Forgive & Forget: Self-Stabilizing Swarms in Spite of Byzantine Robots

by

Yotam Ashkenazi1, Shlomi Dolev, Sayaka Kamei,

Fukuhito Ooshita. and Koichi Wada

Technical Report #19-04

September 12, 2019

Forgive & Forget
Self-Stabilizing Swarms in Spite of Byzantine Robots

Yotam Ashkenazi[1], Shlomi Dolev[1], Sayaka Kamei[2], Fukuhito Ooshita[3] and Koichi Wada[4]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Israel {yotamash, dolev}@post.bgu.ac.il
[2] Department of Information Engineering, Graduate School of Engineering, Hiroshima University, Japan. s-kamei@se.hiroshima-u.ac.jp
[3] Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan. f-oosita@is.naist.jp
[4] Department of Applied Informatics, Faculty of Science and Engineering, Hosei University, Japan. wada@hosei.ac.jp

Abstract. In this paper, we consider the case in which a swarm of robots collaborates in a mission, where a few of the robots behave maliciously. These malicious Byzantine robots may be temporally or constantly controlled by an adversary. The scope is synchronized full information robot operations, where a robot that does not follow the program/policy of the swarm is immediately identified and can be remembered as Byzantine.
As robots may be suspected of being Byzantine due to benign temporal malfunctions, it is imperative to forgive and forget, otherwise, a robot cannot assume collaborative actions with any other robot in the swarm. Still, remembering for a while may facilitate a policy of surrounding, isolating and freezing the movement of the misbehaving robots, by several robots, allowing the rest to perform the swarm task with no intervention.
We demonstrate the need to periodically forgive and forget to realize swarm several tasks including patrolling/cleaning in the presence of possible Byzantine robots. The policy for achieving the task consists of blocking the movement of the Byzantine robot(s) by some of the robots, while the rest patrol/clean the plane.
We present and use self-stabilizing (non-two faced) Byzantine pulse and clock synchronizations that are of independent interest.

1 Introduction

Swarms of robots acting towards a common task are already part of our lives, be it a swarm of autonomic cars, a swarm of the unmanned aerial vehicle (UAV), or a swarm of nano-robots.

When dealing with robots in practice, we better assume that some of the robots are Byzantine, faulty or malicious [11]. These robots may not follow the algorithm either because of a fault or because of a malicious adversarial takeover.

Such malicious takeover may imply the most disturbing behavior of the maliciously controlled robot. Obviously, when all participants are Byzantine the swarm can be regarded as malicious as well, not following actions for achieving the planned goal. Since faults and takeovers can be accumulated over time, the possibility for swarm participants to stop functioning as they should do grows with time. Self-stabilizing algorithms [6,7] may cope with such faults, imposing automatic recovery of individuals in the swarm, and regaining collaboration among the recovered participants.

Many research were made on how to cope with a given threshold (e.g., less than one third) of Byzantine participants [1,4,10]. However, these researches are not aimed to cope with temporal periods in which all (or almost all) participants are Byzantine. The correctness of (non-stabilizing) algorithms is based on the consistency of the initial configuration and the preservation of the consistency as long as the threshold on the number of Byzantine is respected. This approach is too optimistic, the approach of self-stabilization is more promising, recreating the consistency thread from any arbitrary configuration whenever the minimal conditions hold (e.g., less than one-third of the participants are Byzantine).

In this paper, we consider the case of a self-stabilizing robot swarm in the presence of Byzantine robots. Typically, Byzantine robots are assumed to be Byzantine forever. Correct robots may detect and record the identity of Byzantine robots in their variables, so they can ignore or take a countermeasure to the Byzantine robots activities. In the scope of self-stabilization, such records can be set to different records for each participant, where the records of the other participants are not mutually known. For example, starting in a configuration in which each participant has a (wrong) record that all other participants (but itself) were identified as Byzantine. Then all may try to take a countermeasure to all, even if none is actually Byzantine, in fact, nullifying swarm collaboration. Note that the unknown records of the other participants may imply that their observable moves will be regards as Byzantine by others.

Many algorithms are based on failure detectors [3], where each participant lists the suspected participants. Such an abstraction is useful in a fault-prone system, excluding the suspected participants' actions and concentrating on the non suspected participants to gain progress. In our settings, where the program and identifiers of all participants, as well as all actions and all inputs of all participants, are observed by each participant, an indication on a Byzantine participant is immediate following a step that does not obey the program and inputs. Once there is an indication that a participant is Byzantine, other non-Byzantine robots may surround the Byzantine robot and block its movements, allowing the rest of the robots achieve their task with no intervention.

However, in the scope of self-stabilization, the recorded indication on a participant being Byzantine may be corrupt as well, and therefore we propose the *forgive and forget* framework. The period in which the robots remember the indications (before the robots simultaneously forgive and forget) has to be tuned to allow us to perform the task in addition to the possible need of capturing the Byzantine participants.

To cope with the unreliable indication on whether a participant was Byzantine and even in case the indication is reliable, to allow the (self) recovery of a robot (say, by a periodical restart) we propose a forgive and forget approach. We assume that many of the Byzantine participants can recover after a while, say by periodically rebooted, patched with new software parts, or scanned and cleaned from mall-ware. Thus, the impression of one robot that another is Byzantine should be constantly reexamined and verified. Obviously, a robot may suspect all other participants being Byzantine, and if the suspicion is not periodically verified, global collaboration is at risk. This is why, in this research, we will assume that all the robots periodically and (to simplify arguments, use self-stabilizing Byzantine clock synchronization to impose that the robots) simultaneously forget their suspicions in their Byzantine suspicion list.

We will present a method to decide when should the robots forget their suspicions and reset the Byzantine list. The method reset the suspicions in a way that ensures that the robot in the swarm can repeatedly succeed in achieving a useful task in spite of the presence of several Byzantine participants. The task is repeatedly achieved even though many, or even all of the participants were Byzantine in some instances in the past.

The proposed method is based on calculating the minimum continuous time we can suspect a Byzantine robot before we consider the robot as a non Byzantine robot again. Roughly speaking, once the suspicion lists are simultaneously reset, the Byzantine may avoid identifying themselves as such, and while doing so assist in completing the task. Alternatively, the Byzantines deviate from their program and discovered as Byzantine by all non-Byzantine participants. Once discovered, our method suggests to surround and block the Byzantine movement by several robots, and let the other robots complete the task/mission.

We demonstrate our approach by presenting several games. These games are based on periodical restart (respected by the non-Byzantine robots), where every K time units (measured in number of movement steps a robot can take) the non-Byzantine robots reset their Byzantine suspected list and show that K is big enough to ensure that if a robot exhibit Byzantine behavior before the tasks/missions are achieved, then some non-Byzantine robots can catch/block the Byzantine robot while other non-Byzantine robots complete the tasks.

2 Preliminaries

We abstract a geographical region by regarding the geographical region as a (chess) board over which the robots move.

A *board* is defined as a $N \times N$-grid $G = (V, E)$, where $V = \{t_{i,j} \mid 1 \leq i, j \leq N\}$ and $E = \{\{t_{i,j}, t_{i',j'}\} \mid 1 \leq i, i', j, j' \leq N \wedge |i - i'| + |j - j'| = 1\}$. Tiles t and t' are *neighboring* iff $\{t, t'\} \in E$ holds. We define *up*, *down*, *left*, and *right* directions as directions from $t_{i,j}$ to $t_{i,j-1}$, from $t_{i,j}$ to $t_{i,j+1}$, from $t_{i,j}$ to $t_{i-1,j}$, and from $t_{i,j}$ to $t_{i+1,j}$, respectively.

Robots have the following characteristics and capabilities. Robots are anonymous, for the sake of continence, they are assigned by unique IDs by each participant, the chosen ID chosen for a particular robot by other robots may be totally different. Robots have memories where they maintain variables. Robots cannot communicate with other robots explicitly, however they can communicate implicitly by observing positions of all other robots.

In this paper, some robots may be *Byzantine*. Byzantine robots can make arbitrary movements that do not obey the algorithm choice. On the other hand, similarly to non-Byzantine robots, Byzantine robots can move only to its neighboring tile that is not occupied by another robot. We say that a Byzantine robot r on tile t is *captured* if all of neighboring tiles of t are constantly occupied by fixed non moving non-Byzantine robots. Thus, a captured Byzantine robot cannot move unless some robots on its neighboring tiles free their tile.

A *configuration* c is defined as a combination of positions and memory states of all robots. A sequence of configurations $E = c_1, c_2, \dots$ is an *execution* from initial configuration c_1 if, for every $i \geq 1$, c_{i+1} is reached from c_i by a step of all robots. A *problem* $\mathcal{P}$ is defined as a set of *legal executions*, where the set of legal executions consists of executions that exhibit the desired behavior.

Definition 1. (Self-stabilization) *An algorithm $\mathcal{A}$ is a* self-stabilizing *algorithm for $\mathcal{P}$ if there exists a set of* safe configurations C_{safe} *such that both of the following properties hold.*

- Convergence*: Starting in any arbitrary configuration the system eventually reaches a configuration in C_{safe}.*
- Closure*: Any execution that starts from a configuration in C_{safe} belongs to $\mathcal{P}$.*

All robots execute steps synchronously. At the beginning of a step, each robot observes the positions of other robots. Depending on the observation and its own memory, the robot updates the memory and decides on its next movement. If the robot decides to move, it tries to move to the neighboring tile in the decided direction. When robot r_1 tries to move to a tile occupied by r_2, r_1 successfully moves to the tile if r_2 moves to another tile in the same step. If multiple robots try to move to the same tile, one of them moves to the tile and other robots fail to move. In this case, an adversary that chooses the worst scenario for our algorithms, decides which robot moves to the tile. If a robot moves, it completes the movement before the beginning of the next step.

For the synchronization, we employ a self-stabilizing Byzantine pulse clock synchronization algorithm, e.g., [9], that either uses lights [8] or micro-movement to communicate[2,5]. Since the Byzantine robots cannot be two faced, as we assume that every robot observes all robots, then the following simple (non-two-faced) Byzantine clock synchronization will work.

Self-stabilizing Byzantine pulse synchronization. As for the synchronization of the pulses, each participant produces an individual independent pulse series out of which a common pulse series for all the non-Byzantine is produced by the following scheme. Each robot uses a vector *lipv* to record the last time each participants had an individual pulse, so all non-Byzantine robots eventually have an identical recording for each robot that produces individual pulses. If a robot r_b does not produce an individual pulse for *maxPT* period then (it is surely Byzantine, and) all non-Byzantine participants put a special sign $\perp$ in the entry for r_b in their *lipv* vector. Thus, eventually after $maxPT$ all non-Byzantine have an identical values in *lipv*. *lipv* is used to produce common pulses every time a participant produces an individual pulse, as long as the previous pulse this participant was at least *individual pulse period lower bound*, *ipp* time ago. Too close by pulses of different participants are spaced out deterministically by their order in time, simultaneous pulses are ordered according to the identifiers of the robots. Thus, all non-Byzantine participants observe the same sequence of pulses.

Self-stabilizing Byzantine pulse-clock synchronization. Each participant r_i learns the pulse-clock of the other robots, and if there exists a pulse number, j, that appears in a majority of the robots then r_i assigns its pulse-clock to be $cp_i = (j+1)\ mod\ K$, otherwise, r_i assigns $cp_i := 0$.

Longer pulse period can be produced by the self-stabilizing Byzantine pulse-clock synchronization, where the new pulse is identified by pulse number 0. In turn the new spaced pulse can be used as a source for another version of self-stabilizing Byzantine pulse-clock synchronization.

Note that robots cannot read memories of other robots. Robots have a sense of direction, that is, all robots agree on the up, down, left, and right directions. At most one robot can occupy a tile of the board at any given instance.

Cleaning game. In a cleaning game, Byzantine robots contaminate all the tiles of the board, and the non-Byzantine robots cooperate to clean the board infinitely often. Non-Byzantine robots can clear tiles if they visit the tiles. In addition, non-Byzantine robots can stop the contamination behavior of a Byzantine robot by capturing the Byzantine robot. If a Byzantine robot is not captured, it can contaminate all tiles instantaneously by staying at the same tile for even one step. We say the board is clean if all tiles except for tiles occupied by Byzantine robots are clean.

Definition 2. (Cleaning game task) *The cleaning game task is defined by a set of executions $\mathcal{LE}$ such that each E in $\mathcal{LE}$ consists of infinitely many configurations in which the board is clean.*

Creating a fan game. In a creating a fan game, non-Byzantine robots cooperate to create and spin the fan infinitely often. Byzantine robots can interfere in this process by preventing the non-Byzantine robots to create and spin the fan. In addition, non-Byzantine robots can prevent the Byzantine interfering by capturing the Byzantine robot.

Definition 3. (Creating a fan game task) *The creating a fan game task is defined by a set of executions LE such that each E in LE consists of infinitely many configurations in which the fan spin infinitely often.*

3 Cleaning a Ring Board

In this section, we demonstrate the usefulness of our forgive and forget approach by considering a simple cleaning game for the case of a small ring.
Consider a 3×3 board where robots can move only on the outer bound of the board. That is, the board is regarded as a 8-tile ring board. The eight tiles are renamed as $t_1, t_2, \ldots, t_8$ clockwise starting in the left upper tile. Four robots, at most one of which is Byzantine, reside on this ring board.

The ring algorithm in a nutshell. We first describe the conceptual parts of the algorithm depicting the movements with a series of figures.

All robots move clockwise. When robots identify that another robot stops, the robot add the Byzantine robot to the Byzantine list. Each Byzantine robot has two neighbors, r_i and r_j, and their goal is to move to the Byzantine direction from both sides. The third honest robot r_c, moves to the Byzantine direction using the shortest path, if distances are equal, r_c moves clockwise. When the cleaning robot r_c cannot move and the Byzantine robot is blocked, r_c starts to clean the board. Every K steps all robots reset the Byzantine list and continue with cleaning the borders (the board can still be infected from actions taken prior to the reset).
Execution example with no Byzantine. We consider the case in the next series of figures (Fig. 1). The initial configuration is the leftmost one. The figures depict an execution of eight steps of four robots that can be repeated forever, where robots are represented by green tile.

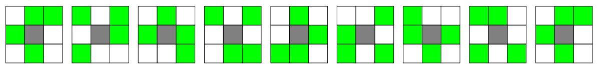


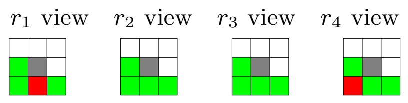
Fig. 1: Ring with four non-Byzantine Robots

We now prove that our algorithm for the ring cleaning must Forgive & Forget as otherwise it Deadlocks.

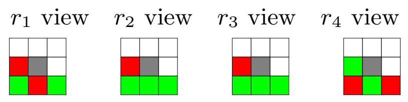
Forgive & Forget – In Fig. 2 we depict with four series of figures four configuration scenario, where robots have different views. The different views of the robots are detailed (views ordered 1-2-3-4). The four configurations form an execution.

Assuming the non-Byzantine robots follow our algorithm in which two non-Byzantine robots (the neighbors of the Byzantine) try to block the Byzantine and the third non-Byzantine robot tries to clean the board (move every step). We now demonstrate the obvious observation, that in an execution E that starts in a configuration in which non-Byzantine participant suspects different robots being Byzantine more and more non-Byzantine robots are suspected being Byzantine. Thus, motivating the need for synchronous forgive and forget that sets all views

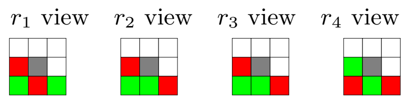
• First configuration: r_1 assumes robot r_2 is Byzantine (Byzantine already blocked by robots r_1 and r_3) and waits for r_4 to clean the board. Robots r_2 and r_3 do not suspect any other robot try to move clockwise but cannot as r_4 does not move from blocking r_3 that r_4 suspects. Namely, r_4 assumes robot r_3 is Byzantine (Byzantine already blocked by robots r_4 and r_2) and waits for r_1 to clean the board.



• Second configuration: r_1 and r_4 suspect two robots being Byzantine



• Third and fourth configurations: r_2 and r_3 also suspect two other robots being Byzantine.



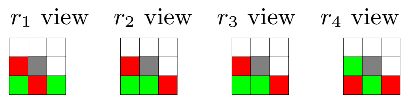


Fig. 2: Different suspicion views lead to more suspicions

of non-Byzantine to be identical. In the first configuration of E two robots r_1 and r_4 suspect different robots to be Byzantine (r_2 and r_3, respectively). Since the algorithm is defined to cope with at most one Byzantine, the actions of the non-Byzantine when more than one robots (say all but itself) is suspected maybe arbitrary (say they stop) then there maybe an infinite suffix where the cleaning requirement does not hold.

In the execution Initial non-Byzantine robots are depicted by green tiles, while Byzantine robot by a red tile. Our description assumes that the ID of a robot is determined starting from the upper left corner and continuing clockwise.

Any algorithm in which a robot stops moving if it suspects more than one robot to be Byzantine, and any suspicion lasts forever, is not self-stabilizing, as the initial configuration can be one in which all robots suspect more than one and no robot moves subsequently. Hence, the first configuration is a deadlock configuration. Moreover, in particular cases as depicted in the execution above, when participants start suspecting different participants, (while non are actually Byzantine), as the suspicion list is unknown to the other participants, the actions

```
Algorithm 1: Cleaning the Ring on the Board (continued)
  Input: An integer $K$. Represent the number of steps before the robots forget.
  Parameters: Robot variables: MyID, $K$, NumberOfByzantine,
              ByzantineLocation, ByzantineIndex, RobotList (a list of 4
              robots) - for each robot in the list: Id, IsByzantine,
              CurrentLocation, NextLocation, PreviousLocation
  Functions :
1 Initialization():
    Initialize all variables to null or 0.
2 Look():
    Look and locate all the robots starting with the left upper corner (location 0).
    The first robot located get the id 0, the next one is 1, etc.
3 ComputeNoByzantine():
    Which compute for each robot the next move – clockwise (location + 1 mod
    8). Return a list of robots next move
4 ComputeWithByzantineNotBlocked():
    Compute for each robot the next location. For each Byzantine neighbor – the
    non-Byzantine robot closest to the Byzantine from both sides, tries to move
    one step to the Byzantine location, for the cleaning robot, move to the
    Byzantine with shortest path, if equal, go clockwise. Return a list of robots
    next move
5 ComputeWithByzantineBlocked():
    Compute for each robot the next location. for each Byzantine neighbor, do not
    move, for the cleaning robot continue the clean, if block, change direction.
    Return a list of robots next move
6 ComputeWithMoreThanOneByzantine():
    Which computes for each robot the next move – do not move. Returns a list of
    robots next moves
7 FindByzantine() :
    Return the number of Byzantine (which robot did not get to the expected
    location), ByzantineId and ByzantineLocation. For each robot, validate that
    all robots move to the expected location and count the Byzantine
8 UpdateK() :
    Increase $k$ by 1 and if equal to $K$ initialize $k$ and $NumberOfByzantine$ to 0
9 ValidateByzantineIsBlocked() :
    If Byzantine is blocked return True else False
```

of the non-Byzantine robots can be interpreted as an action that does not follow the algorithm. Thus, yields a reason to increase the suspected participants even beyond the maximal number of Byzantine required for stabilization, yielding a need to forgive and forget.

Next we show that an algorithm exists when forgive and forget is employed.

Algorithm 2 forgets after K steps. The algorithm loops forever and assigns the variables, When getting a global pulse, each robot looks and computes the next steps. If the number of Byzantine equal to 0 the robot computes the next move which is cleaning. If we identify more than 1 Byzantine all robots must stop.

Algorithm 2: Cleaning the Ring on the Board – forget after K steps

```
$Initialization()$:
**Global Pulse:** Upon a global pulse move to the next location
$Look()$
$NumberOfByzantine = MAX(NumberOfByzantine, FindByzantine())$
**if** $NumberOfByzantine = 0$ **then**
    $ComputeNoByzantine()$
**else**
    // **Byzantine exist**
    **if** $NumberOfByzantine > 1$ **then**
        $ComputeWithMoreThanOneByzantine()$
    **else**
        // **One Byzantine exists**
        **if** $ValidateByzantineIsBlocked() = True$ **then**
            $ComputeWithByzantineBlocked()$
        **else**
            $ComputeWithByzantineNotBlocked()$
$UpdateK()$
Go To Global Pulse
```

If we had only 1 Byzantine and he is blocked already, only the cleaning robot needs to clean the board, Otherwise, the robots must capture the Byzantine.

In Fig. 3–5, each non-Byzantine robot is represented by a green tile, each detected Byzantine robot is represented by a red tile, Byzantine robot that acts correctly so far (thus, sill not detected as Byzantine) is represented by a yellow tile. Contaminated tiles are marked with diagonal lines and cleaned tiles marked as white tiles.

Fig. 3 depicts the following situations:

(01) Initial configuration,

(02) non-Byzantine robots move clockwise, Byzantine stops (yielding an immediate detection by non-Byzantine robots) and contaminates the board,

(03-04) non-Byzantine robots try to block (the board is still infected),

(05) non-Byzantine robots blocked the Byzantine (the board is still infected),

(06-08) Two robots blocked the Byzantine, one robot cleans the board,

(09) Two non-Byzantine robots blocked the Byzantine, the board is clean,

(10) Two non-Byzantine robots blocked the Byzantine until resets.

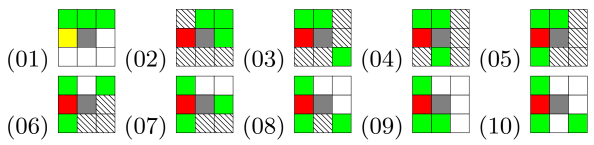


Fig. 3: Capturing the Byzantine and cleaning

In Fig. 4, we now demonstrate that when $K = 5$ the Byzantine wins, as the non-Byzantine robots are unable to fulfill their cleaning task. Notice that the non-Byzantine robots forgive and forget, when the Byzantine robot changes color from red to yellow. Let k $(0 \leq k \leq K)$ be a variable to countup until the resets.

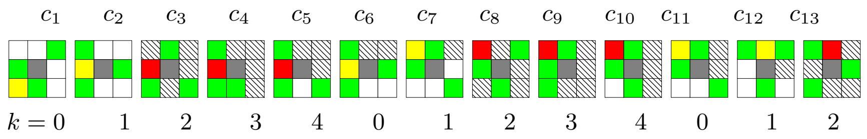

c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13}
$k = 0$ 1 2 3 4 0 1 2 3 4 0 1 2

Fig. 4: Byzantine may prevent cleaning forever

Lemma 1. *When our swarm algorithm is applied to a ring board with four robots, one of which is Byzantine, if the (non-Byzantine) robots forget when $K = 5$, then the cleaning requirement does not hold.*

Proof. We demonstrated a scenario which the third configuration and the thirteenth configuration are the same with an indentation of two steps with $k = 2$. This scenario can be reproduced by Byzantine robot infinitely often, preventing the non-Byzantine robots to complete the cleaning task.

The following details follow the above depicted execution.

c_1: $k = 0$

t_1, t_2, t_4, t_5 are empty. t_3, t_6 and t_8 are occupied by non-Byzantine robots. t_7 is occupied by a Byzantine robot (marked as non-Byzantine robot by all robots).

c_2: $k = 1$

All robots move as expected – Byzantine list is empty.

c_3: $k = 2$

r_3 does not move and stay at t_8. All robots mark r_3 as r_b.

c_4: $k = 3$

r_4 move to t_1 and block the Byzantine. r_1 move to t_5.

c_5: $k = 4$

r_1 move to t_5 and clean the board.

c_6: $k = 0$

r_1 move to t_4 and clean the board. After this step, all robots forget the forgive and the Byzantine list is empty (but they did not complete the task yet as t_2 and t_3 were not cleaned).

c_7: $k = 1$

All $r_1 - r_4$ move clockwise as expected t_3 is not clean.

c_8: $k = 2$

r_3 does not move and all robots mark r_3 as r_b. t_1 is occupied by r_3 (r_b). t_3 is occupied by r_4, t_6 is occupied by r_1, t_8 is occupied by r_2.

c_9: $k = 3$

r_4 moves to t_2, r_1 moves to t_7.

c_{10}: $k = 4$

r_1 moves to t_5 cleaning the board.

c_{11}: $k = 0$

r_1 moves to t_4 cleaning the board. After the step all robots forgive and forget setting the Byzantine lists to be empty (but they did not complete the task yet as t_3 and t_4 are not clean).

c_{12}: $k = 1$

All robots r_1, r_2, r_3 and r_4 move clockwise as expected, t_4 is not clean.

c_{13}: $k = 2$

r_3 does not move and all robots mark r_3 as r_b. t_1 is occupied by r_1. t_2 is occupied by r_3 (r_b), t_4 is occupied by r_4, t_7 is occupied by r_1.

c_{13} and c_3 are the same with an indentation of two steps with $k = 2$. □

Lemma 2. *When our swarm algorithm is applied to a ring board with four robots, one of which is Byzantine, if the (non-Byzantine) robots forget when* $K = 15$*, then the cleaning requirement holds.*

Sketch of proof. We first establish that at most four steps are needed to block the Byzantine robot since non-Byzantine robots detect it. Fig. 5 represents an example of such execution (in the second configuration from left, the Byzantine is detected, and in the first configuration from right, it is blocked).

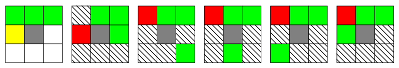


Fig. 5: Blocking the Byzantine

Next we show that we can clean the board after blocking, which takes, in the worst case scenario, five steps. Fig. 6 represent an example of such execution (cleaning starts in the first configuration from left).

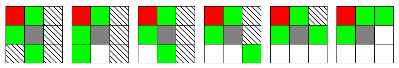


Fig. 6: Cleaning after blocking the Byzantine

Cleaning the board with no Byzantine can take up to four steps. Assume all four robots create a row, and there are four free tiles to clean. Consider the case that the robots forgive and forget while the board is not clean. If the Byzantine moves four steps the non-Byzantine robots will be able to complete the task. The non-Byzantine robots will be able to block the Byzantine after five more steps and then cleaning the board in five more. If we choose K to be equal to fifteen the robots will be able to complete the task infinitely often. □

Proof. The proof considers the following cases.

Cleaning with no Byzantine.

Denote the longest sequence of empty tiles by s.

- In case $s = 1$ the robots will be able to clean the board in one step. c_i: t_2, t_4, t_6 and t_8 are occupied by non-Byzantine robots. t_1, t_3, t_5, t_7 are empty.
  All robots move one clockwise c_{i+1}: t_1, t_3, t_5 and t_7 are occupied by non-Byzantine robots. t_2, t_4, t_6, t_8 are empty. The board is clean.
- In case $s = 2$ the robots will be able to clean the board in two steps. c_i: t_1, t_2, t_5 and t_7 are occupied by non-Byzantine robots. t_3, t_4, t_6, t_8 are empty. c_{i+1}: t_2, t_3, t_6 and t_8 are occupied by non-Byzantine robots. c_{i+2}: t_3, t_4, t_7 and t_1 are occupied by non-Byzantine robots. The board is clean.
- In case $s = 3$ the robots will be able to clean the board in three steps. c_i: t_1, t_2, t_3 and t_7 are occupied by non-Byzantine robots. t_4, t_5, t_6, t_8 are empty. c_{i+1}: t_2, t_3, t_4 and t_8 are occupied by non-Byzantine robots. c_{i+2}: t_3, t_4, t_5 and t_1 are occupied by non-Byzantine robots. c_{i+3}: t_4, t_5, t_6 and t_2 are occupied by non-Byzantine robots. The board is clean.
- In case $s = 4$ the robots will be able to clean the board in four steps. c_i: t_1, t_2, t_3 and t_4 are occupied by non-Byzantine robots. t_5, t_6, t_7, t_8 are empty. c_{i+1}: t_2, t_3, t_4 and t_5 are occupied by non-Byzantine robots. c_{i+2}: t_3, t_4, t_5 and t_6 are occupied by non-Byzantine robots. c_{i+3}: t_4, t_5, t_6 and t_7 are occupied by non-Byzantine robots. c_{i+4}: t_5, t_6, t_7 and t_8 are occupied by non-Byzantine robots. The board is clean.

Cleaning while the Byzantine is Blocked.

- Assume without loss of generality that the Byzantine is blocked at t_7 with non-Byzantine robots at t_6 and t_8 and the cleaning robot locate at t_i to start the cleaning. For each $i = 0 \ldots 8$ the route of the robot as follow:
- t_1, t_2, t_3, t_4, t_5 – the board is clean after four steps
- t_2, t_1, t_2, t_3, t_4, t_5 – the board is clean after five steps
- t_3, t_4, t_5, t_4, t_3, t_2, t_1 – the board is clean after six steps
- t_4, t_5, t_4, t_3, t_2, t_1 – the board is clean after five steps
- t_5, t_4, t_3, t_2, t_1 – the board is clean after four steps. Worst case scenario, the time for cleaning the board with Byzantine is six.

Blocking – Maximum time to block a Byzantine is four steps
The total number of tiles is eight and the number of robots is four. For every Byzantine neighbor, at least one of them decrease the distance to the Byzantine in one step, where *distance* is the number of empty tiles between a neighbor robot and the Byzantine, where no other robots in the way.

- d_i – The number of empty tiles between r_i and the Byzantine r_b where no other robots in the way.
- If d_i is bigger than or equal to two then r_i moves one step towards the Byzantine and decrease d_i by 1.

– If $d_i = 1$, then if r_b moves one step to r_i, d_i still equal to 1. Else (r_b stays at t_i then r_i moves one step towards the Byzantine and $d_i = 0$.
– If $d_i = 0$, then if r_b moves one step away from r_i, then r_i will move to the same direction of the Byzantine and $d_i = 0$. Else if r_b stays in t_i, r_i stays too and $d_i = 0$.
– Now, consider two neighbors with distances d_1 and d_2.
– If d_1=0 and d_2=0 then the Byzantine blocked.
– If d_1=0 and d_2=1 (without loss of generality), if r_b stays in the same tile after one step then d_1=0 and d_2=0. If r_b moves towards the direction of r_2, then r_1 moves in the same direction and both d_1=0 and d_2=0.
– If d_1=0 and d_2 is bigger than or equal to two (without loss of generality), if r_b stays in the same tile then after one step d_1=0 and d_2=d_2-1. If r_b moves towards r_2 direction, then r_1 moves in the same direction yielding d_1=0 and d_2=d_2-1. In case d_1=1 and d_2=1, if r_b stays in the same tile after one step then d_1=0 and d_2=0. If r_b moves to r_2 direction (without loss of generality), then d_1=1 and d_2=0.
– If d_1=1 and d_2 is bigger than or equal to two (without loss of generality), if r_b stays in the same tile then d_1=0 and d_2=d_2−1. If r_b moves towards r_2, then $d_1 = 1$ and $d_2 = d_2 - 1$. If r_b moves towards r_1, then $d_1 = 0$ and d_2 is not changed. If d_1 is bigger than or equal to two and d_2 is bigger than or equal to two, then for any possible Byzantine movement it holds that, $d_1 = d_1 - 1$ and $d_2 = d_2 - 1$.
– We proved that every step reduces at least one distance by 1. $d_1 + d_2$ is less than or equal to four (the free tiles in the board). After 4 steps $d_1 + d_2 = 0$ and the Byzantine is blocked. □

4 Full Board Game

In this section, we demonstrate the usefulness and generality of our forgive and forget approach by considering another cleaning game example, where a full board rather than a ring is considered.

Consider a 8×8 board where robots can move freely on the board (including the center). That is, the board is regarded as a 64-tile board. The tiles are named $t_{1,1}, t_{1,2} \ldots, t_{8,8}$, where the first index is the row of the board and the second index is the column of the board. We further assume that there at least 9 robots at most one of which is Byzantine.

The Full board game algorithm in a nutshell. We first describe the conceptual parts of the algorithm depicting the movements with a series of figures.
• *Cleaning.*

If no Byzantine robots are detected, robots move in a ”snake” fashion to clean the board.

(1) Robots on the first row go right until the end of the row. Namely, a robot located at $t_{1,j}$, $j < 8$, moves to $t_{1,j+1}$ if possible.

(2) A robot on $t_{1,8}$ moves to $t_{2,8}$ if possible.

(3) For even j and $i \in \{2, 3, \ldots, 7\}$, a robot on $t_{i,j}$ moves to $t_{i+1,j}$ if possible.

(4) For even j, a robot on $t_{8,j}$ moves to $t_{8,j-1}$ if possible.

(5) For odd j and $i \in \{3, \ldots, 8\}$, a robot on $t_{i,j}$ moves to $t_{i-1,j}$ if possible.

(6) For odd j, a robot on $t_{2,j}$ moves to $t_{2,j-1}$ if possible.

(7) A robot on $t_{2,1}$ moves to $t_{1,1}$ if possible.

Fig. 7 depict execution example 8×8 with no Byzantine.

The figures depict an execution of seven steps that can be continued forever, where robots are represented by green tile.

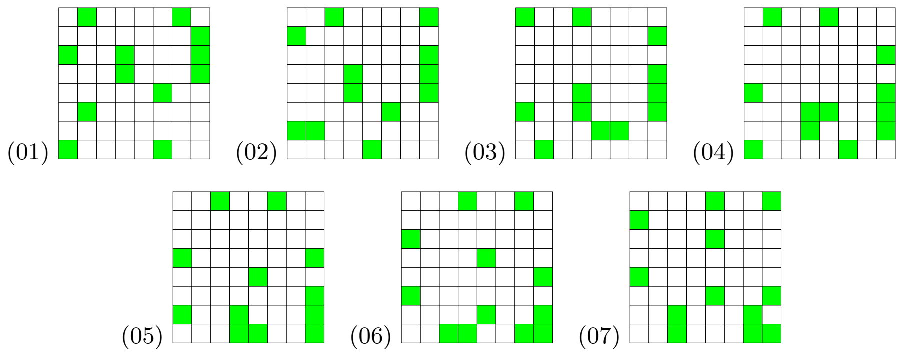


Fig. 7: Full board cleaning with no Byzantine

• *Blocking.*

If robots detect a Byzantine robot, they move to block the Byzantine robot. Simply put, robots first create a perfect row at the middle row and then move toward the Byzantine robot. We explain the details below.

(1) In order to create a perfect row at the middle row (or fifth row) of the board, each robot tries to move to the column tile that intersects the column the robot resides and the middle row.

(2) If the middle row (in the same column of the robot) is occupied by another robot, the robot identifies whether there is a spare column with no robot in the middle row. If such a column exists, the robot tries to fill in the closest tile (if distances are equal then moves to the right).

(3) If Byzantine blocks the row of non-Byzantine robots, the non-Byzantine robot that is not part of the row will block the Byzantine by occupying the tile near the Byzantine. Assume without loss of generality the Byzantine blocks $t_{5,4}$. In this case, the non-Byzantine robot that is not part of the row moves to $t_{6,4}$ or $t_{4,4}$ depending on its initial position.

(4) If Byzantine blocks the next row such that non-Byzantine robots cannot create a row, the non-Byzantine robots try to move close to the Byzantine until the non-Byzantine robot that is not part of the row is able to complete the row. Assume without loss of generality that the non-Byzantine robots create the row

on $t_{5,i}$ except $t_{5,4}$. Byzantine can move to tile $t_{6,4}$ and the robot that is not part of the row move to $t_{6,5}$. In the next move, robots on $t_{5,5}$ and $t_{6,5}$ try to move left and at least one of them succeeds. If the robot on $t_{5,5}$ succeeds and moves to $t_{5,4}$ in the next move the robot on $t_{6,5}$ moves to $t_{5,5}$ and the row is completed. If the Byzantine moves to $t_{5,4}$, the non-Byzantine robot on $t_{6,5}$ moves to $t_{5,4}$ and it is the same configuration as (3).

(5) If the row is missing one tile because of the Byzantine, both robots on the same column try to move to the Byzantine direction.

(6) If the robots created a perfect row, all robots move one step in the Byzantine direction until the Byzantine is part of the row.

(7) If the Byzantine is part of the row, the upper robot tries to move down, the sides robots do not move and all other robots try to move to the Byzantine direction if the tile is free, or one step down if not free.

(8) Every K steps all robots reset the Byzantine list and continue with the cleaning (the board can still be infected from actions taken prior to the reset).

Blocking example - Creating a row.

Fig. 8 depicts the following situations:

(01) Initial configuration,

(02) robots move on the same column to create the row, robot on tile $t_{6,3}$ cannot move up as the another robot occupied tile $t_{5,3}$ so the robot move to $t_{6,2}$

(03) robots move on the same column to create the row.

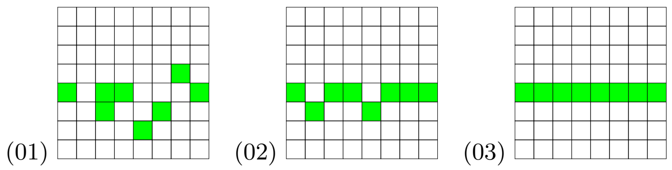


Fig. 8: Creating a row

Blocking example - Byzantine interfere the row creation. If Byzantine blocks the row or the next row, robots try to move until the row is completed or only one tile is missing.

Fig. 9 depicts the following situations:

(01) Initial configuration, Byzantine block robot on tile $t_{6,3}$, row is not completed

(02) Byzantine block robot on tile $t_{6,3}$ and the row is completed except tile $t_{5,2}$.

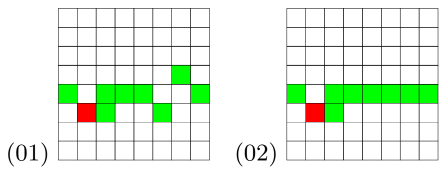


Fig. 9: Byzantine blocks the row creation

If the row is missing one tile because of the Byzantine, both robots on the same column try to move to the Byzantine direction.

• Scenario 1 – Byzantine does not move (See Fig. 10.).

(02) Byzantine block robot on tile $t_{6,3}$ and the row is completed except tile $t_{5,2}$. Both robots on tiles $t_{5,3}$ and $t_{6,3}$ tries to move left.

(03) Byzantine does not move, so the robot from $t_{5,3}$ move to tile $t_{5,2}$

(04) Robot from $t_{6,3}$ moves to $t_{5,3}$, row is completed.

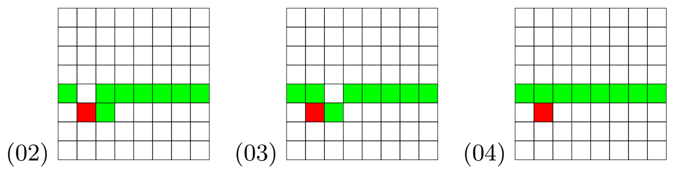


Fig. 10: Byzantine does not move scenario

• Scenario 2 – Byzantine move and blocked the row (See Fig. 11.).

(02) Byzantine block robot on tile $t_{6,3}$ and the row is completed except tile $t_{5,2}$. Both robots on tiles $t_{5,3}$ and $t_{6,3}$ tries to move left.

(03) Byzantine moves to tile $t_{5,2}$, so the robot from $t_{6,3}$ move to tile $t_{6,2}$

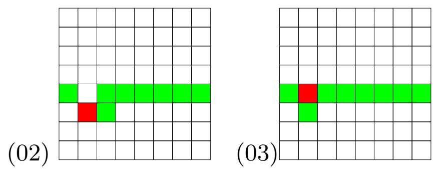


Fig. 11: Byzantine moves scenario

The Blocking Algorithm – Blocking examples. If the robots created a perfect row, all robots move one step in the Byzantine direction until the Byzantine is part of the row (See Fig. 12.).

(01) Initial configuration – robots created a perfect row.

(02) Robots move 1 step towards the Byzantine direction.

(03) Robots move 1 step towards the Byzantine direction.

(04) Robots move 1 step towards the Byzantine direction, robot on tile $t_{7,2}$ cannot move (the robot blocks the Byzantine).

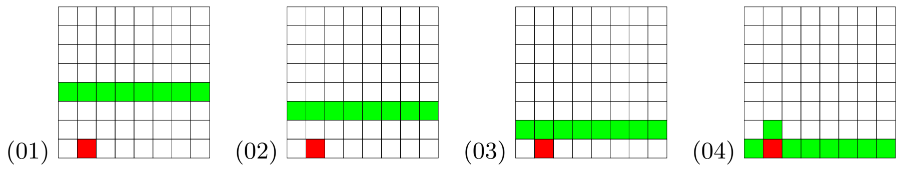


Fig. 12: Robots create a perfect row

If the Byzantine is part of the row (assuming we block the Byzantine from the upper side (as in the example bellow) the upper robot tries to move down, the sides robots do not move and all other robots try to move to the Byzantine direction if the tile is free, or move down one step, otherwise (See Figure 13.).

(01) The Byzantine is part of the row.

(02) The upper robot $t_{6,2}$ tries to move down, the robots on the sides of the Byzantine $t_{7,1}$ and $t_{7,3}$ do not move and all other robots move to down.

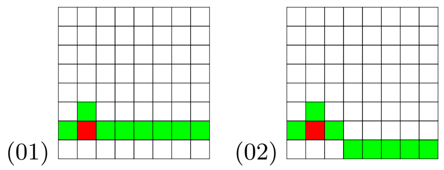


Fig. 13: Byzantine part of the row

• Scenario 1 – Byzantine does not move (See Fig. 18.).

(03) Byzantine does not move. The upper robot $t_{6,2}$ tries to move down, the robots on the sides of the Byzantine $t_{7,1}$ and $t_{7,3}$ do not move and all other robots move left.

• Scenario 2 – Byzantine move and blocked the row (See Fig. **??**.).

(04) Byzantine moves down to $t_{8,2}$. The upper robot $t_{6,2}$ moves down to $t_{7,2}$, the robots on the sides of the Byzantine $t_{7,1}$ and $t_{7,3}$ do not move and all other robots move left. (05) Byzantine does not move. Robot from $t_{7,1}$ moves to $t_{8,1}$,

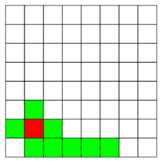
(03) 

Fig. 14: Byzantine does not move

robots from $t_{7,2}$ and $t_{7,3}$ try to move down (and failed). All other robots try to move left. Byzantine is blocked.

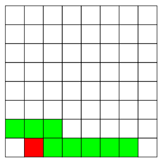
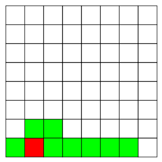
(04)  (05) 

Fig. 15: Byzantine moves down

In case the robots blocked the Byzantine and an empty tile in the corner is exist that the robots can't clean, the robots must move to the corner before the cleaning.
without loss of generality, assume the Byzantine is in tile $t_{7,2}$, the closest honest robot to $t_{8,8}$ move to $t_{6,1}$ or $t_{8,3}$. Assume without loss of generality that the robot is at $t_{8,3}$. Robot at $t_{8,2}$ tries to move to $t_{8,1}$, robot at $t_{8,3}$ tries to move to $t_{8,2}$ and robot at $t_{6,2}$ tries to move to $t_{8,2}$.

(01)

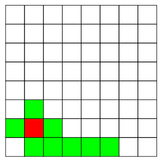


Fig. 16: An empty tile in the corner

Scenario 1 – Byzantine does not move
(02) Byzantine on tile $t_{7,2}$ and the honest robots at $t_{8,1}$ and $t_{8,2}$ (all other honest robots stay on the same tile).

(02)

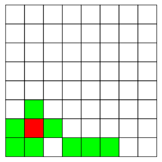


Fig. 17: Byzantine does not move

Scenario 2 – Byzantine moves down
(02) Byzantine on tile $t_{8,2}$ and the honest robots at $t_{8,1}$ and $t_{7,2}$ (all other honest robots stay on the same tile).

(02)

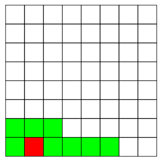


Fig. 18: Byzantine moves down

Algorithm 3: Cleaning the Board (continued)

```
Input: An integer $K$. Represent the number of steps before the robots forget.
Parameters: Robot variables: MyID, $k$, NumberOfByzantine,
            ByzantineLocation, ByzantineIndex, RobotList (a list of $N+1$
            robots) – for each robot in the list: Id, IsByzantine,
            CurrentLocation, NextLocation, PreviousLocation
Functions :
1 Initialization()
    Initialize all variables to null or 0.
2 Look():
    Look and locate all the robots starting with the left upper corner (location 0).
    The first robot located get the id 0, the next one is 1, etc.
3 ComputeNoByzantine():
    Which compute for each robot the next move (snake).
    If robot is on the upper row move right one step
    If robot is on the upper right corner go down one step
    If robot is on even column go down
    If robot is on the lower row go left
    If robot is on odd column go up one step until the second upper line
    For the case of even board: If robot is on the second upper row go left
    For the case of odd board: If robot is on the second row second column: if all
    tiles in the left column are empty, the robot move left, otherwise up. If robot
    on the left column, do down until the end and then up.
    If robot is on the second row left side go one step up
    Return a list of robots next move
4 ComputeWithByzantineBlocked():
    Compute for each robot the next location. For each Byzantine neighbor, do not
    move, for the cleaning robot continue the clean. If next tile is occupied and you
    can bypass the Byzantine without interfere other non-Byzantine robots find
    the shortest path to the next not occupied location. Otherwise, move up and
    down until you can bypass without interfering to other non-Byzantine robots
5 UpdateK() :
    Increase $k$ by 1. If $k$ equals $K$ initialize $k$ and NumberOfByzantine to 0
```

Algorithm 4 first assign variables (line 1). When getting a global pulse (line 2), each robot looks and computes the next steps. If the algorithm identifies Byzantine robot (line 8) algorithm 5 is called and the robot computes the next move which is cleaning with Byzantine (line 10).

Algorithm 4: Cleaning the Board – forget after K steps

```
1  $\mathcal{I}nitialization()$:
2  **Global Pulse:** Upon a global pulse move to the next location
3  $\mathcal{L}ook()$:
4  $NumberOfByzantine = MAX(NumberOfByzantine, FindByzantine())$
5  if $NumberOfByzantine = 0$ then
6      $ComputeNoByzantine()$
7  else
8      // Byzantine exist
9      if (Call Algorithm 5 return True) then
10         $ComputeWithByzantine()$
11 $UpdateK()$
12 Go To Global Pulse
```

Algorithm 5: The Blocking Algorithm (continued)

```
Input: An integer $K$. Represent the number of steps before the robots forget.
Parameters: Robot variables: MyID, $k$, NumberOfByzantine,
            ByzantineLocation, ByzantineIndex, RobotList (a list of 4
            robots) – for each robot in the list: Id, IsByzantine,
            CurrentLocation, NextLocation, PreviousLocation
Functions :
1 $ComputeWithByzantineNotBlocked()$:
    Compute for each robot the next location. If the robots created a perfect row,
    all robots move one step in the Byzantinedirection until the Byzantine is part
    of the row.
    If the Byzantine is part of the row, the upper robot tries to move down, the
    sides robots do not move and all other robots try to move to the
    Byzantinedirection if the tile is free, or one step down if not free.
2 $CreateARow()$:
    Compute for each robot the next location and create a row at the middle of
    the board.
    The robot moves step by step in the same column until the middle.
    If the middle row (in the same column of the robot) is blocked, the robot
    identifies whether there is a spare column with no robot, if so, the robot tries
    to block the closest tile (if equal then moves to the right).
    If Byzantine blocks the row or the next row, robots try to move until the row
    is completed or one tile is missing.
    If the row is missing one tile because of the Byzantine, both robots on the
    same column try to move to the Byzantine direction
3 $ComputeWithMoreThanOneByzantine()$:
    Which compute for each robot the next move – do not move. Return a list of
    robots next move
4 $FindByzantine()$ :
    Return the number of Byzantine (which robot did not get to the expected
    location), ByzantineId and ByzantineLocation. For each robot, validate that
    all robots move to the expected location and count the Byzantine
5 $UpdateK()$ :
    Increase $k$ by 1 and if $k$ becomes equal to $K$ initialize $k$ and $N$ numberOf
    Byzantine to 0
6 $ValidateByzantineIsBlocked()$ :
    If Byzantine is blocked return True else False
```

Algorithm 6: The Blocking Algorithm

```
// Byzantine exist
if $NumberOfByzantine > 1$ then
    $ComputeWithMoreThanOneByzantine()$
else
    // One Byzantine exists
    if $ValidateByzantineIsBlocked() = True$ then
        Return True
    else
        if $ValidRow() = True$ then
            $ComputeWithByzantineNotBlocked()$
        else
            $CreateARow()$
Return False
```

Algorithm 6 starts in identifying the number of Byzantine (line 1), if we have more than 1, all robots must stop (line 3). If the algorithm identify 1 Byzantine which is blocked, the algorithm return True, Otherwise the robots trying to block the Byzantine by creating a row (line 12).

Lemma 3. *When our swarm algorithm is applied to a general board with* 9 *robots, one of which is Byzantine, if the (non-Byzantine) robots forget when* $K = 171$*, then the cleaning requirement holds.*

Proof. The proof is partitioned into several stages of progress.

Cleaning with no Byzantine. The worst case scenario to clean the general board is when all robots are neighbors. The board size is 64 tiles so the number of steps to clean the board is 64 - 9 = 55

Creating a row. The maximum steps to move between two points on the board is 16. In order to create the row, each robot move less (or equal) to 16 steps.

Blocking the Byzantine. Assume the non-Byzantine robots create a row (or a row with the Byzantine interfering), the number of free tiles (on the side of the Byzantine or the side of the non-Byzantine robot in case the Byzantine interfere) is 32. Every step we decrease that number. Assume we decrease that number by one every step (we actually decrease that number by much more than one), it will take 32 steps to block the Byzantine.

Cleaning with Byzantine. To bypass the Byzantine and the blockers we need maximum 8 steps. In the worst case scenario, the number of non-Byzantine that can clean the board is 5 (9 - 4). So the number of steps to clean the board with Byzantine is 64 - 5 (cleaning with no Byzantine) + 8 = 67

Consider the case that the robots forgive and forget while the board is not clean. If the Byzantine moves 55 steps the non-Byzantine robots will be able to complete the task. The non-Byzantine robots will be able to block the Byzantine after 48 more steps and then cleaning the board in 67 more. If we choose K to be equal to $55 + 48 + 67 + 1 = 171$ the robots will be able to complete the task infinitely often. □

5 Creating a fan

In this section, we demonstrate the usefulness of our forgive and forget approach and the generality of the Byzantine capturing policy, by considering a new game in which the robots must create and rotate a fan. We consider only the case of a full board.

Creating a fan An execution of a robot swarm algorithm satisfies the creation of a fan iff the robots create a perfect fan with robots (without loss of generality) on $t_{1,2}, t_{2,2}, t_{3,2}, t_{2,1}$ and $t_{2,3}$ and rotate it to $t_{1,3}, t_{2,2}, t_{3,1}, t_{1,1}$ and $t_{3,3)}$ on a full board based on the general game board.

A Byzantine robot can easily prevent the non-Byzantine robots in creating and rotating the fan by moving to a free tile which is a part of the fan or to the tile planned for the next fan move.

The fan game algorithm in a nutshell. We first describe the conceptual parts of the algorithm depicting the movements with a series of figures.

In order to create and spin the fan, robots identify the blocked Byzantine robot and its location. The non-Byzantine validate if they can create a fan on the first quarter, if they cannot, they will validate the second quarter and so on. Assume without loss of generality that the Byzantine is blocked in the first quarter, the robots will create the fan on the second quarter: $t_{1,7}$, $t_{2,6}$, $t_{2,7}$, $t_{2,8}$ and $t_{3,7}$. In the next steps the robots rotate the fan to $t_{1,6}$, $t_{1,8}$, $t_{2,7}$, $t_{3,6}$ and $t_{3,8)}$ and continues in the same manner. The five closest robots to the desired location of the fan move. These robots first move horizontally and then vertically.

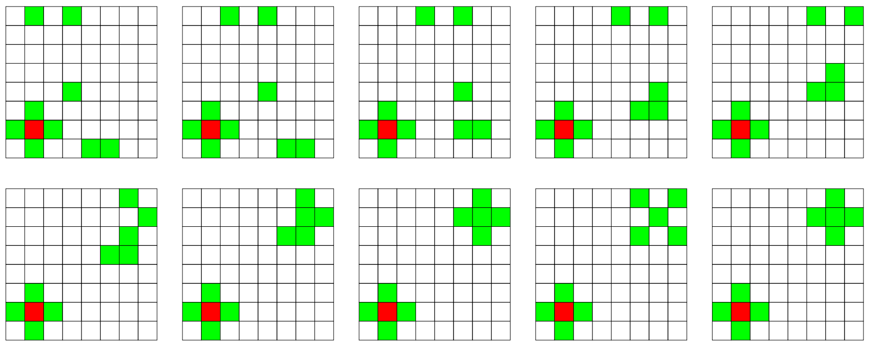


Fig. 19: Fan in the presence of Byzantine

Algorithm 7: Creating a fan

```
   Functions :
1  $Initialization()$
     Initialize all variables to null or 0.
2  $Look()$:
     Look and locate all the robots starting with the left upper corner (location 0).
     The first robot located is assumed to have the id 0, the next one is 1, etc.
3  $ComputeNoByzantine()$:
     Which compute for each robot the next move.
     Create a fan on the left upper quarter (first quarter).
     The five closest robots to the fan desire location move, the robots move
     horizontally first and then vertically.
4  $ComputeWithByzantine()$:
     Which compute for each robot the next move.
     Find in which quarter the Byzantine or his blockers are located and choose the
     first quarter without any of them. Start with the first quarter, then the second
     quarter and so on.
     The five closest robots to the fan desire location move, the robots move
     horizontally first and then vertically. In case a honest robot blocked in a corner
     assume without loss of generality on $t_{8,1}$, the the blocked and the blocker (on
     the same row $t_{8,2}$ move together to $t_{8,2}$ and $t_{8,3}$ respectively
5  $FindByzantine()$ :
     Return the number of Byzantine (which robot did not reach the expected
     location), ByzantineId and ByzantineLocation. For each robot, validate that
     all robots move to the expected location and count the Byzantine
6  $UpdateK()$ :
     Increase $k$ by 1. If $k$ equals $K$ initialize $k$ and NumberOfByzantine to 0
```

Algorithm 8: Creating a fan (continued)

```
1  $\mathcal{I}nitialization()$:
2  **Global Pulse:** Upon a global pulse move to the next location
3  $\mathcal{L}ook()$:
4  $NumberOfByzantine = MAX(NumberOfByzantine, FindByzantine())$
5  **if** $NumberOfByzantine = 0$ **then**
6      $ComputeNoByzantine()$
7  **else**
8      // **Byzantine exist**
9      **if** *(Call Algorithm 5 return True)* **then**
10         $ComputeWithByzantine()$
11 $UpdateK()$
12 Go To Global Pulse
```

Algorithm 8 starts with variables assignment (line 1). Upon a global pulse (line 2), each robot looks and computes the next steps. If the algorithm identifies Byzantine robot (line 8) algorithm 5 is called and the robot computes the next move which is creating and moving the fan (line 10).

Lemma 4. *When our swarm algorithm is applied to create a fan on the general board with* 10 *robots, one of which is Byzantine, if the (non-Byzantine) robots forget when* $K = 91$*, then the fan creation and spinning requirement holds.*

Proof. The proof considers several stages of progress.

Creating a fan with no Byzantine. The maximum steps to move between two points on the board is 16. In order to create the fan, each robot move less (or equal) to 16 steps. Spinning the fan is 1 more step.

Creating the row and blocking the Byzantine. Same as the general algorithm - 48 steps

Creating a fan with Byzantine. To bypass the Byzantine and the blockers we need maximum 8 steps. In the worst case scenario, the number of steps to create the fan with Byzantine is 17 (creating and spinning the fan with no Byzantine) + 8 = 25

Consider the case that the robots forgive and forget while before the fan span. If the Byzantine moves as the algorithm more than 17 steps the non-Byzantine robots will be able to complete the task. The non-Byzantine robots will be able to block the Byzantine after 48 more steps and then creating and spinning the fan with 25 more. If we choose K to be equal to 17 + 48 + 25 + 1 = 91 the robots will be able to complete the task infinitely often. □

6 Conclusions

In this research, we consider the swarm of robots collaborating on a mission. Swarms consisting of many autonomous robots should be designed to cope with faulty/non-functioning/malicious-Byzantine participating robots. We suggest a framework in which Byzantine participants are surrounded/captured/arrested by several of the non-Byzantine robots while the rest are free to achieve the swarm task without the intervention of the Byzantine robots.

The recovery of Byzantine robots and the refresh of prejudice indications are integrated by repeated periodical synchronous refreshes. The chosen periods are tuned to ensure liveness towards achieving the swarm task. Typically, the periodic refreshes will not influence the task progress when no mall-functioning/Byzantine robot participates.

We demonstrate our forgive and forget framework, presenting algorithms for two specific tasks. We believe that the forgive and forget framework opens new practical direction for coping with Byzantine participants in swarms. The interest is not limited to the case of a single Byzantine participant, nor to the specific games presented.

References

1. Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.).* Wiley series on parallel and distributed computing. Wiley, 2004.
2. Zohir Bouzid, Shlomi Dolev, Maria Potop-Butucaru, and Sébastien Tixeuil. Robocast: Asynchronous communication in robot networks. In *OPODIS*, pages 16–31, 2010.
3. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
4. Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin Parvédy. Fault-tolerant and self-stabilizing mobile robots gathering. In *DISC*, pages 46–60, 2006.
5. Yoann Dieudonné, Shlomi Dolev, Franck Petit, and Michael Segal. Explicit communication among stigmergic robots. *Int. J. Found. Comput. Sci.*, 30(2):315–332, 2019.
6. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
7. Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000.
8. Shlomi Dolev, Sayaka Kamei, Yoshiaki Katayama, Fukuhito Ooshita, and Koichi Wada. Brief announcement: Neighborhood mutual remainder and its self-stabilizing implementation of look-compute-move robots. In *DISC*, 2019.
9. Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.
10. Samia Souissi, Taisuke Izumi, and Koichi Wada. Byzantine-tolerant circle formation by oblivious mobile robots. In *CCCA*, pages 1–6, 2011.
11. Volker Strobel, Eduardo Castelló Ferrer, and Marco Dorigo. Managing byzantine robots via blockchain technology in a swarm robotics collective decision making scenario. In *AAMAS*, pages 541–549, 2018.