Fast Leader (Full) Recovery despite Dynamic Faults[⋆]

Ajoy K. Datta[1], Stéphane Devismes[2], Lawrence L. Larmore[1], and Sébastien Tixeuil[3]

[1] Department of Computer Science, UNLV, USA, Firstname.Lastname@unlv.edu
[2] VERIMAG, Université de Grenoble, France, Stephane.Devismes@imag.fr
[3] LIP6, UPMC Sorbonne Universités & Inria, France, Sebastien.Tixeuil@lip6.fr

Abstract. We give a leader recovery protocol that recovers a legitimate configuration where a single leader exists, after at most k arbitrary memory corruptions hit the system. That is, if a leader is elected before state corruptions, the *same* leader is elected after recovery. Our protocol works in any 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 memory corruption may still occur while the network has started recovering the leader.

1 Introduction

Self-stabilization [1] 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.

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 [2–5]. Defining the number of faults hitting a network using some kind of Hamming distance,[4] variants of the self-stabilization paradigm have been given, *e.g.*, k-stabilization [2] 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 have been used for (1) circumventing impossibility results in self-stabilization (*e.g.* deterministic leader election or recovery in anonymous networks) and (2) obtaining recovery times that only depend on the number of faults k (as opposed to n or D, the network size or diameter). The algorithm given here recovers in $O(k^2)$ rounds, and satisfies both conditions.

The concept of only-k-dependent recovery time has been refined under the name of *time adaptivity* (or fault locality) [3–5], 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 to be not greater than k (that is, they are k-stabilizing), but the latest work [3] 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

[⋆] See www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2012-18 for the full version.

[4] The minimal number of processes whose state must be changed to recover a correct configuration.

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 more slowly (that is, the recover time depends 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 [6]) remains large.

The problem of correcting global states quickly using self-stabilizing algorithms was investigated for the purpose of *fault containment* [6–9] (that is, preventing local memory corruptions from propagating to the whole network). The state of the art in this matter nevertheless requires that only a single process is corrupted [6], faulty processes are surrounded by many correct ones so that few faults can be caught quickly [8], the network is fully synchronous [7], or that the recovery guarantee is only probabilistic [9]. 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. We give a leader recovery protocol, LE(k), that recovers a legitimate configuration where a single leader exists, after at most k arbitrary memory corruptions hit the system. That is, if a leader 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", our protocol recovers the full correct state quickly ($O(k^2)$ rounds). With respect to fault-containment, LE(k) can handle up to k faults, faults can be arbitrarily spread, the network is fully asynchronous, and the scheduling is unfair, and finally the recovery property is deterministic.

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

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*, and the other its *predecessor*. The orientation is consistent: the successor of the predecessor of any 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.

k-Stabilization. Let $\mathcal{A}$ be a distributed algorithm. An ordered pair (γ, γ') is a *step* of $\mathcal{A}$ if there exist a non-empty subset S of processes enabled in γ such that γ' is the result of the atomic execution one enabled action per process of S on γ. 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 $\mathcal{A}$. A *k-fault computation* of $\mathcal{A}$ is 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 $\mathcal{A}$, and (3) the sequence is either infinite, or ends at a final configuration, where no process is enabled. $\mathcal{A}$ is *silent* if all its k-fault computations end at a final configuration.

k-fault computations are driven by a *daemon* that chooses when the faults occur and which processes execute an action when there is a step. We assume the *unfair distributed daemon*, which is otherwise unconstrained. In particular, it can choose to never select an enabled process in any step, unless it is the only enabled process.

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 Algorithm 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 at most k faults occur, the computation will end, and the last configuration will be the same as the first.

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. 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. A rumored leader is 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* has a *home process* and a *candidate process*. The *home process* is the one that initiated the query, and the candidate of the query is the one of its home process.

A query 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*. 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. The traversal of a query consists of (1) moving along the first query track toward the leftmost process in the interval of relevance of its candidate, then (2) crossing to the second query track and traversing (rightward) the whole interval of relevance of the candidate to count the votes for it, and finally (3) crossing to the third query track and moving leftward along that track until returning to its home process to report the total number of votes for the candidate.

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.

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 to be for true leader, ℓ. 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. 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.

We use two additional variables to count the number of consecutive processes to the left (resp. right) of a process, including the process itself, which have no query, probe, or report token. The value of these variables are only eventually correct, this is why we cannot directly used them to decide that a query is missing. Rather, we use them to stop generating probe waves: while the count is less or equal to $6k+1$ in some direction, the process p does not generate a probe in that direction, because there could exist a token up to $6k + 1$ hops away from p in that direction, and its home could be p.

Deadlock Prevention. As with denials, the rumors, probes, and reports can overwrite others with lower priority. This ensures that these waves cannot be deadlocked.

However, the query tracks should be carefully addressed. 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, (think of the chopstick between two philosophers in the classic Dining Philosophers problem), and a process must have both adjacent resources to initiate a query, and must hold onto both while it is querying. A resource can be held by only one of its two neighboring processes. It is implemented using two flags, one at each node. To prevent contention, we allow a process to pass a token query flag to its neighbor, but not to seize the token.

The number of outstanding queries never exceeds the number of legitimate queries plus the number of rogue queries. 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. So, the number of outstanding queries never exceeds $\frac{n}{2} + 9k < n$ Thus, assuming $n \geq 18k+1$, the third query track cannot be deadlocked because there is always some empty place in that track. Similarly, the other query tracks also cannot be deadlocked.

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11) (1974) 643–644
2. Beauquier, J., Genolini, C., Kutten, S.: k-stabilization of reactive tasks. In: PODC. (1998) 318
3. Burman, J., Herman, T., Kutten, S., Patt-Shamir, B.: Asynchronous and fully self-stabilizing time-adaptive majority consensus. In: OPODIS. Volume 3974. (2005) 146–160
4. Kutten, S., Patt-Shamir, B.: Stabilizing time-adaptive protocols. TCS **220**(1) (1999) 93–111
5. Kutten, S., Peleg, D.: Fault-local distributed mending. J. Algorithms **30**(1) (1999) 144–165
6. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing **20**(1) (2007) 53–73
7. Ghosh, S., He, X.: Scalable self-stabilization. JPDC **62**(5) (2002) 945–960
8. Beauquier, J., Delaët, S., Haddad, S.: Necessary and sufficient conditions for 1-adaptivity. In: IPDPS. (April 2006)
9. Dasgupta, A., Ghosh, S., Xiao, X.: Probabilistic fault-containment. In: SSS. (2007) 189–203