 16, 17], when the recovery time depends on the *actual* distance f to a legitimate configuration in the initial state. Initial work on time adaptivity required the initial distance not to be greater than k (that is, they are k-stabilizing), but the latest work [7] does not have this limitation and is thus also self-stabilizing; however, it is important to note that it distinguishes between "output" stabilization (which considers only the output variables of each process that are mentioned in the problem specification) and the "state" stabilization (which considers the global state, *i.e.,* all variables used by the protocol). In all aforementioned work, only the output is corrected quickly (that is, depending on f or k), while the global state is recovered slowly (that is, depending on D or n).

Output *vs.* state stabilization has an important practical consequence: if a *new* fault occurs after output stabilization yet before state stabilization, output complexity guarantees are not maintained after the new fault. For networks that are subject to intermittent failures, protocols should strive to provide state stabilization. As a consequence, the "fault gap" (defined as the minimum time between consecutive faults that can be handled by the protocol [15]) remains large (that is, system wide).

Correcting the global state quickly was investigated in the context of self-stabilization for the purpose of *fault containment* [14, 15, 2, 3, 8] (that is, preventing local memory corruptions from propagating to the whole network). The state of the art in this matter nevertheless requires that either only a single process is corrupted [14, 3], faulty processes are surrounded by many correct ones so that few faults can be caught quickly [3], the network is fully synchronous [15], or the recovery guarantee is only probabilistic [8]. The "fault gap" that results from those approaches is significantly reduced, as only a delay that depends on the fault span must separate consecutive faults.

Our contribution.

In this paper, we give a leader recovery protocol that recovers a legitimate configuration where a single leader exists, after k memory corruption faults hit the system in an arbitrary manner. That is, if a leader ℓ is elected before state corruption, the *same* leader is elected after recovery. Our protocol works for an anonymous bidirectional, yet oriented, ring of size n, and does *not* require that processes know n, although the knowledge of k is assumed. If $n \geq 18k + 1$, our protocol recovers the leader in $O(k^2)$ rounds using $O(\log k)$ bits per process, assuming unfair scheduling.

With respect to "output stabilization" [4, 5, 11, 12, 1, 6, 7, 16, 17, 1, 6], our protocol recovers the full correct state quickly (in time $O(k^2)$). With respect to fault-containment [14, 15, 3, 8], LE(k) can handle up to k faults [13, 14, 3], faults can be arbitrarily spread and the network is sparse [3], the network is fully asynchronous and the scheduling is unfair [15], and the recovery property is deterministic [8].

LE(k) also exhibits an interesting property with respect to the "fault gap" metric. In our approach, the k tolerated memory corruptions need not occur in the initial state. In fact, they may occur in a dynamic way after the network has started recovering the leader. In other words, faults that can be handled by our protocol are not only arbitrarily placed, but also arbitrarily timed. For a particular set of k faults, the fault gap between those faults is optimal, that is, zero. However, a delay, that depends on k, still must be observed between sets of k faults in a computation.

2 Preliminaries

Computational Model.

We consider *distributed systems* of n *deterministic anonymous processes* organized into an *oriented ring* : each process p distinguishes one of its neighbors as its *successor* p^+, also called its *right* neighbor, and the other its *predecessor*, or *left* neighbor, p^-. The orientation is consistent: the successor of the predecessor of a process p is p.

Communication between neighboring processes is carried out using a finite number of *locally shared variables*. Each process has its own set of shared variables which it can write and which its two neighbors can read, *i.e.*, the ring is *bidirectional*.

The *state* of a process is defined to be the vector of values of its variables. A *configuration* of the system consists of a state for each process. A process can change its state by executing its *local algorithm*. We assume uniformity, that is, all processes have the same local algorithm. The set of local algorithms defines a *distributed algorithm* on the ring. The local algorithm executed by each process is described using a finite set of *guarded actions* of the form: If $\langle guard \rangle$ then $\langle statement \rangle$. The *guard* of an action at process p is a Boolean expression involving only variables of p and its neighbors. The *statement* of an action of p updates some variables of p. An action can be executed only if its guard is true. An action of a process p is *enabled* in a configuration γ if its guard is true in γ, and p is said to be enabled in γ if at least one of its actions is enabled in γ.

A *computation* is a sequence of configurations γ_0, $\gamma_1, \ldots$ such that γ_0 is an arbitrary configuration, and for each configuration γ_i, the next configuration γ_{i+1} is obtained by atomically executing the statement of at least one action that is enabled in γ_i. A computation is *maximal*, meaning that it is infinite or ends at a *final* configuration, *i.e.,* a configuration at which no process is enabled. A distributed algorithm is *silent* if all its computations end at a final configuration.

The above definitions imply that we use an *unfair distributed scheduler*, or *daemon*. At each step, if the configuration is not final, the daemon selects a non-empty set of enabled processes, each of which then executes an action. *Unfairness* is the property that the daemon is otherwise unconstrained, *i.e.,* it can choose to never select an enabled process, unless it is the only enabled process at some step.

We adopt the convention that an enabled process, once selected, will not stop executing until it is no longer enabled. For example, if a process p is only enabled to execute Action A, and if execution of A causes p to be enabled to execute only Action B, then, if p is selected, it will execute Actions A and B consecutively in a single step.

k-Stabilization.

Let $\mathcal{A}$ be a distributed algorithm. We say that an ordered pair (γ, γ') is a *fault* of $\mathcal{A}$ if there is exactly one process of the network which has a different state in γ' than in γ, and if γ' does not follow from γ by any step of $\mathcal{A}$. We define a *k-fault computation* of $\mathcal{A}$ to be a sequence of configurations $\gamma_0\gamma_1 \cdots$ such that:

1. There are at most k choices of i for which (γ_i, γ_{i+1}) is a fault of $\mathcal{A}$.

2. For all other i, (γ_i, γ_{i+1}) is a step of $\mathcal{A}$.

3. The sequence is either infinite, or ends at a final configuration.

Let $\mathcal{L}$ be a non-empty set of final configurations of $\mathcal{A}$. For a given integer $k > 0$, $\mathcal{A}$ is said to be *k-stabilizing w.r.t.* $\mathcal{L}$ if every k-fault computation of $\mathcal{A}$ which begins at some configuration $\lambda \in \mathcal{L}$ is finite and ends at λ. $\mathcal{L}$ is called the set of *legitimate* configurations of $\mathcal{A}$.

In the problem we address $\mathcal{L}$ has n members; for each process ℓ, there is exactly one legitimate configuration in which ℓ is the leader.

3 High Level Overview of LE(k)

In a legitimate configuration of LE(k), there is one leader process ℓ, and no action of LE(k) is enabled. Once a fault occurs, LE(k) starts. If a legitimate configuration is achieved, LE(k) is done. If at most k faults occur, the computation will end, and the last configuration will be the same as the first, namely a legitimate configuration where ℓ is the leader.

Define the *interval of relevance* of a process p to be the set of all processes within distance $3k$ of p, which has $6k + 1$ processes in all. Every process has a *vote*, and in a legitimate configuration, every process within ℓ's interval of relevance votes for ℓ, while every other process' vote is $\perp$. Since the system is anonymous, a process p's vote for a process q is a *relative address*, namely i where q is i steps to the right of p, or $-i$ if q is i steps to the left of p. In particular, in a legitimate state, ℓ will be the unique process whose vote is 0.

Since a fault can change any variable, it can change the vote of a process. We will see that a single fault can cause up to three processes to change their votes, but not more. Thus, throughout any k-fault computation of LE(k), there will be at least $3k + 1$ votes for ℓ, and at most $3k$ votes for any process other than ℓ.

Every process p has a *rumor* field as well, which is either $\perp$, or is the "rumor" that some process, say q, is the leader. In a legitimate configuration the rumor fields of all processes are the same as their votes.

Processes do not change their votes easily, but rumors spread rapidly. If the rumor field of a process p is different from its vote, it must decide whether to change its vote to match the rumor. To make this decision, p initiates a *query* to count votes for the rumored leader. In the following, a rumored leader will be called a *candidate*. If the rumor field is $\perp$, p can initiate a query where the candidate is the process that p is voting for.

A *query* is initiated by a process, say p, called the *home process* of the query, and traverses a path of query variables called its *query path*. During that traversal, the query visits every process within the interval of relevance of its candidate process, say q, and counts all votes for q. Upon returning to p, it reports the count of votes. If q receives at least $3k + 1$ votes, p concludes that q is the leader; otherwise, p concludes q is not the leader.

There are three *query tracks*, which span the entire ring, each intersecting each process at a *query variable*. The *first* query track consists of the variable p.*track*_I for each process p, while the *second* and *third* query tracks consist of the variables p.*track*_II, or p.*track*_III, respectively, for each p. A *query* consists of one *live query token*, which is located in one of the query tracks. The process at which the live token is located is called the *host* of the query. Each query, initiated by its *home process*, moves leftward along the first query track, eventually crossing to the second query track. It then moves rightward along the second query track, eventually crossing to the third query track, and finally moves leftward along that track until it returns to its home process.

A query moves by forward copying and rear deletion. When a live token is copied to the next query variable in the path, the old copy is designated *dead* and must be deleted before the live token can be copied forward. We think of the query as a moving virtual token, which must visit every process within its candidate's interval of relevance, namely the processes within distance $3k$ on both sides of the candidate.

Each query has a *home process* and a *candidate process*. Each query counts the number of processes voting for its candidate while it traverses the second query track. At the end of its path, it reports that total to its home process.

During the time the query is outstanding, its home process p will not change its vote (unless it faults) but its rumor field might change. If the candidate of the query differs from p's vote, and if the query reports that the candidate has at least $3k + 1$ votes in the interval of relevance, then p changes its vote to be for that candidate and initiates a new rumor that the candidate is the leader, unless its rumor variable is already for that candidate. Otherwise, *i.e.,* the query reports no more than $3k$ votes for the candidate, p does not change its vote (or changes it to $\perp$ if the vote was already for the candidate) and initiates a *denial*, which floods the interval of relevance with the information that the candidate is not the leader, and then self-deletes. That denial wave (unless it is interrupted by a fault or a higher priority denial wave) causes all rumors for the candidate to be deleted.

If p's rumor field is $\perp$, but p is voting for a process q, then p initiates a query where q is the candidate. If the query counts at least $3k + 1$ votes for q, then p changes its rumor to q; but if the query counts at most $3k$ votes, p changes its vote to $\perp$ and also issues a denial for q.

If a process p is voting for a false leader, it will eventually change to vote for true leader, ℓ. First of all, if another process, say q, is voting for ℓ but has a rumor supporting some other candidate, say m, it initiates a query with m as the candidate. When q discovers that m is not the leader, it issues a denial of the rumor. If another false rumor spreads to q, it will again send out a query. Eventually, q will send a query whose candidate is ℓ. When this query returns with the information that ℓ has at least $3k + 1$ votes, q will issue the rumor that ℓ is the leader. Processes voting for false leaders will see this rumor, and will then initiate their own queries, confirming that ℓ is the leader.

Rogue Queries.

Faults can create *rogue queries*. A query is rogue if its home is a process p but p did not initiate it. We will see that one fault can cause up to nine rogue queries to be created. In the worst case, there is no way to distinguish a rogue query from one that was initialized normally. Thus, LE(k) cannot specifically delete rogue queries.

Lost Queries.

If a process p initializes a query and that query is deleted due to a fault, then p could, in principle, wait forever for the query to return. If p suspects that its query has been deleted, it sends out a *probe wave*, either to the left or the right, whichever is the direction of the missing query, and if it receives back the *report* that there is no query, it returns to the resting state, allowing it to initiate a new query if necessary.

4 Formal Definition of LE(k)

Relative Addresses.

If p is a process, we write p^+ and p^- for the right and left neighbors of p, respectively. Similarly, for any non-negative integer i, p^i and p^{-i} are the processes i positions to the right, respectively left, of p. If $q = p^i$ for any integer i, we say that i is p's *relative address* of q.

Types.

We need some particular data types:

1. *Relative address*. A variable of this type is either $\perp$ or a signed integer which is the relative address of some process. The range of possible values of a relative address is $\{-6k-1, \ldots, 6k+1\}$.
2. *Status type*, which has the values *resting*, *requesting*, *query_L*, *query_R*, and *query_returned*.
3. *Query type*. A variable x of query type is either $\perp$, or an ordered quadruple: $(x.\mathit{home}, x.\mathit{candidate}, x.\mathit{vote\_count}, x.\mathit{no\_trail\_copy})$ where $x.\mathit{home}$ and $x.\mathit{candidate}$ are relative addresses, $x.\mathit{vote\_count}$ is a non-negative integer, and $x.\mathit{no\_trail\_copy}$ is Boolean. The field $x.\mathit{no\_trail\_copy}$ indicates that there is no copy of x trailing it in the query path.
4. *Flag type* = $\{\mathsf{a}, \mathsf{b}\}$. Flags determine whether a given process has the privilege to initiate a query.

Variables of LE(k).

We now give a formal list of the variables of a process p:

1. $p.\mathit{vote}$, of relative address type, but restricted to the range $\{-3k \ldots, 3k\}$. If not $\perp$, this is the relative address of the process that p "believes" (perhaps falsely) to be the leader.
2. $p.\mathit{rumor}$, of relative address type, either $\perp$ or in $\{-3k \ldots, 3k\}$. The set of all rumor variables of all processes constitute the *rumor track*. If $p.\mathit{rumor} = i$, p "suspects" that p^i is the leader.
3. $p.\mathit{L\_deny}$ and $p.\mathit{R\_deny}$, of relative address type, but restricted to the range $\{-3k \ldots, 3k\}$. These variables for all processes constitute the two *denial tracks*, along which *denial waves* move in the leftward or rightward direction, respectively. If $p.\mathit{L\_deny} = i$ or $p.\mathit{R\_deny} = i$, that indicates that some process has sent a query to count votes and has reported that p^i does not have a majority of votes. The purpose of denials is to kill rumors. Two directions are needed to avoid head-on collisions of denial waves.
4. $p.\mathit{track}$_I, $p.\mathit{track}$_II, and $p.\mathit{track}$_III, of query type. There can be up to three queries hosted by p at any given step, one moving left on the first query track, one moving right on the second track, and one moving left on the third track. A process p can initiate a query, placing the token on the first track, and when the query has completed its path, it will return to p again on the third track, carrying a count of the number of votes for its candidate.

5. p.*status*, the *status* of p, of status type. We list the meanings of these status values:

   (a) p.*status* $=$ *requesting* means that p wants to initiate a query, but has not yet done so.

   (b) p.*status* $=$ *query_L* means that there is an outstanding query whose home is p, and which is to the left of p or is at p.*track*_I. That is, the query is either on the first query track, or is on the second query track but has not yet reached the second query track variable of its home, *i.e.,* p.*track*_II.

   (c) p.*status* $=$ *query_R* means that there is an outstanding query whose home is p, and which has reached p.*track*_II but has not yet reached p.*track*_III. That is, the query is on the second or third track, and is to the right of p or at p.*track*_II.

   (d) p.*status* $=$ *query_returned* means that there is an outstanding query whose home is p, and that this query has already traversed its query path and has returned to p.*track*_III, but has not yet been processed (concluded). When the query is concluded, p.*status* will change to *resting*.

   (e) p.*status* $=$ *resting* means none of the above.

6. p.*L_flag*, p.*R_flag* $\in \{\mathsf{a}, \mathsf{b}\}$, the left and right flags of p, of flag type. We can think of there being n *query resources*, one for each adjacent pair of processes. Each of those resources is shared by two neighboring processes, and each process shares two resources. Just as in the dining philosophers' problem, a process must “hold" both of its shared resources to have an outstanding query. A resource can be held by only one of its two neighboring processes.

   The rule is that the resource which is shared by neighbors p and p^+ is held by p if p.*R_flag* $= p^+$.*L_flag*; otherwise it is held by p^+. We say that p has the *query privilege* if p holds both resources. It is impossible for two neighboring process to both have the query privilege.

   The default value of each flag variable is a. Thus, in a final configuration, every process holds its right resource but not its left, and hence no process has the query privilege.

7. p.*L_probe*, p.*R_probe*, p.*L_report*, and p.*R_report*, of relative address type. If p.*L_probe* $= i$, then p^i is sending a “probe" to determine whether there is a query whose home process is p^i, and if p.*L_report* $= i$, that query was not found, and a report wave is returning to p^i to report that the query was not found. Right probe and report waves are similar. p.*L_probe* and p.*L_report* are either $\perp$ or non-negative. p.*R_probe* and p.*R_report* are either $\perp$ or non-positive.

8. p.*num_L_null*, p.*num_R_null*, non-negative integers, in the range $\{0, \ldots 6k+2\}$.

   The correct value of p.*num_L_null* (p.*num_R_null*) is the number of consecutive processes to the left (right) of p, counting p itself, that have no query, no probe wave, and no report wave.

   The maximum distance from a query to p is $6k+1$, but we must be able to count up to $6k+2$ to accommodate p itself.

The Query Path.

For each process p and each query initiated by p, there is a fixed sequence of query variables that the query must pass through; we call this the *path* of the query, or sometimes, the *query path*. If p^i is the candidate of the query, then $|i| \leq 3k$, and the query path is the following sequence: p.*track*_I, p^-.*track*_I $\ldots$ p^{i-3k-1}.*track*_I, p^{i-3k-1}.*track*_II, p^{i-3k}.*track*_II $\ldots$ p^{i+3k+1}.*track*_II, p^{i+3k+1}.*track*_III, p^{i+3k}.*track*_III $\ldots$ p^+.*track*_III, p.*track*_III.

Figure 1 shows an example of a query path in a case where $k = 2$ and $i = 2$. Each process is represented vertically in the figure, and the four boxes represent the first and second query variables of the process, the vote of the process, and the third query variables of the process. Each vote is a relative address, and the votes for p^2 are enclosed in ovals.

For each query variable on the query path, an ordered triple is shown, consisting of the *home*, *candidate*, and *vote_count* of the query at the time its token is at that variable. Other variables are not shown.

When the query reaches p along the third track, it informs p that the candidate has nine votes, a majority of the thirteen possible in the interval of relevance. Process p will then change its vote to 2 to vote for the candidate.

It may not be clear why the query track has radius $3k+1$ around the candidate instead of merely $3k$; after all, the smaller radius would suffice to count all votes for the candidate. We do this in order to prevent a query from changing tracks at its home process, since that would complicate the code.

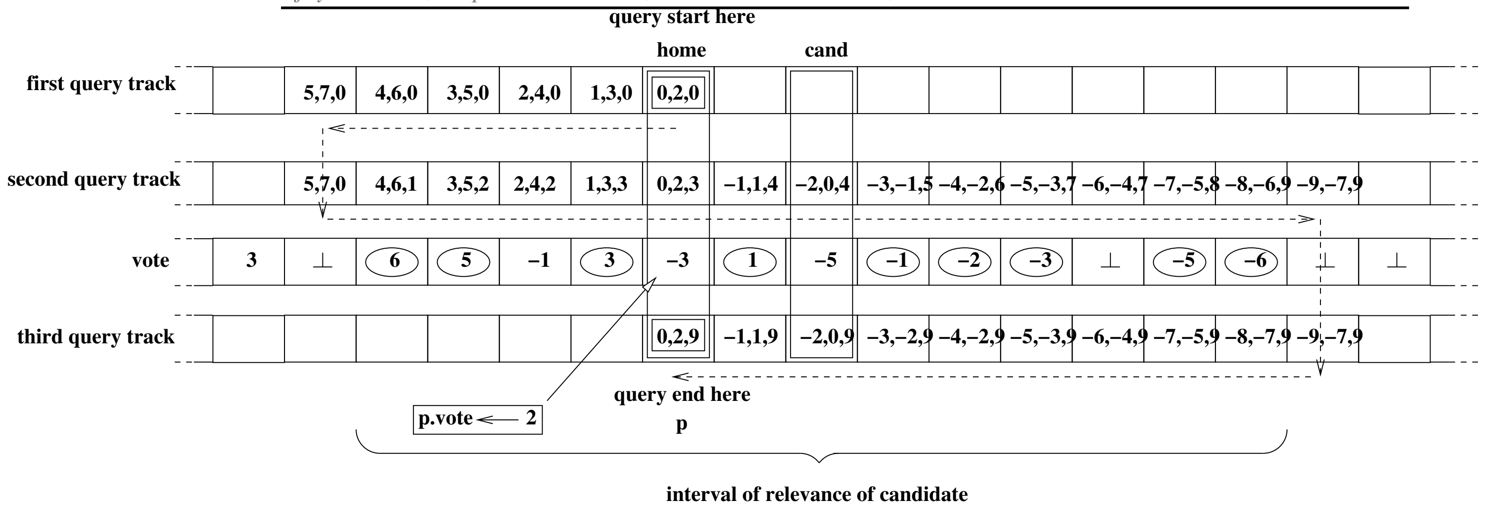

Figure 1: Query Path.

Functions of LE(k).

1. *Satisfied*(p), Boolean, p is *satisfied*. This predicate is true if $p.rumor = p.vote$. If *Satisfied*(p), then p will not initiate a query. In a legitimate configuration, all processes are satisfied.

2. *Suggestion*(p), relative address type. $\mathit{Suggestion}(p) = \begin{cases} \bot \text{ if } p.rumor = p.vote \\ p.vote \text{ if } p.rumor = \bot \\ p.rumor \text{ otherwise} \end{cases}$

   The process p can initiate a query with candidate *Suggestion*(p), unless *Suggestion*$(p) = \bot$.

3. *Querying*$(p) \equiv p.status \in \{query\_L, query\_R, query\_returned\}$, *i.e.,* p has an outstanding query.

4. *Need_to_Query*(p), Boolean, which is true if *Suggestion*$(p) \neq \bot$ and if neither p nor either of its neighbors has an outstanding query. Formally: *Need_to_Query*$(p) \equiv$ *Suggestion*$(p) \neq \bot \wedge \neg$*Querying*$(p^-) \wedge \neg$*Querying*$(p)$ $\wedge \neg$*Querying*(p^+)

5. *Query_Privileged*$(p) \equiv p.R\_flag = p^+.L\_flag \wedge p.L\_flag \neq p^-.R\_flag$, Boolean, meaning that p holds both of its resources.

6. If x is a variable of query type, namely the variable $p.track$_I, $p.track$_II, or $p.track$_III, where p is its host process, and if $x \neq \bot$, we can define *Pred*(x) and *Succ*(x), the query variables which precede and succeed x in its query track, as shown in Figure 1.

   Let $j = x.candidate$.

   (a) If $x = p.track$_I and $j \geq -3k$, then *Pred*$(x) = p^+.track$_I.
   (b) If $x = p.track$_II and $j = 3k + 1$, then *Pred*$(x) = p.track$_I.
   (c) If $x = p.track$_II and $j \leq 3k$, then *Pred*$(x) = p^-.track$_II.
   (d) If $x = p.track$_III and $j = -3k - 1$, then *Pred*$(x) = p.track$_II.
   (e) If $x = p.track$_III and $j \geq -3k$, then *Pred*$(x) = p^+.track$_III.
   (f) In all other cases, *Pred*$(x) = \bot$.
   (g) If $x = p.track$_I and $j \leq 3k$, then *Succ*$(x) = p^-.track$_I.
   (h) If $x = p.track$_I and $j = 3k + 1$, then *Succ*$(x) = p.track$_II.
   (i) If $x = p.track$_II and $j \geq -3k$, then *Succ*$(x) = p^+.track$_II.
   (j) If $x = p.track$_II and $j = -3k - 1$, then *Succ*$(x) = p.track$_III.

(k) If $x = p.\mathit{track}\_\mathrm{III}$ and $j \le 3k$, then $\mathit{Succ}(x) = p^-.\mathit{track}\_\mathrm{III}$.

(l) In all other cases, $\mathit{Succ}(x) = \bot$.

7. We define Boolean functions $\mathit{Has\_trailing\_copy}(x)$ and $\mathit{Has\_forward\_copy}(x)$ for any query variable x. $\mathit{Has\_trailing\_copy}(x)$ means that x and $\mathit{Pred}(x)$ have the same home process, while $\mathit{Has\_forward\_copy}(x)$ means that x and $\mathit{Succ}(x)$ have the same home process. More formally, let p be the host of x and q the host of $y = \mathit{Pred}(x)$. Then $\mathit{Has\_trailing\_copy}(x)$ if and only if neither x nor y is $\bot$, and one of the following conditions holds:

   (a) $q = p^-$ and $y.\mathit{home} = x.\mathit{home} + 1$

   (b) $q = p$ and $y.\mathit{home} = x.\mathit{home}$

   (c) $q = p^+$ and $y.\mathit{home} = x.\mathit{home} - 1$

   Similarly, let r be the host of $z = \mathit{Succ}(x)$. Then $\mathit{Has\_forward\_copy}(x)$ if and only if neither x nor z is $\bot$, and one of the following conditions holds:

   (d) $r = p^-$ and $z.\mathit{home} = x.\mathit{home} + 1$

   (e) $r = p$ and $z.\mathit{home} = x.\mathit{home}$

   (f) $r = p^+$ and $z.\mathit{home} = x.\mathit{home} - 1$

Actions of LE(k).

We define and explain the actions of LE(k). More details are given in Section 5.

1. **Request Privilege.** If $p.\mathit{status} = \mathit{resting}$ and $\mathit{Need\_to\_Query}(p)$, then $p.\mathit{status} \leftarrow \mathit{requesting}$. This action informs the neighbors of p that p needs both resources.

2. **Yield Privilege.** A process can yield the query privilege because it does not want to initiate a query, or because one of its neighbors has a higher priority claim on the privilege.

   (a) A process might withdraw its request for the query privilege because it becomes satisfied, or because a neighbor process initiates a query.

   If $p.\mathit{status}{=}\mathit{requesting}$ and $\neg\mathit{Need\_to\_Query}(p)$, then $p.\mathit{status} \leftarrow \mathit{resting}$. This reverses the action Request Privilege.

   (b) To prevent contention, we allow a process to pass a token query flag to its neighbor, but not to seize the token. If p holds its left query flag token, *i.e.,* $p.\mathit{L\_flag} \ne p^-.\mathit{R\_flag}$, then p passes the token to p^- by reversing its left flag, *i.e.,* changing a to b or b to a, provided at least one of the following conditions holds.

   i. $p.\mathit{status} = \mathit{resting}$

   ii. $p.\mathit{status} = p^-.\mathit{status} = \mathit{requesting}$ and $\mathit{Suggestion}(p^-) > \mathit{Suggestion}(p)$

   If p holds its right query flag token, *i.e.,* $p.\mathit{R\_flag} = p^+.\mathit{L\_flag}$, then p passes the token to p^+ by reversing its right flag, provided at least one of the following conditions holds.

   iii. $p.\mathit{status} = \mathit{resting}$ and $p^+.\mathit{status} = \mathit{requesting}$

   iv. $p.\mathit{status} = \mathit{resting}$ and $p.\mathit{R\_flag} = \mathsf{b}$

   v. $p.\mathit{status} = p^+.\mathit{status} = \mathit{requesting}$ and $\mathit{Suggestion}(p^+) \ge \mathit{Suggestion}(p)$

   Note the asymmetry between the conditions for yielding the left and right flags. If one of two neighboring processes, say p and p^+, is satisfied, it yields to the other. But if both are unsatisfied, only one can have the resource, and the rules above determine which.

   In particular, if one of the two currently has an outstanding query, then it has the resource, and the other process cannot have an outstanding query, since it lacks the resource, and it also yields its other resource to its other neighbor if that neighbor is unsatisfied.

   If two neighbors both have status *requesting*, the one with the smaller value of *Suggestion* will yield to the other, and if they have equal values of *Suggestion*, the one on the left yields to the one on the right. That rule would lead to deadlock if all processes had the same value of *Suggestion*, but, as we will see later, this is impossible.

Another potential problem is *livelock*. If both neighboring processes are satisfied, they could yield their common resource back and forth to each other forever. Rule (iv) prevents that. If both processes have status *resting*, and $p.R\_flag = p^+.L\_flag = \mathsf{a}$, then p does not yield the resource. Thus, all flags have the value a in a final configuration.

3. **Interrupt Query.** If $Querying(p)$ and $\neg Query\_Privileged(p)$, then $p.status \leftarrow resting$. This action is enabled only if either p or a neighbor faults.

4. **Initiate Query.** If $p.status = requesting$, $Query\_Privileged(p)$, and $p.track\_\mathrm{I} = \bot$, then p initiates a query by executing the following assignments:

   (a) $p.track\_\mathrm{I} \leftarrow (0, Suggestion(p), 0, \textsc{true})$

   (b) $p.status \leftarrow query\_L$.

   The purpose of the query is to report to p how many processes' votes are for the candidate, $Suggestion(p)$.

5. **Copy Query Forward.** A query advances along its track by copying the live query token to its successor variable, and then deleting the old token. The basic paradigm is that a token can be copied forward when there is no trailing copy, and a token can be deleted if there is a forward copy.

   There are three actions required to implement advancement of a query. The first of these is Copy Query Forward. The other two are Delete Trailing Query and Mark Sole Query, given below.

   Let x, y be query variables such that the host of x is a process p, the host of y is q, and $x = Succ(y)$. (In particular, that implies that $y \neq \bot$.) Let $i = y.home$ and $j = y.candidate$. We break the definition of Action Copy Query Forward into three cases, depending on whether $p^+ = q$, $p^- = q$, or $p = q$,

   (a) If $p^+ = q$, then the guard of the action is the conjunction of the following conditions:

      i. $y.no\_trail\_copy$

      ii. $x = \bot$

      while the statement of the action is as follows.

      iii. $x.home \leftarrow i + 1$

      iv. $x.candidate \leftarrow j + 1$

      v. $x.vote\_count \leftarrow y.vote\_count$

      vi. $x.no\_trail\_copy \leftarrow \textsc{false}$

      Conditions (i) and (ii) mean that y holds the only current token for that query, while (vi) is executed to indicate that x and y each now has a token for the query.

      Vote counts are not changed as a query traverses either the first or third query track.

   (b) If $p^- = q$, then the guard of the action is the conjunction of the following conditions:

      i. $y.no\_trail\_copy$

      ii. $x = \bot$

      while the statement of the action is as follows.

      iii. $x.home \leftarrow i - 1$

      iv. $x.candidate \leftarrow j - 1$

      v. $x.vote\_count \leftarrow \begin{cases} y.vote\_count + 1 \text{ if } p.vote = j - 1 \\ y.vote\_count \text{ otherwise} \end{cases}$

      vi. $x.no\_trail\_copy \leftarrow \textsc{false}$

      This case is similar to (a) above, except that we must count all votes for the candidate as we move through the middle query track.

(c) If $p = q$, then the guard of the action is the conjunction of the following conditions:

i. $y.\mathit{no\_trail\_copy}$

ii. $x = \bot$

while the statement of the action is as follows.

iii. $x.\mathit{home} \leftarrow i$

iv. $x.\mathit{candidate} \leftarrow i$

v. $x.\mathit{vote\_count} \leftarrow \begin{cases} 0 \text{ if } x = p.\mathit{track}\_\text{II} \\ y.\mathit{vote\_count} \text{ if } x = p.\mathit{track}\_\text{III} \end{cases}$

vi. $x.\mathit{no\_trail\_copy} \leftarrow \text{FALSE}$

6. **Delete Trailing Query.**

If x is a query variable, $\neg\mathit{Has\_trailing\_copy}(x)$, and $\mathit{Has\_forward\_copy}(x)$, then $x \leftarrow \bot$.

If there is a copy of x in $\mathit{Succ}(x)$, then x is not a live query. In order to delete x, we need to know that there it has no trailing copy, since otherwise that trailing copy would become a rogue query.

7. **Mark Sole Query.**

If x is a query variable, $\neg\mathit{Has\_forward\_copy}(x)$, and $\neg x.\mathit{Has\_trailing\_copy}$, then $x.\mathit{no\_trail\_copy} \leftarrow \text{TRUE}$.

If neither $\mathit{Succ}(x)$ nor $\mathit{Pred}(x)$ contains a copy of x, then $x.\mathit{no\_trail\_copy}$ is set to true, indicating that there is no copy of x.

8. **Query Right.** If $p.\mathit{track}\_\text{II}.\mathit{home} = 0$ and $p.\mathit{status} = \mathit{query\_L}$, then $p.\mathit{status} \leftarrow \mathit{query\_R}$. This happens when the query moves from being to the left of p to being to the right of p.

9. **Query Return.** If $p.\mathit{track}\_\text{III}.\mathit{home} = 0$ and $p.\mathit{status} = \mathit{query\_R}$, then $p.\mathit{status} \leftarrow \mathit{query\_returned}$. This happens when the query returns to its home process.

10. **Conclude Query.** When a query has returned to its home p, after traversing the query loop, $p.\mathit{status} = \mathit{query\_returned}$. The information collected by the query is then processed, and finally, the query is deleted.

The process p will not conclude the query until there is no trailing copies of the query and both of its denial tracks are empty. The reason is that it is possible that p will need to initiate denial waves when it concludes the query. Thus, the guard for the action Conclude Query is the conjunction of the following conditions:

- $p.\mathit{status} = \mathit{query\_returned}$
- $p.\mathit{track}\_\text{III}.\mathit{no\_trail\_copy}$
- $p.\mathit{L\_deny} = p.\mathit{R\_deny} = \bot$

What happens next depends on whether the query has returned "positive" news, namely that the candidate has the votes of the majority of processes in its interval of relevance, or "negative" news, *i.e.,* it has a minority of votes.

Let i be the candidate of the query, *i.e.,* $p.\mathit{track}\_\text{III}.\mathit{candidate}$. If the news is positive, *i.e.,* $p.\mathit{track}\_\text{III}.\mathit{vote\_count} \geq 3k + 1$, then p starts the rumor that i is the leader, initiates denial of any competing rumor, and changes its vote to i. More formally stated, p executes the following three statements:

(a) If $p.\mathit{rumor} \neq \bot$ and $p.\mathit{rumor} = j \neq i$, then $p.\mathit{L\_deny} \leftarrow j$ and $p.\mathit{R\_deny} \leftarrow j$.

(b) $p.\mathit{vote} \leftarrow i$.

(c) $p.\mathit{rumor} \leftarrow i$.

On the other hand, if the news is negative, *i.e.,* $p.\mathit{track}\_\text{III}.\mathit{vote\_count} \leq 3k$, p deletes the rumor, if any, that i is the leader, initiates denial that i is the leader, and deletes its vote for i, if any. More formally stated, p executes the following three statements:

(d) If $p.\mathit{rumor} = i$, then $p.\mathit{rumor} \leftarrow \bot$. Otherwise, the rumor is unchanged.

(e) $p.L\_deny \leftarrow i$ and $p.R\_deny \leftarrow i$.

(f) If $p.vote = i$, then $p.vote \leftarrow \bot$. Otherwise, the vote is unchanged.

In either case, p then deletes the query token and returns to resting status by executing the following statements:

(g) $p.track\_\text{III} \leftarrow \bot$.

(h) $p.status \leftarrow resting$.

11. **Erroneous Query Return.** Because of faults, the home process may not be expecting a query. In this case, the unwelcome query is simply deleted. The guard of this action is the conjunction of the following conditions:

- $p.track\_\text{III}.home = 0$
- $p.track\_\text{III}.no\_trail\_copy$
- $p.status \in \{query\_L, resting\}$

The statement of the action is $p.track\_\text{III} \leftarrow \bot$.

12. **Advance Denial.** To avoid endless back and forth propagation of denials, we allow every denial wave to move in only one direction. Thus, a process p can copy a denial from its right neighbor's left denial variable or from its left neighbor's right denial variable. A denial overwrites another denial that has lower priority, where that priority is the same as the priority of rumors.

(a) The action $p.L\_deny \leftarrow i$ is enabled if the following conditions hold:

i. $-3k + 1 \le i \le 3k$
ii. $(p.L\_deny = \bot) \vee (p.L\_deny < i)$
iii. $p^+.L\_deny = i - 1$

(b) The action $p.R\_deny \leftarrow i$ is enabled if the following conditions hold:

iv. $-3k \le i \le 3k - 1$
v. $(p.R\_deny = \bot) \vee (p.R\_deny < i)$
vi. $p^-.R\_deny = i + 1$

Note that, although a rumor may not move to a process occupied by a matching denial, a denial can move to a process occupied by a matching rumor. That rumor will then be deleted by Action Delete Rumor, below.

13. **Delete Trailing Denial.** A denial is deleted if it has been copied forward or has reached the end of its interval of relevance, and there is no following copy of the denial.

(a) If $p.L\_deny = i$, then the action $p.L\_deny \leftarrow \bot$ is enabled if the following conditions hold:

i. $(i = 3k) \vee (p^-.L\_deny = i + 1)$.
ii. $p^+.L\_deny \neq i - 1$.

(b) If $p.R\_deny = i$, then the action $p.R\_deny \leftarrow \bot$ is enabled if the following conditions hold:

iii. $(i = -3k) \vee (p^+.R\_deny = i - 1)$.
iv. $p^-.R\_deny \neq i + 1$.

A denial wave is temporary; once it reaches the end of its interval of relevance, it disappears.

Conditions (ii) and (iv) prevent deletion of a denial token if it has a trailing copy, since that trailing copy would otherwise become an extra denial wave.

14. **Spread Rumor.** A positive rumor spreads both leftward and rightward until it reaches the end of its interval of relevance, or until it encounters a denial of that rumor on either of the two denial tracks. If neighboring processes have different rumors, the larger rumor can overwrite the smaller.

Formally, the guard for the action $p.rumor \leftarrow i$ is the conjunction of the following conditions.

   (a) $-3k \leq i \leq 3k$.
   (b) $p.\mathit{rumor} = \perp$ or $p.\mathit{rumor} < i$.
   (c) $(p^+.\mathit{rumor} = i - 1) \vee (p^-.\mathit{rumor} = i + 1)$.
   (d) $(p.\mathit{L\_deny} \neq i) \wedge (p.\mathit{R\_deny} \neq i)$.

   Denial waves cause rumors to be deleted; thus, (d) prevents a rumor from spreading to a process with a denial wave with the same parameter.

15. **Delete Rumor.** A denial causes a rumor to be deleted.

    If $p.\mathit{rumor} = i$ and either $p.\mathit{L\_deny} = i$ or $p.\mathit{R\_deny} = i$, then $p.\mathit{rumor} \leftarrow \perp$.

16. **Update Null Counters.** The correct value of $p.\mathit{num\_L\_null}$ ($p.\mathit{num\_R\_null}$) is the number of consecutive processes to the left (right) of p, including p, which have no query token, no probe token, and no report token. Since tokens farther than $6k+1$ away from p cannot have p as their home, the maximum value of $p.\mathit{num\_L\_null}$ ($p.\mathit{num\_R\_null}$) is set to $6k+2$.

    (a) If any of the three query variables of p is not $\perp$, or if either of the two probe variables of p is not $\perp$, or if either of the two report variables of p is not $\perp$, then both null counters of p are set to zero, *i.e.,* $p.\mathit{num\_L\_null} \leftarrow 0$ and $p.\mathit{num\_R\_null} \leftarrow 0$.
    (b) Otherwise, $p.\mathit{num\_L\_null} \leftarrow \min\{p^-.\mathit{num\_L\_null} + 1, 6k+2\}$ and $p.\mathit{num\_R\_null} \leftarrow \min\{p^+.\mathit{num\_R\_null} + 1, 6k+2\}$.

    In action (a), we consider probes and reports, since otherwise a process which has a null counter of value $6k+2$ would generate many probe waves in succession instead of just one, when only one is needed.

17. **Initiate Probe.** If $p.\mathit{status} = \mathit{query\_L}$ ($\mathit{query\_R}$) but $p.\mathit{num\_L\_null}$ ($p.\mathit{num\_R\_null}$) is $6k+2$, indicating that there is no outstanding query to its left (right), p initiates a left (right) probe to verify that its query is no longer alive.

    (a) If $p.\mathit{status} = \mathit{query\_L}$ and $p.\mathit{num\_L\_null} = 6k+2$, then $p.\mathit{L\_probe} \leftarrow 0$ and $p.\mathit{num\_L\_null} \leftarrow 0$.
    (b) If $p.\mathit{status} = \mathit{query\_R}$ and $p.\mathit{num\_R\_null} = 6k+2$, then $p.\mathit{R\_probe} \leftarrow 0$ and $p.\mathit{num\_R\_null} \leftarrow 0$.

    A left (right) probe is initiated if p believes it should have an outstanding query to its left (right), but the value of $p.\mathit{num\_L\_null}$ ($p.\mathit{num\_R\_null}$) indicates that this query does not exist. The value of $p.\mathit{num\_L\_null}$ ($p.\mathit{num\_R\_null}$) is set to zero to prevent p from immediately sending a second probe.

18. **Advance Probe.** A left or right probe advances for $6k+1$ steps, or until it finds a live query token with the correct home.

    (a) If $(p.\mathit{L\_probe} = \perp) \vee (p.\mathit{L\_probe} \leq p^+.\mathit{L\_probe})$, $p^+.\mathit{L\_probe} + 1 \notin \{p.\mathit{track\_}\mathrm{I}.\mathit{home}, p.\mathit{track\_}\mathrm{II}.\mathit{home}, p.\mathit{track\_}\mathrm{III}.\mathit{home}\}$, and $0 \leq p^+.\mathit{L\_probe} \leq 6k$, then $p.\mathit{L\_probe} \leftarrow p^+.\mathit{L\_probe} + 1$.
    (b) If $(p.\mathit{R\_probe} = \perp) \vee (p^-.\mathit{R\_probe} - 1 > p.\mathit{R\_probe})$, $p^-.\mathit{R\_probe} - 1 \notin \{p.\mathit{track\_}\mathrm{I}.\mathit{home}, p.\mathit{track\_}\mathrm{II}.\mathit{home}, p.\mathit{track\_}\mathrm{III}.\mathit{home}\}$, and $-6k \leq p^-.\mathit{R\_probe} \leq 0$, then $p.\mathit{R\_probe} \leftarrow p^-.\mathit{R\_probe} - 1$.

    To prevent deadlock on the probe tracks, a probe of higher parameter will overwrite a probe ahead of it. A probe will not move to a process which contains a query with the same home process as the probe's parameter.

19. **Delete Probe.** Probes advance by forward-copy/rear delete. Also, if a probe meets a live query whose home is the parameter of the probe, the probe will be deleted.

    (a) If $p.\mathit{L\_probe} \neq \perp$, $p^+.\mathit{L\_probe} = \perp$, and $p^-.\mathit{L\_probe} = p.\mathit{L\_probe} + 1$, then $p.\mathit{L\_probe} \leftarrow \perp$.
    (b) If $p.\mathit{R\_probe} \neq \perp$, $p^-.\mathit{R\_probe} = \perp$, and $p^+.\mathit{R\_probe} = p.\mathit{R\_probe} - 1$, then $p.\mathit{R\_probe} \leftarrow \perp$.

    Actions (a) and (b) cause normal deletion of a trailing probe token.

(c) If $p.\mathit{L\_probe} + 1 \in \{p^-.\mathit{track\_}\mathrm{I}.\mathit{home}, p^-.\mathit{track\_}\mathrm{II}.\mathit{home}, p^-.\mathit{track\_}\mathrm{III}.\mathit{home}\}$ and $p^+.\mathit{L\_probe} \neq p.\mathit{L\_probe} - 1$, then $p.\mathit{L\_probe} \leftarrow \perp$.

(d) If $p.\mathit{R\_probe} - 1 \in \{p^+.\mathit{track\_}\mathrm{I}.\mathit{home}, p^+.\mathit{track\_}\mathrm{II}.\mathit{home}, p^+.\mathit{track\_}\mathrm{III}.\mathit{home}\}$ and $p^-.\mathit{R\_probe} \neq p.\mathit{R\_probe} + 1$, then $p.\mathit{R\_probe} \leftarrow \perp$.

Actions (c) and (d) cause deletion of a probe token when it encounters the query it has been seeking.

(e) If $p.\mathit{L\_probe} = p.\mathit{L\_report} = 6k + 1$, then $p.\mathit{L\_probe} \leftarrow \perp$.

(f) If $p.\mathit{R\_probe} = p.\mathit{R\_report} = -6k - 1$, then $p.\mathit{R\_probe} \leftarrow \perp$.

Actions (e) and (f) cause deletion of a probe token when it reaches the end of the search interval. However, it cannot be deleted until the report token has been initialized, by Action Initiate Report(a) or (b), below.

20. **Initiate Report.** When a probe has traveled distance $6k + 1$ without finding a live query with the correct home, it sends a report back to the home process.

    (a) If $p.\mathit{L\_probe} = 6k + 1$ and $p.\mathit{L\_report} = \perp$, then $p.\mathit{L\_report} \leftarrow 6k + 1$.

    (b) If $p.\mathit{R\_probe} = -6k - 1$ and $p.\mathit{R\_report} = \perp$, then $p.\mathit{R\_report} \leftarrow -6k - 1$.

    Action (a) must execute before Action Delete Probe(e) can execute, while Action (b) must execute before Action Delete Probe(f) can execute.

21. **Advance Report.** A report moves back toward its home process.

    (a) If $(p.\mathit{L\_report} = \perp) \vee (p.\mathit{L\_report} < p^-.\mathit{L\_report} - 1)$, then $p.\mathit{L\_report} \leftarrow p^-.\mathit{L\_report} - 1$

    (b) If $(p.\mathit{R\_report} = \perp) \vee (p.\mathit{R\_report} < p^+.\mathit{R\_report} + 1)$, then $p.\mathit{report} \leftarrow p^+.\mathit{report} + 1$

22. **Delete Report.**

    (a) If $p.\mathit{L\_report} \neq \perp$, $p^-.\mathit{L\_report} = \perp$, and $p^+.\mathit{L\_report} = p.\mathit{L\_report} - 1$ then $p.\mathit{L\_report} \leftarrow \perp$.

    (b) If $p.\mathit{R\_report} \neq \perp$, $p^+.\mathit{R\_report} = \perp$, and $p^-.\mathit{R\_report} = p.\mathit{report} + 1$ then $p.\mathit{R\_report} \leftarrow \perp$.

    In Advance Report(a) and (b), and in Delete Report(a) and (b), a report advances by forward-copy/rear-delete. If a report has been copied forward, it can be deleted, except that any trailing copy must be deleted first.

    (c) If $p.\mathit{L\_report} \in \{p.\mathit{track\_}\mathrm{I}.\mathit{home}, p.\mathit{track\_}\mathrm{II}.\mathit{home}, p.\mathit{track\_}\mathrm{III}.\mathit{home}\}$ then $p.\mathit{L\_report} \leftarrow \perp$.

    (d) If $p.\mathit{R\_report} \in \{p.\mathit{track\_}\mathrm{I}.\mathit{home}, p.\mathit{track\_}\mathrm{II}.\mathit{home}, p.\mathit{track\_}\mathrm{III}.\mathit{home}\}$ then $p.\mathit{R\_report} \leftarrow \perp$.

    In (c) and (d), a report is deleted if it meets a query with the same home process.

23. **Close Report.** If p sends a probe which finds a query whose home is p, then no report returns to p.

    However, if a report returns to its home process p, then that process knows that its query is missing, and its status changes to *resting*. It will later initiate a replacement query, if necessary.

    More formally, if $p.\mathit{L\_report} = 0$ and $p^-.\mathit{L\_report} \neq 1$ then $p.\mathit{L\_report} \leftarrow \perp$ and $p.\mathit{status} \leftarrow \mathit{resting}$, while if $p.\mathit{R\_report} = 0$ and $p^+.\mathit{R\_report} \neq -1$, then $p.\mathit{R\_report} \leftarrow \perp$ and $p.\mathit{status} \leftarrow \mathit{resting}$.

5 Detailed Overview of LE(k)

We analyze LE(k) as the interleaving of several *themes*, each of them can be given by a high level definition, and whose implementation can involve several actions.

Voting.

Our first theme concerns votes. There are exactly $6k + 1$ processes which can vote for ℓ, the processes in the interval of relevance of ℓ. We need to ensure that, at any given configuration, a majority of these are voting for ℓ, and that the number of processes voting for any process other than ℓ is never greater than $3k$.

Initially, ℓ has $6k + 1$ votes. If a process p faults, then $p.vote$ could change. However, the votes of up to two other processes could be changed as a result of that fault.

A process q will change its vote if a query whose home process is q reports that some other process has a majority of votes. When p faults, if x is a query whose home is q and is in the second or third query variable of its host, the fault could change the value of $x.vote\_count$. Thus, q might erroneously change its vote when it closes that query.

A change in the vote count of a query in the first track has no effect, since $x.vote\_count \leftarrow 0$ whenever a query x moves from the first track to the second. Since the fault could alter $p.track$_II and $p.track$_III, two additional processes could change their votes later.

Since there can be at most $3k$ instances of a process voting for a false leader during the computation, we conclude that ℓ never has fewer than $3k + 1$ votes, and no other candidate ever has more than $3k$ votes.

5.0.1 Rumors.

Our second theme is *rumors*. When a process changes its vote, or has its vote confirmed when it closes a query, it initiates a rumor whose parameter is the same as its vote. A rumor floods the interval of relevance of its parameter process, but if there are competing rumors, the rumor for the rightmost of those processes dominates. When $p.rumor \leftarrow i$, where $p.vote \neq i$, then p does not change its vote to i immediately; instead, it may initiate a query to count the votes for p^i. No process ever changes its vote based on rumor alone.

5.0.2 Denials.

There are two denial tracks. The parameter of each denial is the relative address of a process. Denial waves move leftward on one track or rightward on the other. A denial wave deletes itself as it passes, and deletes itself when it reaches one end of the interval of relevance of its parameter process. When a denial meets a rumor with the same parameter, the rumor is deleted. However, the denial wave is not deleted. When a process concludes that a given candidate process is not the leader, it can generate denial waves in both directions. This action has the effect of deleting all rumors for that candidate.

Queries.

If a process p it not satisfied, it can initiate a query whose candidate is p^i, where $i = \mathit{Suggestion}(p)$. The query moves by forward-copy/rear-delete along the query path, which begins at $p.track$_I and ends at $p.track$_III. The query path includes $q.track$_II for all q in the interval of relevance of p^i, namely p^j for all $i - 3k \leq j \leq i + 3k$. As the query moves along the second track, it counts votes for p^i and stores the total as *vote_count*.

The mechanism by which a query moves along its path permits at most one trailing copy of the query at any time. The purpose of this rule is to prevent the formation of additional rogue queries: if there were three copies of the query in a row, a fault to the middle process could cause the other two to become separate queries.

We remark that none of the other kinds of waves that spread by copy-forward/delete-rear follow this rule. Denial waves, probe waves, and report waves can have more than one trailing copy behind the live token.

When the query returns to p, if it reports that p^i has received at least $3k+1$ votes, then $p.vote \leftarrow i$, and $p.rumor \leftarrow i$. If $p.rumor = j \neq i$, p also initiates denial waves which deny that p^j is the leader.

If, on the other hand, the query reports that p^i not the leader, p initializes denial waves which deny that p^i is the leader. Furthermore, if $p.vote = i$, then $p.vote \leftarrow \perp$.

Status and the Query Privilege.

The variable $p.status$ reveals whether p is *querying*, *i.e.,* currently the home process of an outstanding query, and if so, whether the query is to the left or the right of p, and if not, whether p wants to initiate a query.

To avoid congestion in the query tracks, LE(k) never allows two neighboring processes to be querying simultaneously. There is a resource between each pair of adjacent processes, and a process must have both adjacent resources to initiate a query, and must hold onto both while it is querying.

The actions of LE(k) never permit a process to take a resource away from a neighbor (although a fault could cause that to happen), but do permit it to yield a resource it is holding by reversing the appropriate flag. Whenever $p.\mathit{status} = \mathit{resting}$, it yields both resources, except that p does not yield its right resource to p^+ if $p^+.\mathit{status} = \mathit{resting}$ and both flags are a. This exception to the usual rule permits LE(k) to be silent when a legitimate configuration is achieved, since otherwise the resource would be passed back and forth between the neighbors forever.

If a node is requesting and its neighbor is resting, it will hold onto the common resource, waiting for the other resource. If neighbor nodes are both requesting, the one with the larger value of *Suggestion* has priority, and the other will yield to it. In case of equal values, the leftmost of the two neighbors has priority. Deadlock would occur if all processes in the ring had equal values of *Suggestion*, but this is impossible; in fact, no sequence of consecutive processes with the same suggestion can be longer than $4k$. In case of such a sequence, the leftmost of those processes will initiate a query, the next process will yield its flags, the third process will then be able to initiate a query, and so forth, until approximately half the processes are querying.

5.0.3 Lost Queries.

A fault could delete an outstanding query, in which case its home process p could wait forever for it to return. In order to prevent this, we permit a process to send a *probe wave* searching for its query. If the probe does not find the missing query, a *report wave* returns to p with that information.

If $p.\mathit{num\_L\_null} = 6k + 2$, then p concludes that there is no query to the left of p with home process p; similarly, if $p.\mathit{num\_R\_null} = 6k + 2$, p concludes that there is none to the right. In either case, p initiates a probe wave (either left or right, respectively) which moves out to distance $6k+1$ from p, the farthest possible distance of the query. If the probe wave actually meets the query, it deletes itself, and no report is sent back to p. Otherwise, when the probe wave reaches the process $6k + 2$ from p in the appropriate direction, a report wave returns to p. Upon receiving this report, $p.\mathit{status} \leftarrow \mathit{resting}$. If p still needs to initiate a query, it will do so at a later step.

6 Proof Sketch

We now give a summary description of how LE(k) converges to a legitimate configuration.

We first note that the number of false votes cannot exceed $3k$ (see paragraph *Voting* of the previous section), and thus all but at most $3k + 1$ processes have received no votes from any process. Faults can also create false rumors, but the number of processes which are rumored to be the leader is at most $3k + 1$, as well. However, a false rumor could be held by many processes, and the number of processes with false rumors is $O(k^2)$.

A fault by a process p could cause as many as nine rogue queries; one on each query variable of p, one for each probe or report wave, and up to two more by faults in its status and flags. Thus, the number of rogue queries cannot exceed $9k$.

6.1 Deadlock

We need to prove that no track of LE(k) can be deadlocked. There is no deadlock possible for the rumor track. The rule that denials, probes, and reports can overwrite others with lower priority ensures that none of the two denial tracks, two probe tracks, or two report tracks can be deadlocked.

The third query track cannot become deadlocked, since the number of outstanding queries never exceeds the number of legitimate queries plus the number of rogue queries, which is never greater than $\frac{n}{2} + 9k < n$. (This is why we should assume that $n \geq 18k + 1$.) Because of the flags, no more than half of the processes can have legitimately initiated outstanding queries, and there are no more than $9k$ rogue queries. Thus, there is always some empty place in the third query track. By a similar argument, using the fact that every variable in the third query track will eventually be $\perp$, we can show that the second query track cannot become deadlocked. Similarly, we can show that the first query track cannot become deadlocked.

6.2 Partial Correctness

Since faults can never cause more than $3k$ processes to change their votes (see paragraph *Voting* of the previous section), there are always at least $3k + 1$ votes for the previous leader ℓ. Thus, it is not possible for LE(k) to reach a legitimate configuration where a different leader is elected.

On the other hand, suppose LE(k) has reached a final configuration, *i.e.,* no process is enabled, but the configuration is not legitimate. All query, denial, probe, and report variables must be $\perp$. If there are some unsatisfied processes, at least one of them will become enabled to initiate a query, contradiction, while if all are satisfied, there must be two adjacent processes which have rumors for different candidates. At least one of those rumors will be dominated by the other, and thus enabled to change, contradiction. Thus, any final configuration of LE(k) must be legitimate.

6.2.1 Congestion.

Although LE(k) converges, we still need to prove that it reaches a final configuration within $O(k^2)$ rounds. We will use a potential argument. For each rogue query x, define $\phi(x)$ to be the distance, along its query path, from the host of x to its home. Let $\Phi = \sum \phi(x)$, where the sum is taken over all existing and future rogue queries. We can then prove that Φ decreases at an average rate of $\Omega(1)$ per round, if there are any rogue queries left in the configuration. We can also prove that, if there are no rogue queries, LE(k) reaches a legitimate configuration in $O(k)$ rounds. Using these two facts, we can prove the round complexity.

6.2.2 How it works.

Consider the first configuration after the last fault. The largest false rumor in this configuration will be eliminated within $O(k)$ rounds, as follows. At least one process will hold the largest false rumor and be enabled to initialize a query. After $O(k)$ rounds, this query will return with the information that its candidate is not the leader, and a denial wave will eliminate all rumors that it is the leader. This decreases the number of processes rumored to be leader by one, and after $O(k)$ such sequences, no process, other than ℓ, will be rumored to be the leader.

At this point, if there is still a process that is not satisfied, it will initiate a query whose candidate process is ℓ. When this query returns, it will initiate the rumor that ℓ is the leader, and within $O(k)$ additional rounds, all processes in the interval of relevance of ℓ will have the rumor that ℓ is the leader. If there are any remaining processes whose votes are not for ℓ, they will send out queries and correct their votes within $O(k)$ additional rounds. Thus, the time complexity of LE(k) is $O(k^2)$ rounds.

7 Concluding remarks

The fact that number of faults is permitted to be at most k helps us to circumvent two classical impossibility results in self-stabilization. First, terminating and self-stabilizing leader recovery (or election) is impossible with fewer than $\log(n)$ bits (we use $O(\log k)$ bits per process). Secondly, deterministic and self-stabilizing leader recovery is impossible in uniform rings (our protocol is deterministic yet the ring we consider is uniform). Our result proves that the set of problems that can be solved by k-stabilizing protocols is larger than the set of those that can be solved by self-stabilizing protocols. In addition, we use less memory and less time. Our research suggests the need for further investigation of the relationship between arbitrary recovery and recovering a previous solution.

References

[1] Anish Arora and Hongwei Zhang. Lsrp: Local stabilization in shortest path routing. In *DSN*, pages 139–148. IEEE Computer Society, 2003. 1, 1

[2] Joffroy Beauquier, Sylvie Delaët, and Sammy Haddad. A 1-strong self-stabilizing transformer. In Ajoy Kumar Datta and Maria Gradinariu, editors, *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2006. 1

[3] Joffroy Beauquier, Sylvie Delaët, and Sammy Haddad. Necessary and sufficient conditions for 1-adaptivity. In *Proceedings of IEEE International Conference on Parallel and Distributed Systems (IPDPS 2006)*, April 2006. 1, 1

[4] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. k-stabilization of reactive tasks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 1998)*, page 318, 1998. 1, 1

[5] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. Optimal reactive k-stabilization: The case of mutual exclusion. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 1999. 1, 1

[6] Joffroy Beauquier and Thomas Hérault. Fault-local stabilization: The shortest path tree. In *SRDS*, pages 62–69. IEEE Computer Society, 2002. 1, 1

[7] Janna Burman, Ted Herman, Shay Kutten, and Boaz Patt-Shamir. Asynchronous and fully self-stabilizing time-adaptive majority consensus. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *OPODIS*, volume 3974 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2005. 1, 1

[8] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *SSS*, pages 189–203. Univ. Paris 6, 2007. 1, 1

[9] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. 1

[10] Shlomi. Dolev. *Self-stabilization*. MIT Press, March 2000. 1

[11] Christophe Genolini. Optimal k-stabilization: the case of synchronous mutual exclusion. In *Proceedings of Parallel and Distributed Computing Systems (PDCS'2000)*, pages 371–376, November 2000. 1, 1

[12] Christophe Genolini and Sébastien Tixeuil. A lower bound on k-stabilization in asynchronous systems. In *Proceedings of IEEE 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, pages 211–221, Osaka, Japan, October 2002. 1, 1

[13] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996. 1

[14] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007. 1, 1

[15] Sukumar Ghosh and Xin He. Scalable self-stabilization. *J. Parallel Distrib. Comput.*, 62(5):945–960, 2002. 1, 1

[16] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theor. Comput. Sci.*, 220(1):93–111, 1999. 1, 1

[17] Shay Kutten and David Peleg. Fault-local distributed mending. *J. Algorithms*, 30(1):144–165, 1999. 1, 1

[18] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009. 1