An Optimal Self-Stabilizing Firing Squad*

Danny Dolev[1],** Ezra N. Hoch[1], Yoram Moses[2]

[1] The Hebrew University of Jerusalem
Jerusalem, Israel
[2] Technion—Israel Institute of Technology
Haifa, Israel

Abstract. Consider a fully connected network where up to t processes may crash, and all processes start in an arbitrary memory state. The self-stabilizing firing squad problem consists of eventually guaranteeing simultaneous response to an external input. This is modeled by requiring that the non-crashed processes "fire" simultaneously if some correct process received an external "GO" input, and that they only fire as a response to some process receiving such an input. This paper presents FIRE-SQUAD, the first self-stabilizing firing squad algorithm.

The FIRE-SQUAD algorithm is optimal in two respects: (a) Once the algorithm is in a safe state, it fires in response to a GO input as fast as any other algorithm does, and (b) Starting from an arbitrary state, it converges to a safe state as fast as any other algorithm does.

1 Introduction

The firing squad problem was first introduced in [2,3]. Informally, it is assumed that at any given round a process may receive an external "GO" input, which is considered a request for the correct processes to simultaneously "fire." Roughly, a good solution is a protocol satisfying three properties: (a) if some process fires in round r then all the non-crashed processes fire simultaneously in round r; (b) if a correct process receives a GO input in round r' then it will fire at some later round $r > r'$; and (c) a process fires in round r only if some process received a GO input in some round $r' < r$. (The formal definition disallows a solution in which a single input induces a constant firing.)

Requiring the processes to fire simultaneously captures an important aspect of distributed systems: There are cases in which it is important

* This is the authors self-copy of the paper. The original publication is available at www.springerlink.com

** This research was supported in part by Israeli Science Foundation (ISF) Grant number 0397373.

that activities begin in the same round, *e.g.,* when one distributed algorithm ends and another one begins, and the two may interfere with each other if executed concurrently. Similarly, many synchronous algorithms are designed assuming that all sites start participating in the same round of communication. Finally, simultaneity may be motivated by the fact that a distributed system interacts with the outside world, and these interactions should often be simultaneously consistent. A non-simultaneous announcement to financial (stock) markets may enable unfair arbitrage trading, for example.

Coordinating simultaneous actions is not subsumed by the consensus task. Indeed, even when no transient failures are considered possible (so there is a global clock and no self-stabilization is required), solving the firing squad problem or simultaneously deciding in a consensus task can be considerably harder than plain consensus [4,8]. This implies, in particular, that clock synchronization [7,11,6,12,18] does not suffice for solving the firing squad problem in a self-stabilizing manner; as it can be seen as providing round-numbers to a self-stabilizing environment, which still leaves the firing squad problem as a non-trivial problem.

The firing squad problem is a primary example of a problem requiring simultaneously coordinated actions by the non-faulty processes. Simultaneous coordination has been shown to be closely related to the notion of common knowledge [10,9], and this connection has been used to characterize the earliest time required to reach simultaneous consensus, firing squad, and related problems in a variety of failure models [8,15,1,17,13,16]. One of the consequences of this literature is the fact that the time at which a simultaneous action that is based on initial values or external inputs can be performed depends in a crucial way on the pattern in which failures occur.

A general form of simultaneous agreement called *continuous consensus* was defined in [13]. In this problem, each of the processes maintains a list of events of interest that have taken place in the run, and it is guaranteed that the lists at all non-faulty processes are identical at all times. They present an optimal (non-stabilizing) implementation of such a service, which is a protocol called ConCon. If we define as the events to be monitored by ConCon to be of the form (GO, p, k), corresponding to a GO message arriving at process p at the end of round k, then a firing squad protocol can be obtained from ConCon simply by having the non-faulty processes fire exactly when a (GO, p, k) event first appears in their identical copies of the "common" list. We shall refer to this solution to the firing squad problem based on ConCon by CCfs.

Traditionally, the firing squad problem assumes that processes do not recover, *i.e.,* failed processes stay failed forever. Moreover, even though it is easy to extend the firing squad problem so that it can be repeatedly executed (*i.e.,* allow for multiple firings over time, given that multiple GO inputs are received), it assumes that nothing in the system goes amiss—except possibly for the crash failures being accounted for. Adding support for handling transient faults increases the robustness of a firing squad algorithm in this aspect. Indeed, a self stabilizing solution will, in particular, be able to cope with process recovery: Following process recoveries, the system will eventually converge to a valid state and continue operating correctly.

Transient faults alter a process's memory state in an arbitrary way. A self-stabilizing algorithm [5] is assumed to start in an arbitrary state and be guaranteed to eventually reach a state from which it operates according to its intended specification. Starting the operation at an arbitrary state enables the adversary to "plant" false information, such as the receipt of GO messages in the past, which can cause the algorithm to unjustifiably fire, either immediately, or within a few rounds. One of the challenges in designing an efficient self-stabilizing firing squad algorithm is in bounding the damage that can be caused by such false information in the initial state.

Perhaps the first candidate solution would be to initiate an instance of CCFS in every round, with $t + 1$ instances executing concurrently at any given time, where t is an upper bound on the number of possible crashed processes. Firing would then take place if it is dictated by any of the instances. Since the component instances of such a solution are not themselves stabilizing, all we can show is that such a solution is guaranteed to stabilize after $t+1$ rounds, regardless of the failure pattern. We shall present a solution that does not consist of such a concurrent composition. Moreover, it performs subtle consistency checks to restrict the impact of false information that appears in the initial state. As a result, in some cases we obtain stabilization in as little as two rounds.

The above discussion points out the stabilization time as an important aspect of a self-stabilizing firing squad algorithm. Another central performance parameter is its swiftness: Once the algorithm has stabilized, how fast does it fire given that some process receives a GO input? In addition to solving the self-stabilizing firing squad problem, the algorithm presented in this paper is also optimal in terms of both its stabilization time, and its swiftness.

The main contributions of this paper are:

– A self-stabilizing variant of the firing squad problem is defined, and an algorithm solving it in the case of crash failures is given.
– The proposed algorithm, called FIRE-SQUAD, is shown to be optimal both in terms of the time it requires to stabilize and in terms of the time it takes, after stabilization, to fire in response to a GO input.
– Finally, the optimality is demonstrated in a fairly strong sense: For every possible failure pattern, both stabilization time and swiftness are the fastest possible, in any correct algorithm. In extreme cases this enables stabilization in two rounds and firing in one round.

The rest of the paper is organized as follows. Section 2 describes the model and defines the problem at hand. Section 3 provides lower bounds for the optimality properties. Section 4 describes the proposed solution, FIRE-SQUAD, and proves its correctness and optimality. Finally, Section 5 concludes with a discussion.

2 Model and Problem Definition

The system consists of a set $\mathcal{P} = \{1, \ldots, n\}$ of processes. Communication is done via message passing, and the network is synchronous and fully connected. The system starts out at time[3] $k = 0$, and a communication round r starts at time $k = r - 1$ and ends at time $k = r$. At time k each process computes its state according to its state at time $k-1$, the internal messages it received by time k (sent by other processes at time $k - 1$) and external inputs (if any) that it received at time k. In addition, at any time $k \geq 0$ a process can produce an external output (such as "firing").

Let $\mathcal{I}_p^k \in \{0, 1\}$ represent the external input of process p at time k. We say that p received an external GO input at time k if $\mathcal{I}_p^k = 1$; Otherwise, (if $\mathcal{I}_p^k = 0$), we say that p *did not* receive a GO input. Let $\mathcal{I}_p = \{\mathcal{I}_p^k\}_{k=0}^{\infty}$, let $\mathcal{I}^k = \{\mathcal{I}_p^k\}_{p=1}^{n}$ and let $\mathcal{I} = \{\mathcal{I}_p\}_{p=1}^{n}$. $\mathcal{I}$ is "the input pattern", and $\mathcal{I}^k$ is the (joint) input at time k. In a similar manner define $\mathcal{O}_p^k \in \{0, 1\}, \mathcal{O}_p, \mathcal{O}^k$ and $\mathcal{O}$ as the output pattern. If $\mathcal{O}_p^k = 1$ we say that p fires at time k, and if $\mathcal{O}_p^k = 0$ we say p does not fire at time k. It will be convenient to say that a fire action occurs at time k if $\mathcal{O}_p^k = 1$ for some process p, and similarly that a GO input is received at time k if $\mathcal{I}_p^k = 1$ for some p.

Denote by t an *a priori* bound on the number of faulty processes in the system. For ease of exposition, we assume that $t < n - 1$, so that there are at least two processes that need to coordinate their actions. We

[3] All references to "time" in this paper refer to non-negative integer times.

assume the *crash* failure model, in which a faulty process p does not send any messages after its failing round; it behaves correctly before its failing round, and sends an arbitrary subset of its intended messages during its failing round.

A failure pattern describes for each time k which processes have failed by time k, and for each process that fails in round k (*i.e.,* did not fail by time $k-1$), which of its outgoing communication channels are blocked (and hence do not deliver its messages) in round k. Notice that a process may fail in round k even if all of its messages are delivered. We denote a failure pattern by $\mathcal{F}$, and by $\mathcal{F}^k$ the set of processes that fail in $\mathcal{F}$ by time k. Observe that $\mathcal{F}^k \subseteq \mathcal{F}^{k+1}$; in the crash failure model failed processes do not recover. Similarly, we use $\mathsf{G}^k = \mathcal{P} \setminus \mathcal{F}^k$ to denote the set of processes that are non-faulty at time k. Finally, G will denote the set of processes that remain non-faulty throughout $\mathcal{F}$, *i.e.,* $\mathsf{G} = \bigcap_{k=0}^{\infty} \mathsf{G}^k$. Notice that the set G is always defined in terms of a failure pattern $\mathcal{F}$, which is typically clear from the context.

In addition to crashes, there are also transient faults. Formally, we denote by $\mathcal{S}_p^k$ the state of a process p at time k. We denote by $\mathcal{S}^k = (\mathcal{S}_1^k, \ldots, \mathcal{S}_p^k, \ldots, \mathcal{S}_n^k)$ the state of the entire system at time k. Transient faults are captured by the assumption that the system may start from any (arbitrary) state, and there is some round r such that for all rounds $r' \geq r$ the intended algorithm operates as written. In other words, for any possible state S, if $\mathcal{S}^0 = S$ then eventually (starting from some round r) the algorithm operates correctly.

For the following analysis, each algorithm $\mathcal{A}$ is assumed to have an initial state $\mathcal{S}_{init}^{\mathcal{A}}$. For self-stabilizing algorithms, we fix an arbitrary state as $\mathcal{S}_{init}^{\mathcal{A}}$ (as the algorithm should converge starting from any initial state). The *a priori* bound of t on the number of failures is assumed to be hard-wired into the algorithm, and is not affected by transient faults. Such an algorithm is assumed to be executed only in the context of failure patterns in which at most t processes crash. For such failure patterns $\mathcal{F}$, the algorithm $\mathcal{A}$ produces an output pattern $\mathcal{O}$ starting from state $\mathcal{S}$ given an input $\mathcal{I}$; we denote this output pattern by $\mathcal{O} = \mathcal{A}(\mathcal{S}, \mathcal{I}, \mathcal{F})$.

Informally, the Firing Squad problem requires that: (1) all processes fire together (*"simultaneity"*); (2) if a GO input is received then a fire action occurs (*"liveness"*); and (3) the number of fire actions is not larger than the number of received GO inputs (*"safety"*). Formally,

Definition 1. *Let $\mathcal{O} = \mathcal{A}(\mathcal{S}, \mathcal{I}, \mathcal{F})$ and let G denote the set of processes that remain non-faulty throughout $\mathcal{F}$. We say that $\mathcal{O}$ satisfies the*

$\texttt{FS}(k)$ *properties (capturing correct firing-squad behavior from time* k *on) w.r.t.* $\mathcal{I}$, $\mathcal{F}$, *and* $\mathcal{O}$, *if the following conditions hold for all* $k' \geq k$*:*

1. *(simultaneity) If* $\mathcal{O}_p^{k'} = 1$ *for some* $p \in \mathcal{P}$ *then* $\mathcal{O}_q^{k'} = 1$ *for all* $q \in \mathsf{G}$*;*
2. *(liveness) If* $\mathcal{I}_p^{k'} = 1$ *for some* $p \in \mathsf{G}$*, then there is* $k'' > k'$ *s.t.* $\mathcal{O}_p^{k''} = 1$*;*
3. *(safety) The number of times* k'' *satisfying* $k \leq k'' \leq k'$ *at which a fire action occurs at* k'' *is not larger than the number of times* h *in the range* $0 \leq h < k'$ *at which* GO *inputs are received.*

We can use the $\texttt{FS}(k)$ properties to define when an algorithm solves the firing squad problem in a self stabilizing manner. We first use it to define the stabilization time of an algorithm as follows:

Definition 2 (Stabilization time). *The* stabilization time *of* $\mathcal{A}$ *on* $\mathcal{S}$, $\mathcal{I}$ *and* $\mathcal{F}$, *denoted by* $\texttt{stab}(\mathcal{A}, \mathcal{S}, \mathcal{I}, \mathcal{F})$, *is the minimal* $k \geq 0$ *such that* $\texttt{FS}(k)$ *holds with respect to* $\mathcal{I}$, $\mathcal{F}$, *and* $\mathcal{O} = \mathcal{A}(\mathcal{S}, \mathcal{I}, \mathcal{F})$. *(If* $\texttt{FS}(k)$ *holds for no finite* k*, then* $\texttt{stab}(\mathcal{A}, \mathcal{S}, \mathcal{I}, \mathcal{F}) = \infty$*.)*

Notice that the *"safety"* property in $\texttt{FS}(k)$ relates outputs starting from time k to inputs starting from time 0. Here's why: Since we consider time 0 to be the point at which transient errors end, if the system starts in a state in which "it appears as if" GO inputs were received before time 0, the good processes may fire after time 0 without a GO message actually having been received. Once all firings induced by such "phantom" GO inputs have occurred, we can legitimately require firing events to happen only in response to genuine GO message receipts. We thus think of the stabilization time, at which in particular the safety property of $\texttt{FS}(k)$ holds, as one after which no firing will occur in response to phantom GO messages. Rather, every firing will be justifiable as a response to some GO message received at or after time 0.

Definition 3 (SSFS Algorithm). *An algorithm* $\mathcal{A}$ *solves the Self stabilizing Firing Squad problem (*$\mathcal{A}$ *is an SSFS algorithm, for short) if there exists a* $k < \infty$ *such that* $\texttt{stab}(\mathcal{A}, \mathcal{S}, \mathcal{I}, \mathcal{F}) \leq k$ *for every system state* $\mathcal{S}$*, input pattern* $\mathcal{I}$ *and failure pattern* $\mathcal{F}$*.*

Observe that in a setting with no transient faults, an algorithm $\mathcal{A}$ solves the (non-self-stabilizing) Firing Squad problem if it satisfies $\texttt{FS}(0)$ with respect to $\mathcal{I}$, $\mathcal{F}$, and $\mathcal{O}$, for every $\mathcal{I}$, $\mathcal{F}$ and $\mathcal{O} = \mathcal{A}(\mathcal{S}_{init}^{\mathcal{A}}, \mathcal{I}, \mathcal{F})$.

Notice that Definition 3 implies that any SSFS algorithm $\mathcal{A}$ has at least one memory state from which the firing squad properties are guaranteed to hold. Denote one of these memory states by $\mathcal{S}_{stab}^{\mathcal{A}}$, or simply $\mathcal{S}_{stab}$ when $\mathcal{A}$ is clear from the context.

2.1 Optimality Measures

In this work we are interested in finding an optimal SSFS algorithm. We start by defining stabilization time optimality, which measures how quickly algorithm $\mathcal{A}$ stabilizes.

Definition 4. *An SSFS algorithm $\mathcal{A}$ is said to* optimally stabilize *if the following holds for every SSFS algorithm $\mathcal{B}$ and every failure pattern $\mathcal{F}$:*

$$\max_{\mathcal{S},\mathcal{I}}\{\texttt{stab}(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F})\} \quad \leq \quad \max_{\mathcal{S},\mathcal{I}}\{\texttt{stab}(\mathcal{B},\mathcal{S},\mathcal{I},\mathcal{F})\}\ .$$

Definition 4 defines optimality of an algorithm $\mathcal{A}$ with respect to its stabilization time, *i.e.,* how quickly $\mathcal{A}$ starts to operate according to all of the FS requirements. The intuition behind defining optimality in terms of worst-case $\mathcal{S}$ and $\mathcal{I}$ is to avoid algorithms that are "specific" to an initial memory state or input pattern. Thus, by requiring optimality in the worst-case we ensure that the algorithm cannot be hand-tailored to a specific setting, but rather needs to solve the SSFS problem in a "generic" manner.

We now turn to the issue of comparing the responsiveness of distinct firing squad algorithms. Specifically, we are concerned with how quickly an algorithm fires after a GO message is received (once the algorithm has stabilized). For simplicity, we consider receipts of GO by non-faulty processes, since the problem specification forces a firing following such a receipt. Another subtle issue is that if GO messages are received in different rounds between which there is no firing, then it may be difficult to figure out which GO message the next firing is responding to. Again for simplicity, we will be interested in what will be called *sequential* input patterns, in which a GO is not received before all previous GO's have been followed by firings. More formally, we define:

Definition 5 (Sequential inputs). *Let $\mathcal{A}$ be an SSFS algorithm. We say that the input $\mathcal{I}$ is sequential with respect to ($\mathcal{A}$, $\mathcal{S}$, $\mathcal{F}$) if (i) no* GO *inputs are received according to $\mathcal{I}$ at times $k < \texttt{stab}(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F})$, (ii)* GO *inputs are received in $\mathcal{I}$ only by processes from* G*, and (iii) if $k_1 < k_2$ and* GO *inputs are received at both k_1 and k_2, then there is an intermediate time $k_1 < k' \leq k_2$ at which a fire action occurs.*

The following definition formally captures the number of firing events that occur between the stabilization time and a given time k.

Definition 6. *Let $\mathcal{A}$ be an SSFS algorithm and let $\mathcal{O} = \mathcal{A}(\mathcal{S},\mathcal{I},\mathcal{F})$. We define $\#[(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F}),k]$ to be the number of rounds k' in the range*

$\mathtt{stab}(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F}) \leq k' \leq k$ *such that* $\mathcal{O}_p^{k'} = 1$ *holds for some process* p *(*i.e., *a firing occurs at time* k'*).*

By definition, if $k < \mathtt{stab}(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F})$ then $\#[(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F}),k] = 0$. With the last two definitions, we are now able to formally compare the responsiveness of different SSFS algorithms:

Definition 7 (Swiftness). *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be SSFS algorithms. We say that* $\mathcal{A}$ *is* at least as swift as $\mathcal{B}$ *if* $\mathcal{A}$ *fires at least as quickly as* $\mathcal{B}$ *on all sequential inputs. Formally, we require that for every failure pattern* $\mathcal{F}$*, input* $\mathcal{I}$*, and states* $\mathcal{S}_\mathcal{A}$ *of* $\mathcal{A}$ *and* $\mathcal{S}_\mathcal{B}$ *of* $\mathcal{B}$*, the following holds. If* $\mathcal{I}$ *is sequential both with respect to (*$\mathcal{A}$*,* $\mathcal{S}_\mathcal{A}$*,* $\mathcal{F}$*) and with respect to (*$\mathcal{B}$*,* $\mathcal{S}_\mathcal{B}$*,* $\mathcal{F}$*), then* $\#[(\mathcal{A},\mathcal{S}_\mathcal{A},\mathcal{I},\mathcal{F}),k] \geq \#[(\mathcal{B},\mathcal{S}_\mathcal{B},\mathcal{I},\mathcal{F}),k]$ *holds for every time* k*. An SSFS algorithm* $\mathcal{A}$ *is optimally swift if it is at least as swift as* $\mathcal{B}$ *for every SSFS algorithm* $\mathcal{B}$*.*

We are now in a position to state the main result of the paper: The Fire-Squad algorithm of Figure 1 is an SSFS algorithm, is optimally stabilizing and is optimally swift (Theorem 3).

3 Lower Bounds

In this section we provide lower bounds for the stabilization time and for the swiftness of any SSFS algorithm $\mathcal{A}$. The lower bounds build upon previous results in the field of simultaneous agreement.

Recall that if $\mathcal{A}$ is a non-self-stabilizing Firing Squad algorithm, then $\mathtt{stab}(\mathcal{A},\mathcal{S}^\mathcal{A}_{init},\mathcal{I},\mathcal{F}) = 0$ for all $\mathcal{I}$ and $\mathcal{F}$. Therefore, in the non-self-stabilizing case, it only makes sense to compare algorithms in terms of their "swiftness." In a non-self-stabilizing setting, the firing squad protocol CCfs (based on ConCon [13]) is optimally swift. We will use it as a benchmark and yardstick for expressing and analyzing the performance of self-stabilizing firing squad protocols. To compare the performance of different algorithms, we make use of the following definitions.

Definition 8. *We denote by* $\delta(\mathcal{F},k)$ *the number of processes known at time* k *to be faulty by the processes in* G^k *in a run of* CCfs *with failure pattern* $\mathcal{F}$*.*

Intuitively, $\delta(F,k)$ stands for the number of failures that are *discovered* by time k in a run with pattern $\mathcal{F}$. We remark that $\delta(\mathcal{F},k)$ is well-defined, because the same number of faulty processes are discovered (at the same times) in all runs of CCfs that have failure pattern $\mathcal{F}$. Moreover, since

CCFS detects failures as a full-information protocol does, no algorithm $\mathcal{A}$ can discover more failed processes than CCFS does (see [8]). Thus, $\delta(\mathcal{F}, k)$ is an upper bound on the number of failed process discovered by time k by any algorithm $\mathcal{A}$.

CCFS makes essential use of a notion of *horizon*, which is roughly the time by which past events are guaranteed to become common knowledge. This motivates the following definitions.

Definition 9 (Horizons). *Given a failure pattern $\mathcal{F}$, the* horizon distance *at time k, denoted by $\mathtt{disH}(\mathcal{F}, k)$, is $t + 1 - \delta(\mathcal{F}, k)$. The* absolute horizon *at time k, denoted $\mathtt{absH}(\mathcal{F}, k)$, is $k + \mathtt{disH}(\mathcal{F}, k)$.*

While the absolute horizon is an upper bound on when events become common knowledge, the publication time is a lower bound on this time. It is defined as follows:

Definition 10 (Publication Time). *Given a failure pattern $\mathcal{F}$, the* publication time *for (time) k, denoted by $\pi(\mathcal{F}, k)$, is $\min_{k' \geq k}\{\mathtt{absH}(\mathcal{F}, k')\}$.*

When $\mathcal{F}$ is clear from the context, it will be omitted from $\delta(k)$, $\mathtt{disH}(k)$, $\mathtt{absH}(k)$ and $\pi(h)$.

As shown in [13], for a given failure pattern $\mathcal{F}$, a GO input received at time k is "common knowledge" not before time $\pi(\mathcal{F}, k)$. Thus, for a specific algorithm $\mathcal{A}$, the publication time for 0 bounds (from below) the time k at which the first firing action can occur in $\mathcal{O} = \mathcal{A}(\mathcal{S}_{stab}, \mathcal{I}, \mathcal{F})$.

The publication time $\pi(\mathcal{F}, k)$ is a generalization of notions developed in [8] for Simultaneous (single-shot, non-stabilizing) Consensus. In that paper, a notion of the *waste* of $\mathcal{F}$ is defined, and information about initial values—which can be viewed in our setting as being about external inputs at time 0—becomes common knowledge at time $t + 1 - \mathit{waste}$. In our terminology, this occurs precisely at the publication time $\pi(\mathcal{F}, 0)$ for events of time 0.

The intuition behind the first lower bound is that if CCFS receives a GO input at time 0, then it fires at time $\pi(0)$ (Lemma 1). Since CCFS is optimal, an SSFS algorithm $\mathcal{A}$ cannot fire faster. Therefore, if we consider $\mathcal{A}$ starting in a memory state where $\mathcal{A}$ "thinks" it received a GO input 1 round ago, $\mathcal{A}$ will fire not before time $\pi(0) - 1$.

Lemma 1. *Let $\mathcal{F}$ be any failure pattern and let $\mathcal{I}$ be an input pattern for which $\mathcal{I}_q^k = 0$ for every process q and time $k \geq 0$, except for one process $p \in \mathsf{G}$ for which $\mathcal{I}_p^0 = 1$. The first fire action of $\mathcal{O} = \mathrm{CC}_{\mathrm{FS}}(\mathcal{S}_{init}^{\mathrm{CC}_{\mathrm{FS}}}, \mathcal{I}, \mathcal{F})$ occurs at time $\pi(\mathcal{F}, 0)$.*

Following is the first lower bound result, stating that the worst case stabilization time of every SSFS algorithm $\mathcal{A}$ is at least $\pi(0)$.

Theorem 1. $\max_{\mathcal{S},\mathcal{I}}\{\mathtt{stab}(\mathcal{A},\mathcal{S},\mathcal{I},\mathcal{F})\} \geq \pi(\mathcal{F},0)$ *holds for every SSFS algorithm* $\mathcal{A}$ *and every failure pattern* $\mathcal{F}$*.*

Our second lower bound result, informally stating that any SSFS algorithm cannot fire faster than CCFS, is captured by the following theorem. (Notice that the claim is made with respect to sequential input patterns.)

Theorem 2. *Let* $\mathcal{A}$ *be an SSFS algorithm,* $\mathcal{I}$ *a sequential input,* $\mathcal{F}$ *a failure pattern and* $\mathcal{O} = \mathcal{A}(\mathcal{S}_{stab},\mathcal{I},\mathcal{F})$*. For every* $k \geq 0$ *for which a* GO *input is received in* $\mathcal{I}^k$ *there is no fire action in* $\mathcal{O}$ *during times* k' *satisfying* $k < k' < \pi(\mathcal{F},k)$*.*

4 Solving SSFS

The algorithm FIRE-SQUAD in Figure 1 is an SSFS algorithm that is both optimally stabilizing and is optimally swift. For swiftness, the algorithm is based on the approach used in the CCFS algorithm, in which the horizon is computed by monitoring the number of failures that occur, and a firing action takes place when the receipt of a GO becomes common knowledge. The horizon computation at a process p makes use of reports that p receives from other processes regarding failures that they have observed. Following a transient fault, the state of a process may contain arbitrary (including false) information about failures. In the crash failure model, a process q will learn about (truly) crashed processes in the first round. Consequently, p will compute a correct horizon one round later, once it receives reports from all such processes. Roughly speaking, this can be used as a basis for a (nontrivial) solution that stabilizes within two rounds of the optimal time.

In order to improve on the above and obtain an optimal algorithm, FIRE-SQUAD employs a couple of subtle consistency checks. The first one involves checking the information obtained from other processes regarding failures they observed before the current round started. In the crash failure model, every failure observed by q by time $k-1$ must be directly observable by p no later than time k. So if the set of failures reported to p contains failures that p has not directly observed, then it must be time $k \leq 1$, and p will use the set of failures that it has directly observed in computing the horizon, instead of the set of reported failures. A subtle

Algorithm FIRE-SQUAD (t)

```
0: do forever:                                   /* executed on process p at time k */
                                                 /* process p is unaware of the value of k */
1:    receive all available (Requests_q, Failed_q, Views_q) messages from process q ∈ P;

      /* update variables according to messages of round k and external input */
2:    set Requests[0] := I_p^k;
3:    for 1 ≤ i ≤ t+1: set Requests[i] := max_q{Requests_q[i − 1]};
4:    set Failed' := ⋃_q Failed_q;
5:    set Failed := all processes that p did not hear from this round;
6:    for 1 ≤ i ≤ t: set Views[i − 1] := min_q{Views_q[i]} + 1;

      /* calculate horizon at time k − 1 */
7:    set Horizon := t + 1 − min{|Failed'|, |Failed|};      /* consistency check I */
8:    set Views[Horizon-1] := 1;
9:    for 0 ≤ i ≤ t: set Views[i] := max{Views[i], Horizon − i};      /* check II */

      /* should we fire? */
10:   if for some i' ≥ Views[0] it holds that Requests[i'] = 1 then
11:      for i' ≤ i'' ≤ t + 1: set Requests[i''] := 0;
12:      do "Fire";
13:   fi;

      /* send round k + 1 messages to all processes */
14:   send (Requests, Failed, Views) to all;
15: od.
```

Clean up:

Requests contains only $\{0, 1\}$ values. *Views* contains only values $\in \{0, \ldots, t + 1\}$.

Fig. 1. FIRE-SQUAD: a self-stabilizing firing squad algorithm.

proof shows that, in this case, the computed horizon works correctly if $k = 1$, which is crucial for the algorithm's stabilization optimality. The second consistency check is based on the fact that in normal operation the horizon distance is (weakly) monotone decreasing. The local state contains information about previous horizon computations, and our second consistency check forces it to satisfy weak monotonicity.

We now turn to describe the details of FIRE-SQUAD. The following discussion and lemmas are stated w.r.t. the algorithm and its components. For a variable *var*, we denote by var_p^k the value of *var* at process p after the computation step at time k.

Each process p has a vector $Requests_p[i]$, which represents p's information about a GO input received by some process i time units ago; and this request was not fulfilled yet. More precisely, if $Requests_p^k[i] = 1$, then

some process received a GO input at time $k-i$, and no firing action occurred between time $k-i+1$ and time k. The vector *Requests* contains values for the previous $t+1$ time units and the current time; a total of $t+2$ entries.

In addition, each process has a set *Failed*, which consists of the processes it has seen to be failed in the current round. That is, at time k, process p's $Failed_p^k$ set contains all processes that process p did not received messages from during round k (*i.e.,* messages sent at time $k-1$). *Failed′* is the union of all *Failed* sets (as received from other processes) of the previous round. That is, at time k, ${Failed'}_p^k$ is the union of $Failed_q^{k-1}$ as computed at time $k-1$ by every process q that p received messages from during round k.

Finally, each process keeps track of a vector *Views*. If $Views_p^k[i] = z$ it means that at time $k+i$, data from time $k-z$ is common knowledge. The vector *Views* contains $t+1$ entries, for the current round and the coming t rounds.

For ease of exposition every process p is assumed to send messages to itself. Moreover, a process executing the algorithm is unaware of the current round number. We refer to such rounds using numbers k etc. for ease of exposition in describing and analyzing the algorithm.

4.1 Correctness Proof

A central notion in the analysis of simultaneous actions under crash failures is that of a *clean round* [8]. In the non-stabilizing setting, a round r is clean according to failure pattern $\mathcal{F}$ if no process considered non-faulty by all processes at time $r-1$ is known to be faulty by one or more (non-crashed) processes at time r. In a setting that allows transient faults, we use a slightly different definition for the exact same notion. Consider a process p that fails in round k. We say that p fails *silently* in round k if it is not blocked according to $\mathcal{F}$ from sending messages in round k to any of the processes $q \in \mathsf{G}^k$. Thus, no process surviving round k can detect p's failure in this round.

Definition 11 (Clean Round). *Round r in failure pattern $\mathcal{F}$ is a* clean round *if (i) no process fails silently in round $r-1$, and (ii) all processes (if any) that fail in round r fail silently.*

This definition of a round r being clean in $\mathcal{F}$ coincides with the standard definition of clean rounds previously used in non-stabilizing systems [8]. In protocols such as FIRE-SQUAD, with the property that every process

sends the same message to all other processes in every round, all (non-crashed) processes receive the same set of messages in a clean round.

Due to lack of space we present an overview of the proof. The full proof will appear in the full version of the paper. Following is the main result of the paper:

Theorem 3. FIRE-SQUAD *solves the SSFS problem, it optimally stabilizes and is optimally swift.*

Proof overview: First, notice that once a clean round has occurred, all processes receive the same set of messages, and different processes agree on the value of *Requests* (except for *Requests*[0]). Moreover, in the following round all processes agree on the value of *Requests* perhaps except for the value of *Requests*[0] and *Requests*[1]. In a similar manner, k rounds after a clean round the values of $Requests[k+1], Requests[k+2], \ldots$ are the same at all processes.

Second, consider the value of $Views^k[0]$. By Line 6, $Views^k[0]$ equals the value of $Views^{k-1}[1]+1$. In a similar manner, if $Views^k[i]$ is updated by Line 8 then $Views^{k+i}[0] = Views^k[i]+i$. If k was the last clean round prior to round $k+i$, then $Views[0] = i+1$ holds at time $k+i$. Together with the claim from the previous paragraph, we have that once there was a clean round, if different processes agree on the value of $Views[0]$, then they all agree on the values of $Requests[Views[0]], Requests[Views[0]+1], \cdots$. Thus, if processes agree on the value of $Views[0]$ then they are guaranteed to act simultaneously, either firing together or, together, refraining from firing. Therefore, we turn our attention to analyzing the behavior of $Views[0]$ at the different processes.

Intuitively, the reason the above discussion does not show that all processes agree on the value of $Views[0]$, is the following: Even though all processes update the value of *Views* in a similar manner (Line 6) each process p updates its own $Views_p$ according to the failures that p has seen in the current round. To show that all processes have the same value of $Views[0]$ (for all rounds following a clean round) we show two things: (1) if $Views^k[0]$ is updated in round k, then $Horizon = 1$, *i.e.,* $|Failed'| = t$. This will be observed by all processes, and so they will all set $Views^k[0] = 1$; (2) if $Views^k[0]$ was not updated in round k, then let $k-i$ be the latest round in which the value of $Views^{k-i}[Horizon^{k-i}-1]$ was updated. The proof shows that there must be a clean round between round $k-i$ and round k, thus ensuring that all processes will agree on the value in $Views[0]$ by round k.

Up till now, we have given an overview of the proof that Fire-Squad solves the SSFS problem. To show that it optimally stabilizes and is optimally swift a precise analysis of the convergence of SSFS is required, along with a proof that SSFS will fire no later than any other algorithm (on sequential inputs). To illustrate the tools used in those proofs, we define the following:

Definition 12. *Let*
- $\texttt{minHG}(\mathcal{F}, k) = \min_{p \in \mathsf{G}} \mathit{Horizon}_p^k$*, and*
- $\texttt{bestH}(\mathcal{F}, k) = \min_{k' \geq k}\{k' + \texttt{minHG}(\mathcal{F}, k'+1)\}$.

We write $\texttt{bestH}(k)$ *when* $\mathcal{F}$ *is clear from the context.*

The main point behind this definition is that `bestH` is the equivalent of π with respect to Fire-Squad (recall that π is computed according to CCfs). In the non-self-stabilizing model π is shown to be a lower bound on when a GO input becomes "common knowledge". Thus, the following two lemmas conclude that Fire-Squad is optimally swift.

Lemma 2. $\texttt{bestH}(k) \leq \pi(k)$*, for every* $k \geq 0$.

Lemma 3. *Let input* $\mathcal{I}$ *be sequential with respect to* (Fire-Squad, $\mathcal{S}, \mathcal{F}$)*. If* $\mathcal{I}_p^k = 1$ *for process* p *at time* k *then* $\mathcal{O}_p^{k'} = 1$ *for* $k < k' \leq \texttt{bestH}(k)$.

Finally, we wish to point out a main difference between the proofs of the lower and upper bounds in the self-stabilizing model as opposed to the classical model (with respect to the firing squad problem): in the first round of Fire-Squad the value of *Failed′* (the set of processes that have failed in the previous round) might be contaminated. That is, a process may start in a state where it thinks that some other processes are failed, even though they are correct. Thus, a major property that is used in the classical proofs cannot be used freely in the self-stabilizing model's proofs: the monotonicity of crash failures. In the classical model, the perceived set of crashed processes can only increase, while in the self-stabilizing model it may decrease following the first round.

This explains the purpose of Line 7, which is to perform a consistency check, comparing the reported Failed_q values (from the previous round) to failures that are directly observed by p in the current round (stored in Failed_p). This comparison together with a delicate treatment in the proofs, ensures the optimality of Fire-Squad. That is, to prove that Fire-Squad is optimal up to an additive constant of 1 round is much easier than to prove that Fire-Squad is optimal. We prove the latter, stronger, property.

5 Conclusions and Open Problems

This paper presents Fire-Squad, the first self-stabilizing firing squad algorithm. Fire-Squad is optimal in two important respects: It optimally stabilizes, and is optimally swift. There are many directions in which this work can be extended. These include:

- Fire-Squad assumes the crash fault model. What can be said about the omission fault model? And what about the *Byzantine* fault model? Each such extension seems to be a nontrivial step.
- Fire-Squad works when we assume that failures are permanent. Being an ongoing and everlasting service, firing squad is expected to operate for long periods, in which processes may recover. A more reasonable assumption in this case is that there is a bound (of t) on the number of failures over every interval of m rounds, for some m. (Non-stabilizing) Continuous consensus has recently been studied in this model [14], and it would be interesting to see if the same can be done for self-stabilizing firing squad.

References

1. Rida Bazzi and Gil Neiger. The possibility and the complexity of achieving fault-tolerant coordination. In *PODC '92*, pages 203–214, New York, USA, 1992. ACM.
2. J. E. Burns and N. A. Lynch. The byzantine firing squad problem. *Advances in Computing Research: Parallel and Distributed Computing*, 4:147–161, 1987.
3. Brian A. Coan, Danny Dolev, Cynthia Dwork, and Larry J. Stockmeyer. The distributed firing squad problem. *SIAM J. Comput.*, 18(5):990–1012, 1989.
4. Danny Dolev, Ruediger Reischuk, and Raymond H. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, 1990.
5. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
6. S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.
7. Shlomi Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, 12(1):95–107, January 1997.
8. C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.
9. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.
10. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990. A preliminary version appeared in PODC '84.
11. E. N. Hoch, D. Dolev, and A. Daliot. Self-stabilizing byzantine digital clock synchronization. In *Proc. of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, pages 350–362, Nov 2006.
12. L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

13. Tal Mizrahi and Yoram Moses. Continuous consensus via common knowledge. *Distributed Computing.*, 20(5):305–321, 2008.
14. Tal Mizrahi and Yoram Moses. Continuous consensus with failures and recoveries. In *DISC'08*, pages 408–422, 2008.
15. Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.
16. Yoram Moses and Michel Raynal. Revisiting simultaneous consensus with crash failures. *J. Parallel Distrib. Comput.*, 69(4):400–409, 2009.
17. Gil Neiger and Mark R. Tuttle. Common knowledge and consistent simultaneous coordination. *Distrib. Comput.*, 6(3):181–192, 1993.
18. B. Patt-Shamir. *A Theory of Clock Synchronization.* Doctoral thesis, MIT, Oct 1994.