The Journal of Systems and Software 84 (2011) 29–36

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Fault-tolerant flocking for a group of autonomous mobile robots

Yan Yang[a,*], Samia Souissi[b], Xavier Défago[c], Makoto Takizawa[d]

[a] Western Illinois University, USA
[b] Nagoya Institute of Technology, Japan
[c] Japan Advanced Institute of Science and Technology, Japan
[d] Seikei University, Japan

ARTICLE INFO

Article history:
Received 15 September 2009
Received in revised form 12 June 2010
Accepted 13 August 2010
Available online 26 August 2010

Keywords:
Flocking
Fault tolerance
Distributed algorithms
Mobile robots
k-Bounded scheduler

ABSTRACT

Consider a system composed of mobile robots that move on the plane, each of which independently executing its own instance of an algorithm. Given a desired geometric pattern, the flocking problem consists in ensuring that the robots form this pattern and maintain it while moving together on the plane. In this paper, we explore flocking in the presence of faulty robots, where the desired pattern is a regular polygon. We propose a distributed fault tolerant flocking algorithm assuming a semi-synchronous model with a *k*-bounded scheduler, in the sense that no robot is activated no more than *k* times between any two consecutive activations of any other robot.

The algorithm is composed of three parts: failure detector, ranking assignment, and flocking algorithm. The role of the rank assignment is to provide a persistent and unique ranking for the robots. The failure detector identifies the set of currently correct robots in the system. Finally, the flocking algorithm handles the movement and reconfiguration of the flock, while maintaining the desired shape. The difficulty of the problem comes from the combination of the three parts, together with the necessity to prevent collisions and allow the rotation of the flock. We formally prove the correctness of our proposed solution.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Control and coordination of a set of autonomous robots has been grown very quickly because such cooperation of simple robots can offer several advantages, such as redundancy and flexibility, and allows performing hard tasks that could be impossible for one single robot. One of the important activities of multiple robots is flocking. In short, flocking is the ability for a group of robots to move in formation and to preserve this formation while moving. Moving in formation has many important applications, such as, transporting large objects, exploring hazardous areas, and surveillance.

Until now, the flocking problem has been studied from various perspectives, such as control theory, artificial intelligence, and computational viewpoint. Shi et al. (2009) consider the flocking problem of a group of autonomous agents moving in Euclidean space with a virtual leader, from a control theory viewpoint. A set of control laws is introduced to enable the agent group to generate the desired stable flocking motion. Yousuf and Zaman (2009) surveyed fault tolerance techniques used in mobile agents and mobile agent system.

A self-organized flocking method is proposed by Turgut et al. (2008) for a swarm of mobile robots using alignment and proximal control. They use a mobile robot platform developed specially for swarm robotic studies, briefly describing its sending and communication abilities. The approach is however not fault-tolerant as it is implicitly assumed that no robots crash during flocking.

To the best of our knowledge, only few works studied the flocking problem from computational viewpoint, such as Souissi et al. (2008), Gervasi and Prencipe (2004) and, Canepa and Gradinariu Potop-Butucaru (2007). Among them, the work of Gervasi and Prencipe (2004) and Canepa and Gradinariu Potop-Butucaru (2007) are closest to our work. However, they do not address fault-tolerance in that they both rely on the assumption that robots always function properly, and thus never crash. This is impractical for robots operating in dangerous or complex environments, as robot failures are quite likely to occur.

Perhaps surprisingly, fault tolerance of multiple robot systems has been explored to very little extent so far. Yoshida et al. (1997) proposed a fault-tolerant algorithm to select the active mobile robots from a group of mobile robots. Unfortunately, the authors only considered *initial* crash faults of robots, i.e., a faulty robot that makes no motion from the beginning of execution of the algorithm. Clearly, it is too restrictive for dynamic applications like flocking, since the probability of robot failure during execution is significant, due to external influence from the environment.

In some of our recent work (Souissi et al., 2008), we have proposed an algorithm that solves the flocking problem with crash faults in an asynchronous system. However the limitation of that

* Corresponding author.
E-mail address: yanyang818@gmail.com (Y. Yang).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.08.026

algorithm is that it does not allow the rotation of the formation by the robots. The question remains open whether we can lift such limitation.

In this paper, we propose an algorithm that solves the flocking problem in general (also in a crash failure model), by allowing the formation to move to any direction, as well as rotate, yet in a more restrictive model (semi-synchronous). In this paper, we validate the approach by developing the proof of correctness. The proof shows that our flocking algorithm can avoid the collisions between robots, while allowing them to flock together with a desired formation even if some of the robots crash.

The rest of the paper is organized as follows: In Section 2, we describe the robot network model, the system model, and give some important definitions about flocking. Section 3 describes the three modules for fault tolerant flocking: namely, flocking, failure detection, and rank assignment. Then, the correctness of the proposed modules is proved in Section 4. Finally, we discuss and conclude the paper in Section 5.

2. Robot network model, system model and definitions

2.1. System model

We consider a distributed system consisting of a team of autonomous mobile robots in the semi-synchronous model of Suzuki and Yamashita (Ando et al., 1999) (referred to as SYm). In this model, each robot is able to sense its surroundings, perform computations on the sensed data, and move toward the computed destination. The behavior of a robot constitutes its activation cycle of *looking*, *computing*, *moving* and being idle *wait*. If one such cycle of a robot is executed, we call it the *activation* of a robot.

In the SYm model, robots are autonomous, in the sense that they cannot be distinguished by their appearance, and they do not have any kind of identifiers that can be used during the computation. Also, they cannot communicate with each other explicitly, the only way to communicate being by vision. The local view of each robot includes a unit of length, an origin and the directions and orientations of the two x and y coordinate axes. In particular, we assume that robots have a partial agreement on the local coordinate system. Specifically, they agree on the orientation and direction of one axis, say the y-axis. Also, they agree on the orientation of clockwise/counterclockwise. Unlike similar work, the robots are not oblivious, in that they have a memory allowing them to remember past activations. Further, in our model, a robot can see all of the other robots in the environment. If it was not the case, robots could easily end up partitioned.

We further make the following assumption on the system model. To guarantee the fairness of activation among robots, each robot is activated based on k-bounded scheduler, which is defined as follows.

2.1.1. k-Bounded-scheduler

With a *k-bounded scheduler*, between two consecutive activations of a robot, no other robot is activated more than k times, where k is a constant natural number.

2.1.2. Fault model

In this paper, we address *crash failures*. That means, a robot may fail by crashing, after which it executes no actions (no movement). A crash is *permanent* in the sense that a faulty robot never recovers. However, it is still physically present in the system, and it is seen by the other non-faulty robots. A robot that is not faulty is called a *correct* robot.

2.2. Problem definition

Informally, flocking is the formation and maintenance of a desired pattern[1] while moving, by a team of mobile robots. A group of robots can form any kind of patterns. Without loss of generality, in this paper, we consider the formation as a regular polygon. Also, during the movement of a group of mobile robots, a leader–followers approach (Gervasi and Prencipe, 2004) is adopted. It means that, at any time, there exists a robot leader in the system to lead the other robots, called followers. This leader is elected and known by the other robots in the system. In other words, the followers just need to follow the leader wherever it goes, while keeping the given formation.

In order to provide a formal definition of the flocking problem, we use the following definitions introduced in Gervasi and Prencipe (2004).

Definition 1. (*D-distance*). Given two position snapshots of a group of robots $C=\{c_1, c_2, .., c_n\}$ and $G=\{g_1, g_2, ..., g_n\}$, where c_i and g_i denote the positions of a robot in snapshot C and g respectively, the D-distance between them is defined as follows:

$$D(C, G) = \min_{\pi \in \Pi} \sum_{i=1}^{|C|} dist(c_i, g_{\pi(i)})$$

where Π is the set of all possible permutations of $1, \ldots, |C|$ and $dist(c_i, g_{\pi(i)})$ is the distance between two points c_i and $g_{\pi(i)}$.

Definition 2. (*Target*). Given a pattern P and a position configuration E, we call an undirected target of the robots any formation that is obtained by translating P so that its leader point coincides with the leader of E, and rotating it by an arbitrary angle. We denote such a formation by $T_{P,E}$.

Definition 3. (*Approximate undirected formation*). An undirected target is formed up to ξ if $D(E, T_{P,E}) \leq \xi$.

Definition 4. (*Flocking*) Let $r_1, r_2, \ldots, r_n$ be a group of robots, whose positions constitute a formation E, and let P be a pattern given in input to $r_1, r_2, \ldots, r_n$. The robots solve the approximate formation problem during flocking if, starting from a given desired formation, the robots maintain a target formation up to ξ. That is, during flocking, there exists a formation that satisfies $D(E, T_{P,E}) \leq \xi$.

Definition 5. (*Fault tolerant flocking*) The robots in the system can satisfy the following two conditions: (1) the definition of flocking (see Definition 4); (2) when some robots are crashed, the remaining correct robots still can re-form an approximation undirected formation (see Definition 3) and then flock together.

3. Fault tolerant flocking

In this section, we give a decentralized fault tolerant flocking algorithm for robots. It ensures that all correct robots form an approximate regular polygon and maintain it while moving. During the execution of flocking, two modules called failure detector and rank assignment are used as black boxes. The rank assignment module is to assign each robot a unique rank to help robot select a unique leader robot during flocking; the failure detector module it to select the correct robots for flocking algorithm. The more detail of these two black boxes will be described in the following two subsections.

[1] The pattern means a geometrical graph in 2-dimension plane or three dimension space.

3.1. Flocking algorithm

Initially, all robots are located on a regular polygon. The goal (requirements) of our algorithm is to maintain an approximation of a regular polygon in the absence of crash, and to reform a new regular polygon with the correct robots after a crash occurs. The interesting point of our algorithm is that it can allow the formation to rotate freely.

The main idea of the algorithm is as follows: First, each robot is assigned a unique persistent rank based on its position by rank assignment module. Then, a robot identifies the correct robots by failure detector module. Third, based on the rank of each robot, a unique leader is selected from the set of correct robots. Finally, based on the positions of the leader and the other correct robots, a robot computes a target position and moves. The algorithm ensures that the combined movements of the robots satisfies the flocking requirements.

The specific algorithm is shown in Algorithm 3. Before we explain the details of the algorithm, some variables are introduced as follows:

- r_i: the robot with rank number i;
- $S_{CurPosObs}$: the set of positions of robots in the current activation;
- $S_{PrePosObser}$: the set of observed positions of robots in the previous activation;
- *HisRankPos*: the rank and the set of all robot's positions during a robot's past k activations. The data structure can be <rank, latest k positions>;
- N_{oct}: the number of activations of a robot;
- d: the expected edge length of a regular polygon, assumed $d \geq \xi$;
- $S_{correct}$: the set of positions of the correct robots.

This algorithm is decentralized and is executed by every robot r_i. The input is d, N_{act}, $S_{CurPosObs}$, $S_{PrePosObser}$, *HisRankPos*. First, by using the persist rank module, r_i gets the ranks of all robots. Then, the set of correct robots $S_{correct}$ is given by the failure detector module. The *leader* is chosen as the robot with the smallest rank in $S_{correct}$. If the current robot is *leader*, the procedure Leader_Move is executed; otherwise Follower_Move is executed. In detail, these two procedures work as follows:

For **Leader_Move**: A global variable m is used for the *leader* to control its moving speed per activation to wait for "lazy" follower robots.[2] The *leader* first observes the current robots' positions. If the positions satisfies Definition 3, then the *leader* moves by no more than $\min(d/2k(k+1), \xi/nk)$ to any position p, where $p \notin HisRankPos$ [3]; otherwise, the leader will slow down by using the global variable m, and move by no more than $\min(d/2mk(k+1), \xi/mnk)$ to any position p, where $p \notin HisRankPos$.

For **Follower_Move**: First r_i computes its target position $\text{Target}(r_i)$ based on the function Formation and the corresponding codes in Algorithm 5. Then, it moves to a desired position p, which is not far away $\min(d/2k(k+1), \xi/nk)$ from the current position and $p \notin HisRankPos$.

A question may arise that: why should the speed of a robot's movement be no more than $\min(d/2k(k+1), \xi/nk)$? This is because: during a robot's k activations, the movement distance cannot be more than $\min(d/2k(k+1), \xi/nk) \times k = \min(d/2(k+1), \xi/n)$. Thus, it can satisfy the requirement of Definition 3 and also can avoid collisions between robots.

[2] The robots activated less often than the other robots are called "lazy" robots.

[3] The reason why the target position is different from the position in *HisRankPos* is to satisfy the need of the failure detector module.

Algorithm 1 Fault tolerant flocking (code executed by robot r_i)

```
Variables m: increasing factor;
Input: S_CurPosObs, N_act, HisRankPos, d, S_PrePosObser

Upon activation {
Call Persist_Rank(S_CurPosObs, N_act, HisRankPos, d);
S_correct := Select_Correct_Robots(S_PrePosObser, S_CurPosObs);
n = |S_correct|;
leader := Robot with the smallest rank in S_correct;
if r_i is leader then
  Leader_Move(d, ξ, HisRankPos, S_correct, m);
else
  Follower_Move(d, ξ, HisRankPos, S_correct, leader);
end if
}
```

Algorithm 2 Leader movement (code executed by the leader robot)

```
procedure Leader_Move(d, ξ, HisRankPos, S_correct, m)
  if S_correct satisfies Definition 3 (see Section 2B) then
    Move by not more than min((d/2k(k+1)), (ξ/nk)) to any position p where
      p ∉ HisRankPos;
    m = 1;
  else
    m = m + 1;
    Move by not more than min((d/2mk(k+1)), (ξ/mnk)) to any position p, where
      p ∉ HisRankPos;
  end if
end
```

Algorithm 3 Follower movement (code executed by a follower robot r_i)

```
procedure Follower_Move (d, ξ, HisRankPos, S_correct, leader)
  Variables: S_TargetPos: the set of target positions; Target(r_i): the target
      position of r_i;
  Function: Formation (d, S_correct, leader, HisRankPos) := the function to
      compute the target robot positions;
  S_TargetPos = Formation (d, S_correct, leader, HisRankPos);
  Sort the position in S_TargetPos starting from the leader's by increasing
      order of the y coordinate, and clockwise direction;
  Sort robots in S_correct based on the rank from small to large;
  Assign the jth position of S_TargetPos to the jth robot of S_correct
      (1 ≤ j ≤ |S_correct|);
  if dist(r_i, Target(r_i)) > min((d/2k(k+1)), (ξ/nk)) then
    Find position p on Segment r_i Target(r_i), which satisfies
      dist(r_i, p) > min((d/2k(k+1)), (ξ/nk)) (see Fig. 1);
    Target(r_i) := p;
  end if
  if Target(r_i) ∈ HisRankPos then
    Find a position p which satisfies dist(p, Target(r_i)) < ξ/nk and
      p ∉ HisRankPos;
    Target(r_i) := p;(see Fig. 2)
  end if
  Move to Target(r_i);
end
```

3.2. Persistent ranking for robots

In this part, we provide a simple algorithm that gives a unique identification to each robot in the system. Among other things, this allows to agree on a unique leader. Our main idea is as follows: First, *assign* ranks to robots based on their initial locations during its first k activations; Then, each robot *keeps* its rank during flocking. The specific rank assignment algorithm is given in Algorithm 8, where the procedure Assign_Rank is to assign ranks to robots during a robot's first k activations and Persist_Rank is to keep their rank persistently after their first k activations during flocking. These two procedures work as follows:

- **Assign_Rank** Once a robot is activated, it will get the set of positions of all robots $S_{CurPosObs}$ by sensor. Then, it sorts the positions in $S_{CurPosObs}$ in decreasing order of y-coordinates, and puts the ordered positions into a variable named *RankSequence*; then it sorts the positions of the robots with the same y-coordinate

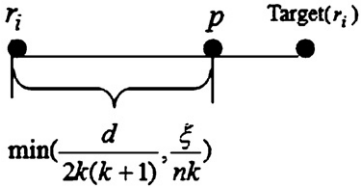

Fig. 1. Finding a new target position p, where r_i is a robot's current position, Target(r_i) is the target position.

from right to left in clockwise direction (the robot located on the right has a rank smaller than the one on the left); Finally, the robot saves *RankSequence* and their corresponding ranks into *HisRankPos*.

- **Persist_Rank** First, by calling the failure detector module, the set of correct robots $S_{correct}$ can be obtained. Referring to *HisRankPos* and $S_{correct}$, a robot excludes the positions of crashed robots from the current position snapshot $S_{CurPosObs}$. Then, for every position p in $S_{CurPosObs}$, it can find its last recent position in *HisRankPos* that is not far away from p by $\min(d/2(k+1), \xi/n)$. This bound on distance is conservative, and it ensures that no collision can possibly occur between robots. Thus, p's rank can be found by corresponding to p's last past position in *HisRankPos*.

Specifically, during the first k activations, a robot moves to its right, perpendicularly to its y axis, by ξ/nk. The moving speed is chosen as ξ/nk per activation, because during this period, each robot could make sure the positions of robots still maintain an approximate polygon after moving. Thus, during the first k activations, the relative positions of robots are not changed and finally all robots get the same initial position configuration. That means, for the same robot, that its rank remains the same in the memory of different robots (Figs. 1 and 2).

Figs. 3 and 4 illustrate how a robot assigns, and tracks the rank of robots in the system, respectively. In Fig. 3, there are four robots in the system, which are initially located at position *A*, *B*, *C* and *D*, respectively. Among these positions, Position *B* and *D* have the same *y* coordinate. Based on the procedure Assign_Rank (Algorithm 8), the ranks corresponding to the positions of the four robots are: (1, *A*), (2, *B*), (3, *D*), and (4,C). This information is recorded in *HistoryRankPos*, i.e., *HistoryRankPos* = {(*rank*, *position*)}={(1, *A*), (2, *B*), (3, *D*), (4, *C*)}. In Fig. 4, all robots keep their ranks. The main idea of keeping robots' rank is to find the robot's last position based on the movement rule and the current position. Consequently, each robot gets its rank based on the information of *HistoryRankPos*. For example, a robot finds its last position *A* which is located in the current position A"s searching disc. Then, by checking *HistoryRankPos*, it knows the information (1,*A*). Therefore, it knows that its rank is 1.

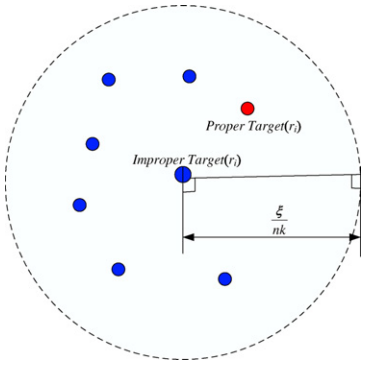

Fig. 2. Finding a proper target position which is not included in the *HisRankPos*, where blue points means the positions in *HisRankPos*, the red point is the computed proper target position. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of the article.)

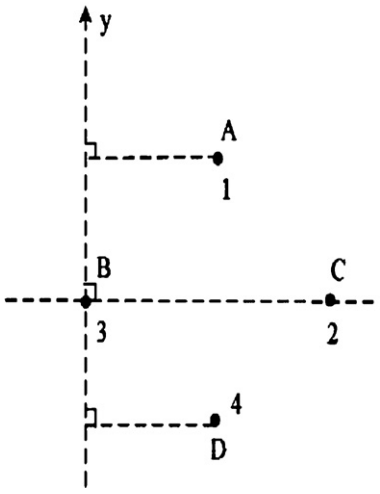

Fig. 3. Assign_Rank during the first k activations: initially, four robots are located at position *A*, *B*, *C* and *D*, where *B* and *C* has the same *y* coordinate. By Algorithm 1, their ranks are 1, 3, 2, and 4 which corresponds to positions *A*, *B*, *C* and *D*, respectively.

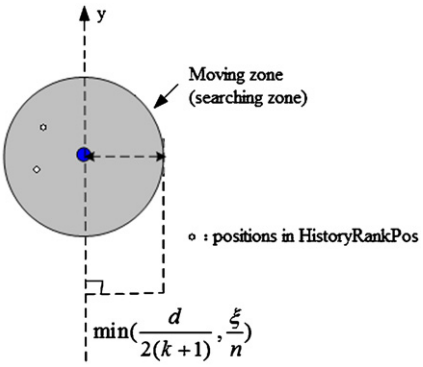

Fig. 4. Robots keep their ranks in their searching zones (moving zone), whose radius is $\min(d/2(k+1), (\xi/n))$, excluding positions in *HistoryRankPos*.

Algorithm 4 Persistent Rank (code executed by robot r_i)
Input: $S_{CurPosObs}$, N_{act}, *HisRankPos*

if ($N_{act} \leq k$) **then**
  First_AssignRank($S_{CurPosObs}$);
  $n = |S_{CurPosObs}|$;
  Move to its right hand by not more than $\frac{\xi}{nk}$ along the perpendicular to its *y*-axis;
else
  Persist_Rank($S_{CurPosObs}$, *HisRankPos*, *d*)
end if

procedure First_AssignRank($S_{CurPosObs}$)
  RankSequence = the positions in $S_{CurPosObs}$ in decreasing order by *y*-coordinate;
  Sort the positions of robots in *RankSequence* with the same *y*-coordinate from left to right by decreasing order
  The rank each robot corresponds to its order in *RankSequence*;
  Save *RankSequence* and the corresponding rank into *HisRankPos*;
end
procedure Persist_Rank($S_{CurPosObs}$, $S_{PrePosObser}$, *d*)
  $S_{correct}$ = Select_Correct_Robot (*HisRankPos*);
  $n = |S_{correct}|$;
  Referring to $S_{correct}$ and *HisRankPos*, exclude the positions of crashed robots from $S_{CurPosObs}$;
  for $\forall p \in S_{CurPosObs}$ **do**
    Find position q which satisfies $q \in S_{PrePosObser}$ and $dist(q, p) \leq \min\left(\frac{d}{2(k+1)}, \frac{\xi}{n}\right)$;
    Get p's rank that corresponds to q's;
    Save p's position into *HisRankPos* and $S_{PrePosObser}$;
  end for
end

3.3. Failure detector

Failure detection is a good method to detect the failure of an agent (robot) or a process. Recently, we proposed many different failure detection methods. In this section, we present a failure detector algorithm called *SelectCorrectRobot* that provides a way to select the correct robots. The main idea of *SelectCorrectRobot* is: based on the movement rule of a robot in flocking—each time a robot moves to a position that is different from before, so a robot can detect when a robot has crashed if that robot has not changed position for too many activations.

Algorithm 5 Selection of Correct Robots

```
procedure Select_Correct_Robot(S_CurPosObs, S_PrePosObser, HisRankPos)
  S_temp := S_PrePosObser;
  S_correct := φ;
  for ∀pi ∈ S_CurPosObs do
    if p_i ∈ S_PrePosObser then
Counter_i = Counter_i + 1;
    else
      Counter_i = 0;
    end if
    if Counter_i > k then
      S_temp = S_temp − {p_i};
    end if
  end for
  for ∀p_i ∈ S_temp do
      S_correct = S_correct ∪ {r_i | r_i is the rank of p_i};
  end for
  return (S_correct);
end
```

Each robot uses the same algorithm to select the non-faulty robots (Algorithm 3.3). The variables used in Algorithm 3.3 are described as follows:

$Counter_i$: a variable used for recording the times that robot r_i did not change its current position;

The input of this algorithm is $S_{CurPosObs}$, $S_{\mathrm{Pre}PosObser}$, *HisRankPos*. First, robot r_i compares the current position configuration $S_{CurPosObs}$ to the previous one recorded in $S_{\mathrm{Pre}PosObser}$. If the two position match, then $Counter_i$ is incremented; otherwise, $Counter_i$ reset to zero. When $Counter_i$ is larger than k, i.e., the monitored robot has not moved for more than k-activations of r_i, this means that it must have crashed (based on k bounded scheduler and the movement rule in flocking). Finally, by corresponding the position and rank in *HisRankPos*, robot r_i will know the rank of the crashed robot. In this way, robot r_i can get the set of all correct robots $S_{correct}$.

4. Correctness

In this section, the proof will be used as a validation method to evaluate and validate the correctness of our proposal. During the execution of fault tolerant flocking algorithm (Algorithm 3), a robot first calls procedure Persist_Rank to assign a rank to each robot, then the failure detector (Algorithm 3.3) is used to select the set of correct robots; after that, the robots move based on Leader_Move (Algorithm 4) or Follower_Move(Algorithm 5). Obviously, based on the logic of fault tolerant flocking algorithm, we prove the correctness of all the algorithms in the sequence: Rank assignment (Algorithm 8), failure detector (Algorithm 3.3), collision avoidance and fault tolerant flocking (Algorithm 3).

4.1. Rank assignment

In the following, we will prove the correctness of Algorithm 8.

Lemma 1. By Algorithm 8, all correct robots agree on the same sequence of ranking, *RankSequence* during the first k activations.

Proof. Initially, all robots are located on a regular polygon whose edge length is equal to d. During the first k activations of a robot, a robot moves to the right by no more than ξ/nk, perpendicular to the y-axis. This ensures that the y-values of the robots do not change during this period. Considering an extreme case in which the robots on the right activate only once during a period in which left robots activate k times. Since the maximum distance that the left robots can move is $(\xi/nk) \times k = \xi/n < d$, robots do not swap positions. The same arguments applies to any two robots in the system. Therefore, after the activation of all robots in the system, they compute the same *RankSequence* and agree on the same ranks for every robots.□

A direct consequence from Lemma 1 can be deduced:

Lemma 2. Algorithm 8 gives a unique rank to every robot in the system during the first k activations.

Lemma 3. By Algorithm 3 and Algorithm 8, the ranking is persistent during the entire execution of the algorithm.

Proof. Based on the rank persistent function, i.e., Function Persist_Rank, each robot matches its last position in the desired disc, which radius is $\min(d/2(k+1), \xi/n)$ at the center of its current position. After that, in each robot's memory, all robots find their last position and then keep their rank.

Therefore, the question is how to make sure a robot finds all robots matching last positions for any number of activations. The worst case occurs when some robots activate k times while a robot activates only once. Otherwise, in any other case, a robot activates less and then the movement distance is less than that during k activations. Therefore, in the following we only need to prove if in the worst case all robots could find their matching last positions. If yes, then in any cases (in any number of activations) using Persist_Rank algorithm, all robots can find their matching last positions, and then keep their ranks. From Lemma 2, we know that: after the first k activations, each robot gets a unique rank.

By Algorithm 3, at each activation the leader moves by no more than $\min(d/2k(k+1), \xi/nk)$. Therefore, during its k activations, the maximum distance that the leader can move is at most $\min(d/2km(k+1), \xi/nmk) \times k = \min(d/2m(k+1), \xi/nm) \leq \min(d/2(k+1), \xi/n)$ since $m \geq 1$.

The same holds with the followers; at each activation a follower moves by $\min(d_l, \min(d/2k(k+1), \xi/nk)) \leq \min(d/2k(k+1), \xi/nk)$. Therefore, during k activations, the maximum distance that a follower can travel is also bounded by $\min(d/2(k+1), \xi/n)$.

Since $k \geq 1$, $\min(d/2(k+1), \xi/n) \leq d/2(k+1) < d/2$ By excluding the crashed robots, each robot can track the other robots' past positions and keep their rank persistent during the flocking.

In the procedure Persist_Rank, a robot first finds the robots which, in the last activation, are within a distance of $\min(d/2(k+1), \xi/n)$ from the current position. Call that area D_r (see line 23 in Rank assignment module). In the set of robots that can satisfy the above condition, there are two categories: crashed robots and one single alive robot (if the current robot does not crash). That is because: (1) initially the distance between any two robots is equal to d, and (2) when no robot have crashed, every follower tries to move to its target position to form a desired regular polygon, i.e., tries to keep a distance d with its neighbors. Between any two consecutive activations of a robot, the other robots cannot be activated more than k times. Hence, the maximal distance that a robot can move is no more than $\min(d/2(k+1), \xi/n)$. Therefore, for any two correct robots, it is impossible to be closer than $\min(d/2(k+1), \xi/n) \leq \min(d/2, \xi/n) \leq d/2$ since $n > 3$, $k > 2$, $\xi \leq d$, unless a robots has crashed.□

From Lemma 1 to Lemma 3, we derive the following theorem.

Theorem 1. Algorithm 1 gives a unique persistent rank for every robot in the system.

4.2. Failure detector

The proposed non-faulty robot selection algorithm (Algorithm 3.3) satisfies the following properties of perfect failure detector (Chandra and Toueg, 1996): strong completeness, strong accuracy, and eventual agreement. Due to space limitation, we do not give the detailed correctness proof.

These properties are proved respectively, in Theorem 2, Theorem 3, and Theorem 4.

Theorem 2. Strong completeness: eventually every robot that crashes is permanently suspected by every correct robot.

Theorem 3. Strong accuracy: No robot is suspected before it crashes.

From Theorems 2 and 3, we can conclude that Algorithm 3.3 satisfies the property of perfect failure detector (Chandra and Toueg, 1996) and also has the following property:

Theorem 4. (Eventual agreement) Eventually all correct robots will have the same set of correct robots.

4.3. Collision between robots

Lemma 4. By Algorithm 8, there is no collision between any two robots in the system during their first k activations.

Proof. From the assumption, we know that the initial distance between any two robots is equal to d. During the first k activations, all robots just move to the right along the y-axis, by less than ξ/nk in each activation. Even if one robot is very active, the furthest distance that it can travel is no more than $(\xi/nk) \times k = (\xi/n) < d$. Therefore, there is no collision between any two robots during the first k activations by Algorithm 1, and the lemma holds.□

Lemma 5. There is no collision between robots after the first k activations by Algorithm 3.

Proof. We prove this lemma in the following two cases.

Case 1: no new robots crashed during flocking.

In this case, we prove the lemma in two steps: no collision between the leader and the followers, and no collision between followers.

Step 1 (no collision between the leader and the followers): Initially the input of the flocking algorithm is a regular polygon whose edge length is equal to d. If a follower and the leader are correct, the follower will follow the leader, trying to keep an approximate regular polygon, that is, a follower tries to keep distance d with its neighbors. Furthermore, during k activations, their movement distance is less than $\min(d_l,\ \min(d/2k(k+1),\ \xi/nk) \times k \leq \min(d/2k(k+1), \xi/nk) \times k = \min(d/2(k+1), \xi/n) < d/2$. Therefore, it is impossible for the leader and the followers to collide.

Step 2 (no collision among followers): In the initial position configuration, the distance between neighbor followers is equal to d. Based on the way the target position is computed (see code in lines 4–7 in procedure Follower_Move), each follower moves to its matched target position, and no two followers' movement paths cross each other. Therefore, there is no collision between any pair of correct followers.

Case 2: some robots crashed during flocking.

In this case, we will prove that there is no collision between crashed robots and correct robots, and no collision occurs between correct robots.

For correct robots, the proof is the same as in Case 1. When there are robots crashed (leader or follower), the crashed robots can no longer move, and their positions are saved in *HisRankPos*. By the algorithm, we know that the target positions of correct robots are different from the positions in *HisRankPos*. Obviously, their target positions are different from those of the crashed robots. Therefore, it is impossible for a correct robot to collide with a crashed one.□

4.4. Fault tolerant flocking

The following proof is for Algorithm 3.

Lemma 6. Algorithm 3 allows the correct robots to correctly keep an approximation of a regular polygon when no new robots crash during flocking.

Proof. To prove the lemma, we use mathematical induction.

(1) Initially, a regular polygon with n robots is the input of the flocking algorithm. When an activation number $k' \leq k$, every robot just adjusts its movement by moving to its right, perpendicularly to its y-axis, by no more than ξ/nk (line 5, Algorithm 8). It is not hard to see that during this period, $D(E, T_{P,E}) \leq \xi$.
(2) Assume that: before an activation $k'(k' \geq k)$ of a robot, an approximate regular polygon is preserved. For any robot $r_i (0 < i \leq n)$, let $P_i^{k'}$ denote the current position of r_i and $T_i^{k'}$ denote the target position of r_i at activation k'. Then, $D(E, T_{P,E}) = \sum_{i=1}^{n} dist(P_i^{k'} - T_i^{k'}) \leq \xi$.

During the next k activations of a robot, if the leader moves forward by d', where $d' < \min(d/2(k+1), \xi/n)$, then the target positions of the followers also change.

For a robot r_i, we assume it is activated x_i times during this period. Then, it will move toward its target $T_i^{k'+x_i}$, which is at a distance of d_l from $T_i^{k'}$. That is, $dist(T_i^{k'} - T_i^{k'+x_i}) = d_l$.

If r_i is the leader robot, $dist(T_i^{k'} - T_i^{k'+x_i}) = 0$.

If r_i is a follower, then it will move toward its target $T_i^{k'+x_i}$. Based on the code at lines 12–15 in the Follower Movement algorithm, after r_i moves, the distance between its actual position and its target position is no more than $(\xi/nk) \times k = \xi/n$. That is, $dist(P_i^{k+k'} - T_i^{k'+x_i}) \leq \frac{\xi}{n}$. Thus, we get $D(E, T_{P,E}) = \sum_{i=1}^{n} dist(P_i^{k+k'} - T_i^{k'+x_i}) \leq 0 + \sum_{i=1}^{n-1} \xi/n = \xi(n-1)/n < \xi$. Here, because k' can be any natural number, we can conclude that all robots can keep an approximate regular polygon when no robots crash during flocking.□

From Lemma 6, we derive the following lemma.

Lemma 7. The flocking algorithm allows robots to make the formation move to any direction including rotation of the formation.

Lemma 8. When some robot crash during flocking, the remaining correct robots can reconfigure into a regular polygon dynamically within finite time.

Proof. If the leader has crashed, then based on the code at line 7 of the flocking module, a new leader is elected dynamically. Once a follower finds that the leader has crashed, it will follow the new leader. At the same time, if the leader finds that the current correct robots cannot form an approximate regular polygon, it will slow down according to the code at lines 6–7 in the leader movement algorithm. For the correct followers, they will compute their target positions based on the code at line 4–7. These followers will approach their target positions gradually in finite time since the followers have higher speed than the leader due to $m > 1$. Therefore a new approximate regular polygon can be reformed using this method.□

5. Maneuverability and bound analysis

Based on the algorithm description and the correctness proof, we know the proposed algorithm lifts the limitation of formation rotation of the paper (Souissi et al., 2008), albeit in a more restrictive

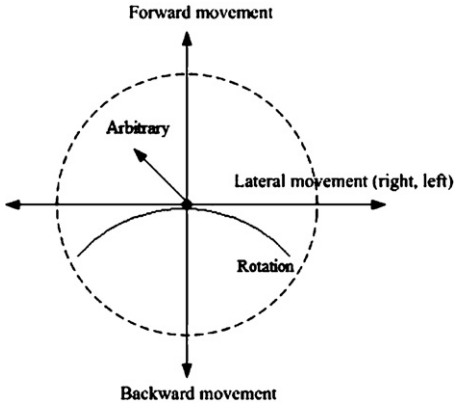

Fig. 5. The possible leader movements: rotation, move forward, move backward, lateral movement (left, right) and random move.

model (semi-synchronous). To show the maneuverability of Algorithm 3 straightly, in this part we analyze the movement of the robots (formation) in details. Also, we further explore the bound of robots movement to keep the formation, i.e., linear speed per activation and angular speed per activation.

5.1. Maneuverability

From Algorithm 3, we know that, once the leader moves, the follower robots will follow the leader's movement. Thus, finally the whole formation formed by all robots will change. In the following, we will analyze how the leader's movement affects the whole formation movement.

The set of all possible leader movement is shown in Fig. 5: rotation, move forward, move backward, lateral movement (left, right), and random move.

In the following, we give how formation changes due to the leader's movement. The formation changes as follows: rotation of the formation (see Fig. 6), formation move forward (see Fig. 7), move in lateral direction and move in arbitrary direction. From Fig. 6, we see that when the leader moves clockwise around a circle, the followers compute their target position based on procedure Follower_Move. All followers rotate in the same way as the leader. At the same time, they can form an approximate regular polygon. For moving backward and forward, the two cases are similar, so it is not necessary to give the formation backward movement. Same with lateral case, here we only show that the formation changes when the leader moves to the right. There is another interesting movement—arbitrary move in the moving zone (searching zone). The arbitrary movement actually is a combination of linear move-

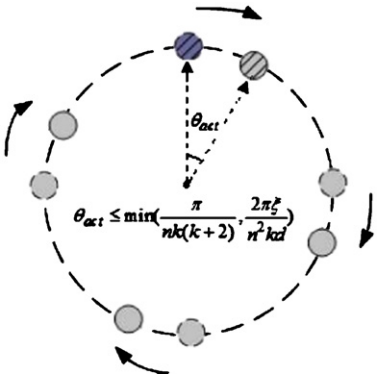

Fig. 6. The formation rotates clockwise: circles with dashed lines are the past positions of robots, and circles with solid line are the current (target) positions..

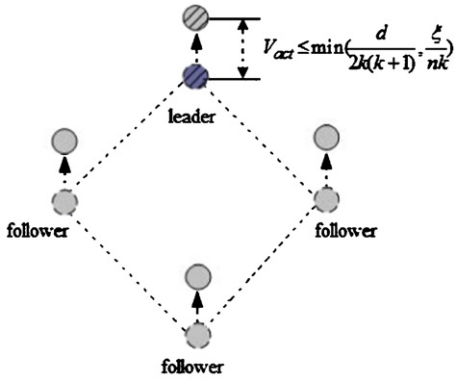

Fig. 7. The formation moves forward: circles with dashed line are the past positions of robots, and the circles with solid line are the current (target) positions.

ment (formation move forward (backward) and lateral move) and rotational movement.

5.2. Bound analysis

In the following, we analyze the formation rotation of robots from the speed per activation and angular velocity per activation. From Algorithm 3, we can get the following properties: the maximum speed per activation of formation rotation is $\min(d/2k(k+1), \xi/nk)$ and the maximum angular velocity per activation is $\min(\pi/nk(k+1), 2\pi\xi/n^2kd)$, where n is the number of correct robots, k is the constant given by the k-bounded scheduler, d is the desired distance between neighbor robots, and ξ comes from Definition 3. That is because, when the distance per activation is equal to d, the angle that passes is $2\pi/n$ based on the definition of a regular polygon). When the distance that a robot passes per activation is $\min(d/2k(k+1), \xi/nk)$, so we can get the angular velocity per activation by using $((\min(d/2k(k+1), \xi/nk) \times 2\pi/n)/d) = \min(\pi/nk(k+1), 2\pi\xi/n^2kd)$. Thus, we get the following theorem:

Theorem 5. When the robots flock, they satisfy the following conditions:

- The linear speed per activation $V_{\text{act}} \le \min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$;
- The angular velocity per activation $\theta_{\text{act}} \le \min(\frac{\pi}{nk(k+1)}, \frac{2\pi\xi}{n^2kd})$.

From the analysis and the figures, we know the proposed flocking algorithm has good maneuverability. The limitation on formation rotation that exists in (Souissi et al., 2008) is lifted by using memory to keep track of robots' ranks instead of keeping the relative positions among robots.

6. Conclusion

In this paper, we proposed a decentralized fault-tolerant flocking algorithm for a group of identically-programmed mobile robots based on k-bounded scheduler in the semi-synchronous model. Our algorithm could make robots avoid collision during flocking. More importantly, the unique characteristic of our flocking algorithm is as follows compared with the existed methods:

- *Distributed*: Each robot executes the exactly same flocking algorithm, while they can coordinate together. Thus, it has good scalability.
- *Fault tolerance*: the proposed algorithm could tolerate the crash of robots, including initial crash and crash during flocking.

- *Free formation rotation*: our algorithm could make the formation of robots rotate freely, unlike restrictions in our previous work (Souissi et al., 2008).

In future work, we plan to evaluate the algorithm quantitatively, using other methods, such as simulation or experimental analysis, in order to further assess the effectiveness of our protocol. When using real robots in a real-world environment, (dynamic or static) obstacles may also exist. Therefore, we have to consider how to deal with these obstacles and avoid collisions. One approach that we will investigate is to combine with our previous work (Yang et al., 2008) in order to propose a flocking protocol, which is fault tolerant, distributed, and also avoids collision between robots and obstacles.

References

Ando, H., Oasa, Y., Suzuki, I., Yamashita, M., 1999. A distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Transactions on Robotics and Automation 15 (October (5)), 818–828.

Canepa, D., Gradinariu Potop-Butucaru, M., 2007. Stabilizing flocking via leader election in robot networks. In: Proceeding of 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Paris, France, November, pp. 52–66.

Chandra, T.D., Toueg, S., 1996. Unreliable failure detectors for reliable distributed systems. Journal of ACM 43 (2), 225–267.

Gervasi, V., Prencipe, G., 2004. Coordination without communication: the case of the flocking problem. Discrete Applied Mathematics 143 (September (1–3)), 203–223.

Shi, H., Wang, L., Chu, T., 2009. Flocking of multi-agent systems with a dynamic virtual leader. International Journal of Control 82 (January (1)), 43–58.

Souissi, S., Yang, Y., Défago, X., 2008. Fault-tolerant flocking in a *k*-bounded asynchronous system. In: Principles of Distributed Systems, 12th International Conference, OPODIS 2008, LNCS 5401, December, pp. 145–163.

Turgut, A.E., Celikkanat, H., Gokce, F., Sahin, E., 2008. Self-organized flocking with a mobile robot swarm. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, vol. 1, pp. 39–46.

Yang, Y., Xiong, N., Chong, N.Y., Défago, X., 2008. A Decentralized and Adaptive Flocking Algorithm for Autonomous Mobile Robots. In: Proceedings of the 3rd International Conference on Grid and Pervasive Computing—Workshops, pp. 262–268.

Yoshida, D., Masuzawa, T., Fujiwara, H., 1997. Fault-tolerant distributed algorithms for autonomous mobile robots with crash faults. Systems and Computers in Japan 28 (2).

Yousuf, F., Zaman, Z., 2009. A survey of fault tolerance techniques in mobile agents and mobile agent systems. In: Second International Conference on Environmental and Computer Science, pp. 454–458.

Yan Yang is a visiting professor in Western Illinois University, Macomb, USA. She obtained her Ph.D. in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2009. Her research includes communication network, information security and dependable distributed system.

Samia Souissi is currently a post doctoral researcher at Department of Computer Science and Engineering, Nagoya Institute of Technology, Japan. She obtained her B.Sc. in Computer Science applied to management from FSEG, Tunisia in June 2001. In September 2007, she received her Ph.D. in Information Science from the Japan Advanced Institute of Science and Technology (JAIST). In addition, from October 2007 to July 2008, she worked as a researcher at the Japan Advanced Institute of Science and Technology. Also, from September 2008 to August 2010, she was awarded the JSPS (Japan Society for the Promotion of Science) Post-doctoral fellowship at the Nagoya Institute of Technology. Her research interests include distributed algorithms, cooperative autonomous mobile robots, and fault-tolerance.

Xavier Défago is an associate professor at the Japan Advanced Institute of Science and Technology (JAIST) since 2003. He obtained his Ph.D. in Computer Science in 2000 from the Swiss Federal Institute of Technology in Lausanne (EPFL; Switzerland). In addition, from 1995 to 1996, he also worked at the NEC C &C Research Labs in Kawasaki (Japan), and has been a researcher for the PRESTO program "Information and Systems" of the Japan Science and Technology Corporation (JST) from 2002 to 2006. He is a member of the ACM, IEEE, and a regular member of the IFIP working group 10.4 on dependable computing and fault-tolerance. His research interests include distributed algorithms, fault tolerance, group communication in computer networks, and cooperative autonomous mobile robot networks.

Makoto Takizawa is a Professor in the Department of Computer and Information Science, Seikei University. He was a Professor and the Dean of the Graduate School of Science and Engineering, Tokyo Denki University. He was a Visiting Professor at GMD-IPSI, Keele University, and Xidian University. He was on the Board of Governors and a Golden Core member of IEEE CS and is a fellow of IPSJ. He received his DE in Computer Science from Tohoku University. He chaired many international conferences like IEEE ICDCS, ICPADS, and DEXA. He founded IEEE AINA. His research interests include distributed systems and computer networks.