Decentralized Fault-tolerant Flocking Algorithms for a Group of Autonomous Mobile Robots

by

Yan Yang

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Associate Professor Xavier Défago

School of Information Science
Japan Advanced Institute of Science and Technology

March 2009

Abstract

Recently, robotics research has been gained a lot of attentions, due to its wide applications, especially in some places that human beings can not survive, like in a planet or fire. There are a lot of interesting applications of multiple robots, such as satellite exploration and surveillance missions. However, in such system one can not possibly rely on assuming fail-proof software or hardware, especially when such robot systems are expected to operate in hazardous or harsh environment. Therefore, the issue of resilience to failure becomes essential. Perhaps surprisingly, this aspect of multiple robot systems has been explored to very little extent so far.

Flocking, as one of important applications of coordination of multiple robots, has a lot of applications, like moving a box from one place to another place, or explores some unknown places. During flocking, mobile robots group to form a desired pattern and move together while maintaining that formation. One difficulty is how to distinguish the crashed robots from those who are staying in "waiting" state since they all don't move during some period. Another difficulty is to make a group of robots move together to form a geographic graph in distributed way. To address the above questions, our research goal is to achieve the effective coordinated flocking even with the crash of mobile robots. More specifically, we focus on solving distributed geographic agreement problems for groups of mobile robots.

First, we proposed a fault tolerant flocking in asynchronous model. Our algorithm ensures that the crash of faulty robots does not bring the formation to a permanent stop, and that the correct robots are thus eventually allowed to reorganize and continue moving together. Furthermore, the algorithm makes no assumption on the relative speeds at which the robots can move. The algorithm relies on the assumption that robots' activations follow a k-bounded asynchronous scheduler, in the sense that the beginning and end of activations are not synchronized across robots (asynchronous), and that while the slowest robot is activated once, the fastest robot is activated at most k times (k-bounded).

By analyzing the above algorithm carefully, we find one flaw in the algorithm due to the limit of design method. That is the formation formed by robots can not be rotated freely. Therefore, in the following part, we design a new method to lift such limit by allowing the formation to move to any direction, including its rotation, yet in a semi-synchronous model. Further analysis demonstrates that the proposed algorithm can make formation freely rotation, and has very good maneuverability.

In practical applications, some parts of a robot, like sensor, moving actuator, or memory etc., is prone to transient crash due to the influence of complex environment. We discuss the (im) possibility of self-stabilization of flocking algorithms with memory corruption in different system models.

Finally, we propose a non-fault tolerant flocking algorithm in order to compare the performance with the above fault-tolerant ones. The described algorithm can effectively adapt to the environment to avoid the collision among robots and obstacles. Furthermore, one interesting thing is when there is no obstacle in the environment; the robots can keep the desired distance with their neighbors.

In all, to the best of our knowledge, our work is the first to consider the fault tolerant issue during robots dynamic flocking. Also it opens some new interesting topics on robots dynamic coordination, like robots coordination in the model of crash and recovery.

Keywords: Mobile robot, decentralized coordination, local interactions, flocking, formation generation, self-organization

Acknowledgments

During my time at Japan Advanced Institute of Science and Technology (JAIST), I have been lucky and honorable to be a member of Dependable Distributed Systems Group (DDSG), Foundations of Software Laboratory. I appreciated Professor Xavier Défago very much for letting me join in this group in 2006. As my supervisor, Professor Xavier Défago is a great technical researcher and an erudite teacher. I would like express my sincere gratitude to my supervisor for advising me and guiding my research during my study at JAIST. He provided me a lot of encouragement and inspiration for my research. Also, he is an incredible source of knowledge, and provided excellent guidance on technical details and publication writing. I am especially thankful for all the time he provided me in improving this dissertation.

Students in the DDSG made it a great place to work. Both supervisors and college-mates made my Ph.D. experience truly unique. In particular, I am very happy to discuss with Dr. Samia Souissi on robot research. Also, I appreciate for Dr. Rami Yared's friendship. Dr. Naixue Xiong gives me a lot of encouragement and friendship during the past six years in research and life. Furthermore, I would like to thank Daiki Higashihara for his kind help.

Professor Takuya Katayama gave me a lot of encouragement to pursue my own ideas and to manage my own research. Most notably, he provided a family- friendly environment that helped me balance my life with my studies.

I would like to thank Professor Nak Young Chong for his helpful discussions and suggestions, who is my supervisor of my sub-theme (minor project). He is able to provide in-depth analysis of my work, and also give me strong spirit encouragement on how to do research.

I really appreciate my defense committees, Prof. Makoto Takizawa, Prof. Koichi Wada, Prof. Tetsuo Asano, Prof.Tomoji Kishi, and Prof. Yasuo Tan, for their great comments for the improvement of this thesis.

I devotes my sincere thanks and appreciation to all of my colleagues for their potential care.

In particular, I sincerely appreciate the support from my family. I am grateful to my parents, who always give me a lot of encouragement and care in my life. Their care gave me inspiration and happiness whenever times were difficult. Thus, they provided an incredible support net while growing up.

Finally, I would like to thank COE (Strategic Development of Science and Technology) foundation in Japan for supporting this research.

Contents

List of Algorithms

Chapter 1

Introduction

In recent years, robotics research has been gained a lot of attentions due to its wide applications. A powerful robot or a group of robots can solve some tasks that are impossible or hard for human beings, like searching after earthquake or exploring on moon, *et al.* (see Figure 1.1).

Nowadays, robotics design has been focused on making system involving a single complex robot towards the design and use of a large number of robots, which are simple, relatively inexpensive, but capable of collaborating to perform complex tasks. Several reasons motivate this shift, including reduced costs, faster computation, more potential for fault tolerance, the possibility of expendability of the system and the reusability of the robots in different applications.

A large range of applications benefit from this shift, particularly applications in dangerous environments, where human lives would be jeopardized, and applications in remote places, either inaccessible to humans or where communication delays render remote controlled robot missions unfeasible. Examples include planetary rover missions, Mars ground preparation, surveillance and inspection of remote sites, such as tight spaces, deep sea, hazardous and hostile environments, search and rescue missions, such as rescuing people trapped under piles of debris in an earthquake disaster, or searching for victims of a flood, exploration of unknown environments, cooperating autonomous vehicles, etc.

1.1 Multi-robot Cooperation

The coordination of multi-robot teams has a wide variety of applications such as mine exploration, load carrying, surveillance-and-security, and search-and-rescue [SCHR01, KOVA03, BLT02, JWFE97].

For such operations, relying on a group of simple robots for delicate operations has various advantages over considering a single complex robot [CC98]. For instance, (1) it is usually more cost-effective to manufacture and deploy a number of cheap robots rather than a single expensive one, (2) higher number yields better potential for a system resilient to individual robot failures, (3) smaller robots have obviously better mobility in tight and confined spaces, and (4) a group can survey a larger area than an individual robot, even if the latter is equipped with better sensors.

Multi-robot systems provide an exciting solution to many real-world problems. Multiple robots can cooperate to manipulate large objects, survey large areas in a short amount of time, and provide system redundancy. These functionalities make them applicable to a variety of tasks including large-scale construction [LALB02], hazardous waste cleanup [RSGHP02], and planetary exploration [SHPB01].

The class of coordination of multiple autonomous mobile robots discussed so far in the literature includes, among others, the following basic tasks: formation of simple geometric patterns [KSIS96], gathering and convergence [CFPS, MCGP02, ISMY96a, ISMY96b], flocking (namely, following the movement of a predefined leader) [GPRE02](see Figure 1.2), uniform distribution over a specified region [KSIS96], and partitioning into groups [KSIS96].

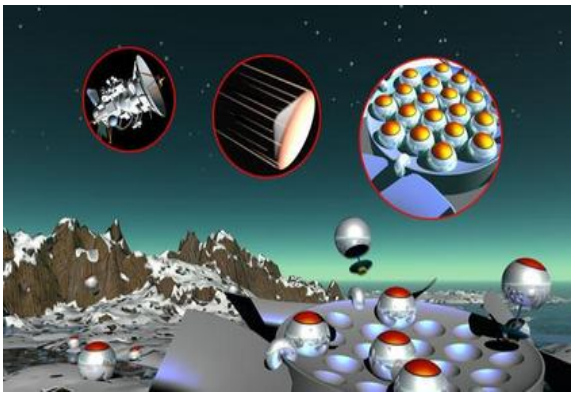


Figure 1.1: A group of robots are searching in the planet or some dangerous places.

Figure 1.2: Example of coordinated social behavior(flocking wild geese).

1.2 Dependability in distributed robot system

In traditional distributed system, increasing the number of components means increasing the probability that some of these components will be subject to failure during the execution of a distributed algorithm. To avoid the necessity of restarting an algorithm each time a failure occurs, algorithms should be designed so as to deal properly with such failures.

Similarly, in mobile robot system, as the common models assume cheap, simple and relatively weak robots, the issue of resilience to failure becomes prominent, since in such systems one cannot possibly rely on assuming fail-proof hardware or software, especially when such robot systems are expected to operate in hazardous or harsh environments [NADP04]. It seems plausible that the inherent redundancy of such systems may be exploited in order to enable them to perform their tasks even in presence of faults.

Recently, there are some research addressed on fault tolerant robot coordination, like in gathering of robots. The first concrete attempt we're aware of for dealing with crash faults is described in [YMF97], where an algorithm is given for the Active Robot Selection Problem (ARSP) in the presence of initial crash faults. The ARSP creates a subgroup of non-faulty robots from a group that includes also initially crashed robots and makes the robots in that subgroup recognize one another. This allows the non-faluty robots in the subgroup to overcome the existence of faults in the system, and they can further execute any algorithm within the group. Then, Agmon and Peleg [AP04] studied fault tolerant algorithms for the problem of gathering n autonomous mobile robots. Subsequently, an algorithm tolerant against

one crash-faulty robot in a system of three or more robots is presented. It is then shown that in an asynchronous environment it is impossible to perform a successful gathering in a 3-robot system with one Byzantine failure. Défago etc. in [DGMP06] significantly extended the studies of deterministic gathering feasibility under different assumptions related to synchrony and faults crash and Byzantine. In addition we extend our study to the feasibility of probabilistic gathering in both fault-free and fault-prone environments.

1.3 Flocking

Flocking is one of the important applications of multiple robots coordination. In detail, flocking is the ability of a group of robots to move in formation and to preserve formation while moving. Flocking has many important applications, for instance, transporting large objects, exploring hazardous areas and surveillance [JSEB00, SHP04, ASER03, RCH98].

At first, the behavior of flocking is observed from the collective motion of a large number of self-propelled entities - is a behavior exhibited by many living beings such as birds, fish, bacteria, and insects [JORE08].

Until now, flocking problem has been studied from various perspectives, such as control theory, artificial intelligence and computational viewpoint [DKG07, GP04, SM03, CP07]. To the best of our knowledge, only few works studied the flocking problem from computational viewpoint, such as [DISC08, GP04, CP07]. Among them, the work of [GP04, CP07] are the closest to our work, however, they rely on the assumption that robots always function properly, and never crash. That assumption is not practical due to the characteristic of weakness of robots, especially if robots operate in dangerous or complex environments.

Perhaps surprisingly, this aspect of multiple robot systems has been explored to very little extent so far. Yoshida *et al*. [YMF97] 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. As we know, it is not applicable in dynamic applications like flocking, because the probability of robot failure during execution is much high due to influence of various environments.

Recently, the problem of flocking has been addressed from a computational point of view. The first such study was by Gervasi and Prencipe [GP04]. In their work, the authors first provided a formal definition for the flocking problem based on a leader-followers approach. The authors proposed a flocking algorithm that applies to formations that are symmetric with respect to the movement of the leader without an agreement on a common coordinate system (except the unit distance).

Coble and Cook [CC98] discussed the fault tolerant coordination of robot teams. In their work, the robots can communicate between each other by exchanging messages. In particular, they applied a symbolic machine learning approach to deal with uncertainties in communication among autonomous robots. In other words, they provided a set of attributes that a robot needs to consider when determining whether its neighbor has been destroyed or it is temporarily obstructed from communicating.

Canepa and Potop-Butucaru [CP07] also studied the flocking problem based on a leader-followers approach from an algorithmic standpoint. In their work, first a leader is elected via probabilistic or deterministic algorithm, and then robots arrange each other in a symmetric formation (circle) with respect to the location of the leader in the CORDA model [PREC01]. In their work, they do not rely on the existence of a specific leader a priori known to every robot in the system. However, the leader cannot be changed during execution. Finally, their flocking algorithm allows robots to preserve the formation in movement by just moving ahead. Their algorithm does not permit any other kind of movements (turning, going backward, scaling, etc.). Furthermore, their work do not consider fault tolerance issue.

In papers [LDC05] and [LC06], the authors tried to solve the weakness of the leader-follower model, i.e., when the leader is dead, how to select a unique leader. The idea of these papers is to move some robots, in order to find the unique furthest one from the new average position of all robots as their leader.

However, when the scale of a swarm of robots becomes very large, the worst case may appear that there exist many leaders at any time and no unique leader can be found in finite time.

Gervasi and Prencipe [GP04] have proposed a flocking algorithm for robots based on a leader-followers model, but introduce additional assumptions on the speed of the robots. In particular, they proposed a flocking algorithm for formations that are symmetric with respect to the leader's movement, without agreement on a common coordinate system (except for the unit distance). However, their algorithm requires that the leader is distinguished from the robots followers.

Canepa and Potop-Butucaru [CP07] proposed a flocking algorithm in an asynchronous system with oblivious robots. First, the robots elect a leader using a probabilistic algorithm. After that, the robots position themselves according to a specific formation. Finally, the formation moves ahead. Their algorithm only lets the formation move straight forward. Although the leader is determined dynamically, once elected it can no longer change. In the absence of faulty robots, this is a reasonable limitation in their model.

1.4 Objectives

Our main (principle) motivation is to to consider the fault tolerance of flocking of a group of mobile robots. In particular, a group of mobile robots can generate and maintain a desired formation during flocking even in the presence of faulty robots.

Our work considers fault tolerance in problems of a dynamic nature (flocking). This is done while ensuring the collision avoidance during flocking. During the movement of robots, the collision between robots and the collision between robots and obstacles are also considered.

Our motivation is to find distributed flocking algorithms in different system models under the weak assumptions. The assumption include the coordinates (Chapter 4, 5 and 6), communication among robots, visibility of sensor (Chapter 7), etc. Crash faults (Chapter 4-6) and transient value failures (Chapter 6).

1.5 Contributions

There are main three contributions in this dissertation. The first one is about the design of two fault tolerant flocking algorithms. The second contribution is the discussion about self-stabilization of robot flocking when robots suffer transient failure. The last but not least is: it is the first time to apply failure detector [CT96] into robot system.

Fault tolerant flocking Flocking algorithms are studied for many years, but to the best of out knowledge, the fault tolerance of flocking has not been discussed so far. Fault tolerant flocking in some sense is a group of mobile robots flock together to coordinate effectively even in presence of robot failure.

First, we formally presents the definition of fault tolerant flocking, which builds a bridge between robots application (flocking) and dependable distributed system. After formally specifying flocking problems and developing an adequate computational model, we find as weak as possible conditions under which flocking can be achieved.

Our main contribution is two deterministic fault tolerant flocking algorithms. One is in k-bounded asynchronous system which is to show that the geometric formation agreement during dynamic flocking. Specially, all robots can coordinate effectively each other to generate a desired formation even in presence of failure of robots. But in asynchronous model, the robots can not make formation rotate freely. To lift such limitation, we propose a novel flocking algorithm which can make robots move freely, yet in a weaker model–semi-synchronous model.

Other contributions can be mentioned. First, we propose the algorithm modules, which is convenient for hardware design and using in practical applications. Second, we propose a useful method how to

distinguish the crashed robots from the correct ones. Using this method, the robot system can break the deadlock situation that the correct robots wait for the crashed robots for ever to keep formation and thus all robots can not move any more. In all, our algorithms provide a dependable platform for dynamic flocking of robots.

Self-stabilization of flocking with transient failure The property of self-stabilization is very important for dependable distributed system. In this dissertation, we analyzed the self-stabilization of flocking of a group of non-oblivious robots for transient failure.

When the robots or some component of robots experience transient failure, one dependable robot system should make sure the robots self-stabilize to a stable state. As a result, in our work, we mainly focus on the corruption of memory. Memory corruption means: some or total of the data in the memory change to other random values. For instance, in magnetic fields, the data in the memory may experience corruption. After the robots move away from the special place, the memory are not influenced any more. In this situation, how to find a way to make robot system self-stabilize and make robot flock together become a very important issue. In all, we provide a general way to analyze the self-stabilization of robot coordination.

Failure detector in robot system The abstract failure detection methods [CT96] are addressed in traditional distributed system for many years. However, there is few work that applies failure detector into robot system, except the work [AP04].

Differently from [AP04], we provide a specific application of failure detector but not an abstract one. Our failure detector provides a specific method to distinguish the crashed robots from the correct ones, as a result, to break the deadlock situation that the correct robots always wait for the crashed ones to satisfy the requirement of flocking, and can not move any more. Thus, it will inspire the design of the failure detector for specific applications.

1.6 Organization

The rest of this dissertation is structured as follows:

Chapter 2 discusses system models and definitions that are used throughout this dissertation.

- Chapter 3 states the problem that needs to address in flocking of a group of mobile robots.
- Chapter 4 presents a fault tolerant flocking algorithm for a group of asynchronous mobile robots.
- Chapter 5 discusses formation rotation issue of mobile robots and presents a free rotation flocking algorithm, yet in a weaker model – semi-synchronous model.
- Chapter 6 discusses the (im)possibility of self-stabilization of the presented fault tolerant flocking algorithms with memory corruption.
- Chapter 7 addresses on the collision avoidance issue between robots, and between robots and obstacles in the environments.
- Chapter 8 summaries the major results of this work and outlines future research directions.

Chapter 2

System Models and Definitions

2.1 System Models

The system consists of a set of autonomous mobile robots $R = \{r_1, r_2, ..., r_n\}$ locating on a two-dimensional plane. Each robot is modelled and viewed as a point in the plane, and is equipped with sensors to observe the positions of the other robots. The local view of each robot is expressed according to a private coordinate system that to be consisted of a unit of length, an origin and the directions and orientations of the two x and y coordinate axes.

The robots are completely *autonomous*. Moreover, they are *anonymous*, in the sense that they are a priori indistinguishable by appearance. They do not have any kind of identifiers that can be used during their computations. Furthermore, there is no direct means of communication among them.

Remark: *why the robots are anonymous? what advantage could it bring?* From the theoretical viewpoint, if the robots are anonymous, it makes the model weaker and better. From the practical situation, even the robots don't know each other's identity number, they still could work together. From the another side to consider, if the robots are not anonymous, that means all robots have identity number, and use communication to exchange their identity number. Furthermore, all robots need to share the same coordinate system, otherwise, they can not know who is where. Therefore, if the robots are not anonymous, it brings a lot of problem and makes the problem more complexity.

In particular, each robot R_i identifies the locations of all robots in R_i's private coordinate system; the result of this step is a multi-set of points $P = \{p_1, ...p_N\}$ defining the current *configuration*. In all, robots first get all robot's location by sensing the environment, perform computations on the sensed data, and move toward the computed destination. This behavior constitutes its cycle of *sensing*, *computing*, *moving* and *being inactive*. We define an activation cycle of a robot as follows:

- *Look*. Here, a robot *observes* the world by activating its sensors, which will return a snapshot of the positions of the robots in the system.
- *Compute*. In this event, a robot *performs* a local computation according to its deterministic algorithm. The algorithm is the same for all robots, and the result of the *compute* state is a destination point.
- *Move*. The robot *moves* toward its computed destination.

Definition 1 *The sequence* Look-Compute-Move-Wait *is called the cycle of a robot. If one such cycle of a robot is executed, we call this one* activation *of a robot.*

Specially, there is no direct communication among them. Hence, the only way to communicate is observing the other robots' position by its sensor. In the "look" step, the robots are indistinguishable, so each robot R_i knows its own location p_i, but does not know the identity of the robots at each of the other points.

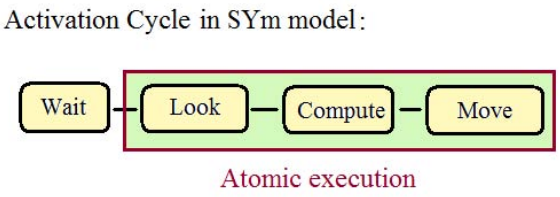


Figure 2.1: An activation cycle of a robot in SYm model.

Note that the "look" and "move" steps are carried out identically in every cycle, and the differences between different algorithms occur in the "compute" step. Moreover, the procedure carried out in the "compute" step follows the same algorithm. If the robots are oblivious, then the algorithm cannot rely on information from previous cycles.

Time is represented as an infinite sequence of time instants during which each robots can be either "active" or "inactive". Each time instant during which a robot becomes active, the robot observes its environment, computes a new position, and moves toward that position. The activation of robots is unpredictable and unknown to robots, with the guarantees that: (1) every robot becomes active at infinitely many time instants, and (2) at least one robot is active during each time instant.

Our computational model for studying and analyzing problems of coordinating and controlling a set of autonomous mobile robots follows three well-known models:

- Fully synchronous model (*FSYNC*) [AP04]
- Suzuki-Yamashita model (SYm model) [KSIS96, AOSY99, SY99]
- CORDA model [PREC01, GPRE01b]

2.1.1 Fully Synchronous Model (*FSYNC*)

The fully synchronous model is introduced in [AP04]. In fully synchronous model, each robot works according to the cycle of *look-compute-move*. In addition, robots operate according to the same clock cycles, and all robots are active on all cycles. There is a lower bound of S units on the minimum movement of a robot in a cycle, and also there is a limit of maximum movement in each cycle.

2.1.2 Suzuki-Yamashita Model (*SYm*)

In the SYm model, time is represented as an infinite sequence of discrete time instants $t_0, t_1, t_2, ...$, during which each robot can be either *active* or *inactive*. Only when a robot in wait state, it is inactive; in other state, a robot executes the activations (*look*, *compute*, *move*), which occur instantaneously. Thus, it results in a form of implicit synchronization(see Figure 2.1). In other words, this model is partially synchronous, in the sense that all robots operate according to the same clock cycles, but not all robots are necessarily active in all cycles. The activations of the different robots can be thought of as managed by a hypothetical "scheduler", whose only "fairness" obligation is that each robot must be activated and given a chance to operate infinitely often in any infinite execution.

The cycle of a robot is finite, and the activation of robots is determined by an activation schedule, which is unpredictable and unknown to the robots. At each time instant, a subset of the robots becomes active, with the guarantees that: (1) Every robot becomes active at infinitely many time instants, (2) At least one robot is active during each time instant and (3) The time between two consecutive activations is not infinite.

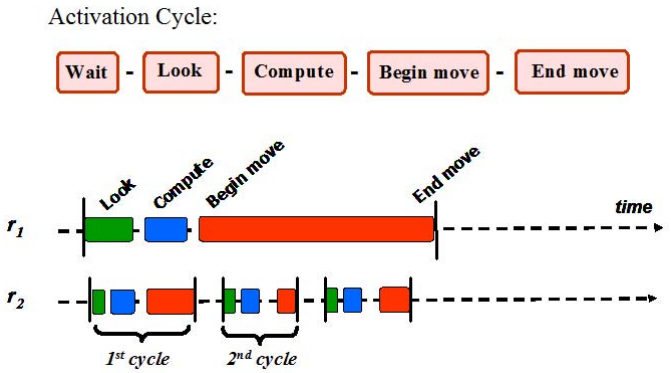


Figure 2.2: The activation of robots in CORDA model.

The atomic action executed by a robot in this model is a computation cycle. The execution of the system can be modelled as an infinite sequence of rounds. In a round one or more robots are activated and perform an activation cycle.

In every single activation, the distance that robot r can travel in one cycle is bounded by δ_{r_i}, then the algorithm returns a point of at most δ_{r_i}. This distance may be different for different robots.

2.1.3 CORDA Model

The CORDA model introduced by Prencipe [PREC01] is similar to the SYm model described above. The most notable difference is that, in the CORDA model, there is no synchronization (explicit or implicit) between robots. Specially, during the period a robot moves (i.e., between "Begin move" and "End move" in Figure 2.2), the other robots can activate many times. Furthermore, a robot can see the other robots that are moving in the plane.

In the CORDA model, the (global) time that passes between two successive states of the same robot is finite, but unpredictable. In addition, no time assumption within a state is made. This implies that the time that passes after the robot starts observing the positions of all others and before it starts moving is arbitrary, but finite. That is, the actual movement of a robot may be based on a situation that was observed arbitrarily far in the past, and therefore it may be totally different from the current situation.

They may be interrupted by the scheduler during the execution of the computation cycle. Moreover while a robot performs an action A, where A can be one of the following atomic actions: observation, local computation or motion, another robot may perform a totally different action B.

In the model, there are two limiting assumptions related to the cycle of a robot.

Assumption 1 *It is assumed that the distance travelled by a robot r in a move is not infinite. Furthermore, it is not infinitesimally small: there exists a constant $\delta_r > 0$, such that, if the target point is closer than δ_r, r will reach it; otherwise, r will move toward it by at least δ_r.*

Assumption 2 *The amount of time required by a robot r to complete a cycle (wait-look-compute-move) is not infinite. Furthermore, it is not infinitesimally small; there exists a constant $\tau_r > 0$, such that the cycle will require at least τ_r time.*

2.1.4 Relation between Models

As discussed in the paper [AP04], the relation between the three models is as follows: The FSYNC model is an extreme model since in this model, robots work according to the same clock cycles and all robots are active on all cycles. The SYm model is semi-synchronous model, in the sense that only some robots may activate in synchronous way and the execution of each activation is *atomic*. The another extreme

model is CORDA (also the weakest model), during which robots are fully asynchronous. Obviously, the relationship among them is: FSYNC is a subset of SYM, and SYM model is a subset of CORDA. That is, using the set signal, they satisfy $FSYNC \subset SYm \subset CORDA$.

2.2 Scheduler

At each configuration, a scheduler decides the set of robots which are allowed to execute their actions. A scheduler is fair if, in an infinite execution, a robot is activated infinitely often.

From the paper [DGMP06], we know there are different kinds of schedulers based on their property (fairness). First, we give the definition of *full activation cycle* for robots.

Definition 2 (full activation cycle) *A* **full activation cycle** *for any robot r_i is defined as the interval from the event Look (included) to the next instance of the same event Look (excluded).*

Definition 3 (Unfair scheduler) *Some robots in the system may never get activated while the other robots activate often.*

Definition 4 (Fair scheduler) *Each robot can activate infinite times during infinite executions.*

Definition 5 (Centralized scheduler) *Each time only one robot get activated.*

Definition 6 (Arbitrary scheduler) *At each configuration an arbitrary subset of robots is activated.*

During the fair schedulers, there is a special scheduler called bounded scheduler.

Definition 7 (Bounded scheduler) *There exists a natural number k, but the value of k is unknown, between two consecutive full activation cycles of the same robot r_i, another robot r_j can execute at most k full activation cycles.*

In this paper we consider the fair version of a subset of bounded scheduler: k-bounded-scheduler.

Definition 8 (k-bounded-scheduler) *Between two consecutive full activation cycles of the same robot r_i, another robot r_j can execute at most k full activation cycles, where k is known.*

This definition allows us to establish the following lemma:

Lemma 1 *With k bounded scheduler, if a robot is activated $k+1$ times, then all (correct) robots have completed at least one full activation cycle during the same interval.*

2.3 Failure Detection

Before we talk about failure detection, first we explain three important concepts–*failure*, *fault*, and *error* based on [ALRL04]. Correct service is delivered when the service implements the system function. A service failure, often abbreviated here to *failure*, is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an *error*. The adjudged or hypothesized cause of an error is called a *fault*. Faults can be internal or external of a system.

In a distributed system, two components of the system, both processes and channels, can fail. It is very important to define types of possible failure for further discussion about existing problems. Now, some failure models are introduced and the types of failure assumed discussed.

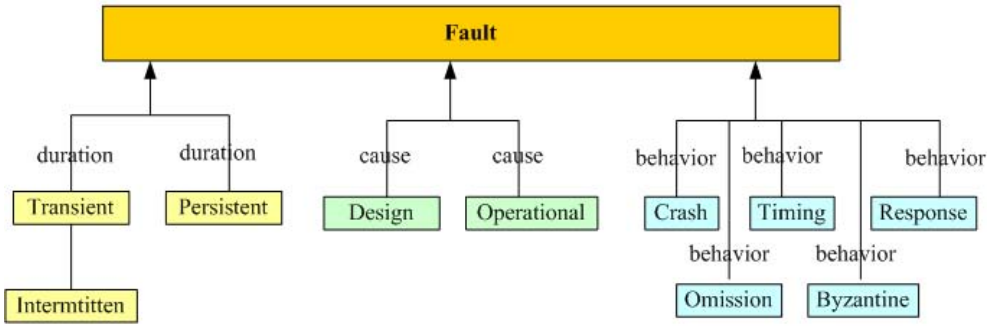


Figure 2.3: Different Classifications of failures [ALRL04].

Here, in robot system, similarly with process failure in the distributed system, the robot can fail for various reasons and they behave differently after failure. Due to the influence of outside environment, a robot or part of components of a robot, like sensor, communication equipment, or moving actuator, may crash. In this dissertation, we mainly discuss the crash of robots and put the crash of components of a robot in our future work.

2.3.1 Failure Classifications

In general, robot failures are mainly classified the following classifications with respect to the behavior of a robot after failure [ABLS90]. Figure 2.3 shows the different failure classifications in distributed system.

- *Crash* failure: A robot halts, but is working correctly until it halts.
- *Omission* failure: A robot fails to respond to incoming requests.
- *Timing* failure: A robot's response lies outside the specified time interval.
- *Response* failure: The robot's response is incorrect.
- *Arbitrary* (*Byzantine*) failure: A robot may produce arbitrary responses at arbitrary times.

Based on the duration of failure, there are *transient failure* and *persistent failure*(*permanent failure*). Following the meaning of permanent, a crash is *permanent* in the sense that a faulty robot stop and never recovers. Compared with permanent failure, transient failure is temporary failure and the period of failure is bounded.

Based on [Gerard00], there is another failure models–*Byzantine behavior*: a robot is said to be Byzantine if it executes arbitrary steps that are not in accordance with its local algorithm.

We call robots that never crash be correct robots, and robots that have crashed be faulty or incorrect robots. Note that correct/faulty are predicates over a whole execution: a robot that crashes is faulty even before the crash occurs. Of course, a robot cannot determine if it is faulty and some other components (i.e., failure detector modules) cannot make robots faulty.

We consider permanent crash (in most of this dissertation and unless stated otherwise). That is, a robot may fail by crashing, after which it stop moving and executes no actions (no movement) any more. 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.

In this chapter, we address ***crash failures***. That is, we consider *initial* crash of robots and also the crash of robots *during execution*. That is, 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

Table 2.1: Eight classes of failure detectors defined based on accuracy and completeness

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* P	*Strong* S	*Eventually Perfect* $\diamondsuit P$	*Eventually Strong* $\diamondsuit S$
Weak	Q	*Weak* W	$\diamondsuit Q$	*Eventually Weak* $\diamondsuit W$

physically present in the system, and it is seen by the other non-crashed robots. A robot that is not faulty is called a *correct* robot.

In distributed coordination of robots, a question arises: how a robot knows a robot has crashed? As we know in distributed system, a process uses failure detector to detect the other process by sending message periodically (actively or passively). First, we introduce what the failure detector is and then we discuss how to design a failure detector for robot system.

2.3.2 Failure detector

In distributed systems with failures, applications often need to determine which processes are up (operational) and which are down (crashed). This service is provided by failure detector. Failure detectors are the core of many fault-tolerant algorithms and applications, such as group membership, group communication, atomic broadcast, atomic commitment, consensus and leader election, etc. Also failure detectors are found in many systems, such as ISIS, Ensemble, Relacs, Transis, Air Traffic Control Systems.

A Failure detector can be viewed as a distributed oracle for giving a hint about the operational state of a process. In fact, a failure detector consists of failure detector modules that communicate with each other by exchanging messages. A process, called a monitoring process, can query its failure detector module about the status of some process, called a monitored process. The monitoring process thus obtains information about whether or not the monitored process is suspected to have crashed.

The principal definitions of failure detectors have been proposed by Chandra and Toueg [CT96]. They define the notion of unreliable failure detectors, which are based on the following model. For every process p_i in the system, there is a module FD_i attached that provides p_i with potentially unreliable information on the status of other processes. At any time, p_i can query FD_i and obtain a set of processes containing those that are suspected of having crashed.

The impossibility result mentioned above (i.e., several distributed agreement problems cannot be solved deterministically in asynchronous systems if even a single process might crash [CT96]), no longer holds if the system is augmented with some unreliable failure detector oracle [CT96]. An unreliable failure detector is one that can, to a certain degree, make mistakes (over and over).

The failure detector is a distributed entity that consists of all modules and whose behavior must exhibit some well-defined properties. Depending on the properties that are satisfied, a failure detector can belong to one of several classes. Now, these properties are as follows:

Completeness properties

- *Strong completeness*: Every faulty process is permanently suspected by *all* correct processes.
- *Weak completeness*: Every faulty process is permanently suspected by *some* correct process.

Accuracy properties

- *Strong accuracy*: *No* process is suspected before it crashes.
- *Weak accuracy*: *Some* correct process is never suspected.
- *Eventual strong accuracy*: There is a time after which *every* correct process is never suspected by any correct process.

- *Eventual weak accuracy*: There is a time after which *some* correct process is never suspected by any correct process.

Perfect failure detector P satisfies strong completeness and strong accuracy. There are eight such pairs, obtained by selecting one of the two completeness properties and one of the four accuracy properties.

As an example, the class $\diamondsuit S$ of failure detectors, is one of the weakest failure detectors to solve Consensus, are defined by *strong completeness* and *eventual weak accuracy*. Interestingly, any given failure detector that satisfies weak completeness can be transformed into a failure detector that satisfies strong completeness. There also exist transformation algorithms for failure detectors from strong completeness to weak completeness. This means that a failure detector with strong completeness and a failure detector with weak completeness are equivalent, thus, $\diamondsuit W$ is also the weakest failure detector for solving Consensus.

The problem is, that in asynchronous distributed systems, it is impossible to implement a failure detector of class $\diamondsuit S$ in a literal sense. The definition of $\diamondsuit S$ failure detector is nevertheless highly relevant in practice. Algorithms which assume the properties of a $\diamondsuit S$ are incredibly robust because they can tolerate an unbounded number of timing failures. In other words, and in a more pragmatic way, an application is guaranteed to make progress as long as the failure detector behaves well for long enough periods. Conversely, the application might stagnate during bad periods and resume only after the next period of stability.

2.4 Position Configuration

When a robot gets activated, first it "looks" the position of all robots and then gets the set of all robots' positions. We call such position set as *position configuration*. Specially, the formal definition is as follows:

Definition 9 (Position Configuration) *When a robot gets activated, the set of the robots' positions observed by a robot is called a position configuration.*

In non-oblivious robot system, the position configuration of robots needs to be saved in the memory. Between activations of a robot, the past position configuration is saved in the memory.

In order to denote the different flocking algorithm conveniently, we denote the flocking algorithm which can translate but can not rotate formation as $\mathcal{F}_T$. The flocking which can rotate the formation is called $\mathcal{F}_R$. Obviously, $\mathcal{F}_R$ also can translate. For the different flocking algorithms, they work in different system models. Now we can use $\mathcal{F}(MODEL)$ to denote the translating flocking algorithm which works in *MODEL*, where $MODEL$ is a variable that can be CORDA, SYm and FSYNC. If the algorithm can tolerate the memory corruption, we denote it be $\mathcal{F}^{MC}(MODEL)$, where MC is the abbreviation of *memory corruption*.

Chapter 3

Problem Statement: Flocking

3.1 General Flocking

A special behavior of large number of interacting dynamic agents called “flocking” has attracted many researchers from diverse fields of scientific and engineering disciplines. The term “flocking” in English means “moving together in large numbers”. This behavior exists in the nature in the form of flocking of birds, schooling of fish, and swarming of bacteria . Solutions to the flocking problem are useful primitives for larger tasks. For instance box pushing or cooperative manipulation, where robots can be asked to move heavy loads.

In the literature, there are many kinds of flocking definition. The earliest is stated by Craig Reynolds in his simulation program, Boids in 1986. He said the basic flocking is controlled by three simple rules:

1) *Separation* - avoid crowding neighbors (short range repulsion)

2) *Alignment* - steer towards average heading of neighbors

3) *Cohesion* - steer towards average position of neighbors (long range attraction)

Separation means that each member, or robot, of a flock tries to keep a minimum distance from every other robot in the flock. Alignment means that each robot tries to go in the same direction as the rest of the flock. Cohesion means that each robot tries to get as close as possible to the rest of the flock (huddle together). With these three simple rules, the flock moves in an extremely realistic way, creating complex motion and interaction that would be extremely hard to create otherwise. Afterward, *avoidance* is added as the fourth requirement by some researchers. It means each robot tries to not get too close, or avoid, robots in other flocks. From these four simple rules of flocking emerges apparently life-like behavior for a flock.

Numerous definitions of flocking can be found in the literature [GP04, Rey87, BH97, YB96], but few of them define the problem precisely. The rare rigorous definitions of the problem suppose the existence of a leader robot and require that the other robots, called followers, follow the leader in a desired fashion [GP04, CP07, GP01, GPRE01b, VGGP01]. The motion of the leader is not constrained by the problem. In contrast, the followers must follow the leader in such a way that the relative positions of the robots always keep a desired distance.

3.2 Formation Flocking

Informally, flocking is the formation and maintenance of a desired pattern[1] while moving, by a team of mobile robots. In this chapter, we take a leader-followers approach [GP04]. That is, at any time, there

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

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, and to keep the given formation while moving.

In order to provide a formal definition of the formation flocking problem, we use the following definitions introduced in [GP04].

Definition 10 (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 $1..., |C|$ and $dist(c_i, g_{\pi(i)})$ is the distance between two points c_i and $g_{\pi(i)}$.

Definition 11 (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 as $T_{P,E}$.*

Definition 12 (Approximation undirected formation) *An undirected target is formed up to ξ if*

$$D(E, T_{P,E}) \leq \xi.$$

Definition 13 (Flocking) *Let $r_1, r_2, ..., r_n$ be a group of robots, whose positions constitute a formation E, and let P be a pattern given in input to $r_1, r_2, ..., r_n$. We consider the presence of a leader robot to lead the group, and the other robots are followers. The leader robot can be assigned dynamically, and any of robots can potentially become a leader if the current leader is crashed. The leader can go anywhere it wants and the followers just try to follow the leader to satisfy an approximate formation. The robots solve the approximate formation problem during flocking if, starting from a given desired formation, the robots can maintain a target formation up to ξ. More specifically, during flocking, there exists a formation that satisfies $D(E, T_{P,E}) \leq \xi$.*

Definition 14 (Formation) *A formation $F = Formation(P_1, P_2, ..., P_n)$ is a configuration, with P_1 the leader of the formation, and the remaining points, the followers of the formation. Note that the leader P_1 is not distinct physically from the robot followers.*

In this chapter, we assume that the formation F is a regular polygon, d is the length of the polygon edge (known to the robots), and $\alpha = (n-2)180°/n$ is the angle of the polygon, where n is the number of robots in F.

Definition 15 (ϵ_r-Approximate Formation) *We say that robots form an approximation of the formation F if each robot r_i is within ϵ_r from its target P_i in F.*

Definition 16 (The Flocking Problem) *Let r_1,...,r_n be a group of robots, whose positions constitute a formation $F = Formation(P_1, P_2, ..., P_n)$. The robots solve the ϵ_r-* Approximate Flocking Problem *if, starting from an arbitrary formation at time t_0, $\exists t_1 \geq t_0$ such that, $\forall t \geq t_1$ all robots are at a distance of at most ϵ_r from their respective targets P_i in F, and ϵ_r is a small positive value known to all robots.*

3.3 Fault Tolerant Formation Flocking

In computing systems, a fault tolerant system can provide services even in presence of faults. Here, we consider fault tolerance of the mobile robots system. That is, a group of robots form and maintain an approximation of a regular polygon during flocking, in spite of the possible presence of faulty robots or some components failure of robots.

In robot system, robots can communicate each other by sending message to detect the failure of robots, like in traditional distributed system. But in the weak system model, like CORDA or SYm model, robots can not communicate explicitly. The only way that a robot "communicate" (obviously implicit) with the other robots is by sensor. When a robot senses the other robots, it can get the position of the other robots (locally or globally). Checking the position change of robots is a unique option to find failure of robots. So, a question may arise: How to distinguish a robot is staying in "waiting" state from a failed one. Also, when detecting the crash of robot by observing the positions of robots, it is not easy to know that a robot stays in its previous positions all the time or the other robots just move to that position. Those complicated problems need to be considered during designing flocking algorithms.

Although we do consider the presence of a leader robot to lead the group, the role of leader is assigned *dynamically* and any of the robots can potentially become a leader. In particular, after the crash of a leader, a new leader must eventually take over that role.

3.4 Challenges of Flocking

Flocking provides interesting solutions to many problems: manipulate of large objects, system redundancy, reducing time complexity for the targeted tasks, however it bring in discussion some specific difficulties. In particular, these robots should achieve their tasks without human intervention based only on the information provided by the robots in the same group. Moreover, they have to explore unknown or quasi unknown environments while avoiding collisions among themselves. Additionally, they have to be able to reorganize whenever one or more robots in the group stop to behavior correctly.

The difficulty of the problem comes from the following points:

- *Generate a desired formation* For formation flocking, all robots need to self-deploy themselves to generate a desired formation in distributed way.
- *Avoid collision between robots* During flocking, the algorithm used in the robots should avoid robot to "kill" each other.
- *Avoid collision between robots and the obstacles* When there are obstacles in the environment, the flocking robots should avoid such obstacles intelligently.
- *Detect failure* When considering the robot may crash, it is necessary to find a way to distinguish the crashed robots from the correct ones who stay in "wait" state.
- *Find the minimum capability that a robot must have* If the minimum capability of a robot must have for flocking coordination, then it is useful for designing how weak the robots.
- *Make robot system self-stabilize when transient failure occurs* Robots may experience transient failure, how to make robots system go back to a "legitimate" configuration becomes very necessary.

Generally, there are two kind of collisions that need to be avoided: *collision between robots* and *collision between robots and obstacles*. During dynamic coordination, the distributed algorithm that robots used should make sure absence of collision. Otherwise, if any two or more robots collide each other or a robot collides with the obstacles in the environment, then it cannot be called effective coordination

and as a result they will never finish their desired task. There are many work on flocking with obstacle avoidance, like in [MLPO05, SM03, YDCW07, JYJJ91]. Most of them is from the viewpoint of control theory or graph theory by designing a potential function. The robots do avoid the collision from the obstacles in the environment, while the computation complexity is not considered. Taking these challenges in mind, we manage to propose the corresponding solution in different system model. Specially, when we explore the fault tolerant formation flocking, obstacles are not considered in the models.

Chapter 4

Fault-tolerant flocking in a k-bounded asynchronous system

Flocking of a group of mobile robots has a lot of wide and practical applications, like large-scale construction, hazardous waste cleanup, space missions or exploration of dangerous or contaminated area. However, as we discussed in the previous chapter, it also go with a lot challenges to achieve the effective coordination, especially in the presence of robot crash.

Therefore, in this chapter, our work is to make mobile robots group to form a desired pattern and to move together while maintaining such formation. Unlike previous studies of the problem, we consider a system of mobile robots in which a number of them may possibly fail by crashing. Our algorithm [SYD08] ensures that the crash of faulty robots does not bring the formation to a permanent stop, and that the correct robots are thus eventually allowed to reorganize and continue moving together. Furthermore, the algorithm makes no assumption on the relative speeds at which the robots can move.

Refer to Chapter 2, we consider the following *system model*: CORDA model of Prencipe [PREC01] with a k-bounded scheduler. 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 y. Nevertheless, such assumptions (e.g., y-axis) seem like a given existing robot sensors (e.g., compasses.) Also, they agree on the clockwise/counterclokwise direction.All robots share the same unit distance. The origin of the local coordinate system of a robot is fixed. Mobile robots may possibly fail by crashing and never recover. Without such assumptions, it is fairly easy to show that the problem is impossible.

Based on the system model, in this chapter, we present a fault-tolerant flocking algorithm for a k-bounded asynchronous robot system. Our algorithm ensures that the crash of faulty robots does not bring the formation to a permanent stop, and that the correct robots are thus eventually allowed to be reorganized and continue moving together. Furthermore, the algorithm makes no assumption on the relative speeds at which the robots can move. We assume the existence of at most $(n-3)$ faulty robots in the system, where n is the number of all robots in the system.

In detail, the proposed algorithm is made of three parts. First, based on the k-bounded scheduler and approximate assumption about moving restriction of robots, we provide a method called perfect failure detector to detect any robot that has crashed. Second, agreed ranks are assigned for every robot based on the consistent position configuration. Finally, based on the above module, the third part of the algorithm ensures that the robots move together while keeping an approximation of a regular polygon, while also ensuring the necessary restrictions on their movement.

Before we proceed, we give the following notations that will be used throughout this chapter. Given some robot r_i, $r_i(t)$ is the position of r_i at time t. $y(r_i)$ denotes the y coordinate of robot r_i at some time t. Let A and B be two points, with $\overline{AB}$, we will indicate the segment starting at A and terminating at B, and $dist(A, B)$ is the length of such a segment. Finally, given a region $\mathcal{X}$, we denote by $|\mathcal{X}|$, the number

of robots in that region at time t. Let S be a set of robots, then $|S|$ indicates the number of robots in the set S.

Specially, for the definition of ϵ_r-Approximate Formation, we assume, $dist(r_i, r_j) < \epsilon_r < (k+1)dist(r_i, r_j)$, where k is from k bounded scheduler, $dist(r_i, r_j)$ is the distance between two robots r_i and r_j.

The presented fault tolerant flocking algorithm consists three modules: perfect failure detector, agreed ranking of robots and flocking algorithm. First, by using perfect failure detector, the robots get the set of positions of correct robots. The output of failure detector as the input of the ranking module. By ranking module, all correct robots are assigned agreed rank. Thus, in flocking module, a unique leader can be chosen and all the other robots follow the elected leader move together.

4.1 Perfect Failure Detection

In this section, we give a simple perfect failure detection algorithm for robots based on a $k-$bounded scheduler in the asynchronous model CORDA. Refer to Chapter 2, a perfect failure detector has two properties: *strong completeness*, and *strong accuracy*. That is,

- *Strong completeness*: Every faulty robot is permanently suspected by *all* correct robots.
- *Strong accuracy*: *No* robot is suspected before it crashes.

Before we proceed to the description of the algorithm, we make the following assumption, which is necessary for the failure detector mechanism to identify *correct* robots and *crashed* robots.

Assumption 3 *At each activation of some correct robot r_i, r_i computes as destination a position that is different from its current position. Also, a robot r_i never visits the same location for the last $k+1$ activations of r_i.*[1] *Finally, a robot r_i never visits a location that was visited by any other robot r_j during the last $k+1$ activations of r_j.*

Remark: For easy to describe our idea, here we make this assumption and later prove that it is always verified by our algorithms, thus completing the loop.

This assumption is only needed to determine a unique ranking for any number of robots (even or odd) as it is the minimal assumption to solve the leader election problem for an even number of robots, as proved by Flocchini et al. [FPSW99, FPSW01].

4.1.1 Algorithm Description

Recall that we only consider *permanent* crash failures of robots, and that crashed robots remain physically in the system. Besides, robots are anonymous. Therefore, the problem is how to distinguish faulty robots from correct ones. As shown in Algorithm 1, this failure detector part is pretty straightforward. It provides a simple perfect failure detection mechanism for the identification of correct robots. The algorithm is based on the fact that a correct robot must change its current position whenever it is activated (Assumption 3), and also relies on the definition of the $k-$bounded scheduler for the activations of robots. So, a robot r_i considers that some robot r_j is faulty if r_i is activated $k+1$ times, while robot r_j is still in the same position.

Algorithm 1 gives as output the set of positions of correct robots $S_{correct}$, and uses the following *variables*:

- $S_{PosPrevObser}$: a global variable representing the set of points of the positions of robots in the system in the previous activation of some robot r_i. These points include the positions of correct and faulty robots. $S_{PosPrevObser}$ is initialized to the empty set during the first activation of the robot.

[1] That is, r_i never revisits a point location that was within its line of movement for its last $k+1$ total activations.

- $S_{PosCurrObser}$: the set of points representing the positions of robots in the current activation of some robot r_i. Note that these points also include the positions of all robots in the system, including faulty and correct ones.
- c_j: a global variable recording how many times some robot r_j did not change its position p_j;

Assumption 3 gives as output the set of positions of correct robots $S_{correct}$.

Algorithm 1 Perfect Failure Detection (code executed by robot r_i)

Initialization: $S_{PosPrevObser} := \emptyset$; $c_j := 0$

```
1:  procedure Failure_Detection(S_PosPrevObser, S_PosCurrObser)
2:    S_correct := S_PosCurrObser;
3:    for ∀ p_j ∈ S_PosCurrObser do
4:      if (p_j ∈ S_PosPrevObser) then                {robot r_j has not moved}
5:        c_j := c_j + 1;
6:      else
7:        c_j := 0;
8:      end if
9:      if (c_j ≥ k) then
10:       S_correct = S_correct − {p_j};
11:     end if
12:   end for
13:   return (S_correct)
14: end
```

4.1.2 Correctness

The proposed failure detection algorithm (Algorithm 1) satisfies the two properties of a perfect failure detector; *strong completeness*, and *strong accuracy*. It also satisfies the *eventual agreement* property. These properties are stated respectively in Theorem 1, Theorem 2, and Theorem 3.

Lemma 2 *If some robot r_i crashes at time t_{crash}, then there is a time t_{mute} after which every correct robot detects the crash of robot r_i, that is: $\exists t_{mute}, t_{mute} \leq t_{crash} + t_{max}$, where t_{max} is the maximum time required for the slowest robot to detect the crash.*

PROOF. Let r_i be a crashed robot. Then, r_i will remain at its current position forever. Let r_f be a correct robot which is the *fastest* robot in the system. By hypothesis on the system model, the time between two consecutive activations of any robot is finite. Then, by definition of the k-bounded scheduler, and Assumption 3, robot r_f detects that r_i has crashed after $(k+1)$ activations (activations of r_f), which takes finite time.

Now, let r_s be a correct robot which is the *slowest* robot in the system. Assume that r_s is activated the least, i.e., r_s is activated only once during the k activations of robot r_f. Then, robot r_s detects that r_i has crashed after $k(k+1)$ activations (activations of r_s), which is also done in finite time. As a result, we can deduce that any correct robot r_j in the system detects the crash of robot r_i in finite time by similar arguments.

Assume that r_i crashes at time t_{crash}, and t_{max} is the maximum time required for $k(k+1)$ activations of the slowest robot, then we can compute t_{mute}, which is the time after which all correct robot detect the crash of robot r_i as follows: $t_{mute} \leq t_{crash} + t_{max}$. Since after time t_{crash}, robot r_i never moves, then after time t_{mute}, r_i will be permanently suspected by all correct robots in the system. $\square_{\text{Lemma 15}}$

As a direct consequence from Lemma 15, we derive the following theorems:

Theorem 1 ***(Strong completeness)****: Eventually every robot that crashes is permanently suspected by every correct robot.*

Theorem 2 ***(Strong accuracy)****: There is a finite time after which correct robots are not suspected by any other correct robots.*

PROOF. Let r_i and r_j be two correct robots. Assume without loss of generality that robot r_i is activated only once during k activations of robot r_j.

If robot r_i is correct, then by Assumption 3, and by the definition of $k-$bounded scheduler, r_i must move by a non-zero distance during the k activations of robot r_j. Also, by Lemma 1, r_i must have finished its move before the start of the $k+1$ activation of r_j. In addition, r_j cannot move to the position that was occupied by r_i by Assumption 3. Since r_j is also correct, it will realize that r_i has changed its position at or before the $k+1$ activation of r_j. Since the time required for the $k+1$ activations of r_j is also finite, r_j will realize that r_i is a correct robot in finite time. $\square_{\text{Theorem 2}}$

From Theorem 1 and Theorem 2, we can conclude that Algorithm 1 has the following property:

Theorem 3 ***(Eventual agreement)****: There is a finite time after which, all correct robots agree on the same set of correct robots in the system.*

4.2 Agreed Ranking for Robots

In this section, we provide an algorithm that gives a unique ranking (or identification) to every robot in the system since we assume that robots are anonymous, and do not have any identifier to allow them to distinguish each other. The algorithm allows correct robots to compute and agree on the same ranking. In particular, the ranking mechanism is needed for the election of the leader of the formation. Note that a deterministic leader election is impossible without a shared y-axis [FPSW01]. Therefore, we assume that robots agree on the y-axis.

4.2.1 Algorithm Description

We first assume that robots are not located initially at the same point. That is, robots are not in the gathering configuration [DGMP06], because it may become impossible to separate them later. In other words, consider two robots that happen to have the same coordinate system and that are always activated together. It is impossible to separate them deterministically. In contrast, it would be trivial to scatter them at distinct positions using randomization (e.g., [DP07]), but this is ruled out in our model.

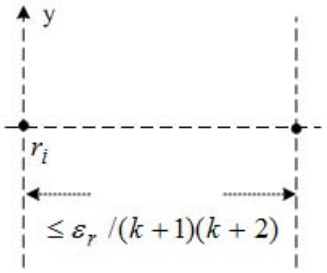
The ranking assignment is given in Algorithm 2. The algorithm takes as input the set of positions of correct robots in the system $S_{correct}$, and returns as output an ordered set of the positions in $S_{correct}$, called $RankSequence$. This ranking of the positions of robots in $S_{correct}$ gives to every robot a unique identification number. The computation of $RankSequence$ is done as follows: $RankSequence = \{S_{correct}, <\}$, where the relation " $<$ "is defined by comparing the y coordinates of the points in $S_{correct}$, and breaking ties from left to right. In other words, the positions of robots in $S_{correct}$ are sorted by decreasing order of $y-$coordinate, such that the robot with greatest y-coordinate is the first in $RankSequence$. When two or more robots share the same y-coordinate, the clockwise direction (called right hand) is used to determine the sequence; a robot r_i that has a robot r_j on its right hand, has a lower rank than r_j in $RankSequence$.

Algorithm 2 Ranking_Correct_Robots (code executed by robot r_i)

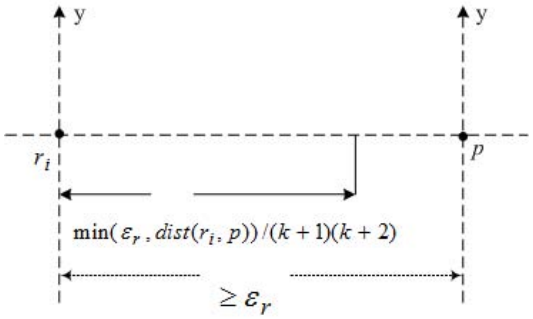
Input: $S_{correct}$: set of positions of correct robots;
Output: $RankSequence$: Ordered set of positions of correct robots $S_{correct}$;
Initialization: $counter_{act}$:= a global variable recording the number of activations of robot r_i;
procedure Ranking_Correct_Robots($S_{correct}$)
  When r_i **is activated**
  $counter_{act} := counter_{act} + 1$;
  $Left(r_i)$:= is the ray starting at r_i and perpendicular to its $y-$axis in counter-clockwise direction.
  Sort the $y-$coordinates of robots in $S_{correct}$ in decreasing order.
  if ($\forall r_j, r_k \in S_{correct}, y(r_j) \neq y(r_k)$) **then**
    $RankSequence :=$ the set $S_{correct}$ in order of decreasing $y-$coordinate;
  else if $y(r_j) = y(r_k)$ **then**
    if (r_j is on $Left(r_k)$) **then**
      $RankSequence := r_j < r_k$;
    else
      $RankSequence := r_k < r_j$;
    end if
  end if
  if ($counter_{act} \leq k$) **then**
    Lateral_Move_Right();
  end if
  Return($RankSequence$);
end

Algorithm 3 Procedure Lateral Move Right (code executed by robot r_i).

procedure Lateral_Move_Right()
  $Right(r_i)$ = is the ray starting at r_i and perpendicular to its $y-$axis in clockwise direction;
  if (If no other robot on $Right(r_i)$) **then**
    r_i moves by at most $\epsilon_r/(k+1)(k+2)$ to $Right(r_i)$(see Figure 4.1(a));
  else {*some robots are in* $Right(r_i)$ *including faulty robots*}
    p = the position of the nearest robot to r_i in $Right(r_i)$;
    r_i moves by $min(\epsilon_r/(k+1)(k+2), dist(r_i, p)/(k+1)(k+2))$ to $Right(r_i)$(see Figure 4.1(b));
  end if
end



(a) r_i has no neighbor on *Right*(r_i): r_i moves at most $\epsilon_r/(k+1)(k+2)$ to $Right(r_i)$.



(b) r_i has the nearest neighbor p on *Right*(r_i): r_i moves by $mathitmin(\epsilon_r/(k+1)(k+2), dist(r_i,p)/(k+1)(k+2))$ to $Right(r_i)$.

Figure 4.1: Zone of movement of the leader.

In order for robots to agree on the same $RankSequence$ initially, some restrictions on their movement are required during their first k activations. The movement restriction is given by procedure Lateral_Move_Right(), and it is made in a way that all robots compute the same $RankSequence$ during their first k activations. In particular, a robot r_i that does not have robots on $Right(r_i)$ can move by at most the distance $\epsilon_r/(k+1)(k+2)$ along $Right(r_i)$ in order to preserve the same $y-$coordinate. Otherwise, r_i moves by $min(\epsilon_r/(k+1)(k+2), dist(r_i,p)/(k+1)(k+2))$ along $Right(r_i)$, where p is the position of the nearest robot to r_i in $Right(r_i)$.

Note that, the bounded distance $min(\epsilon_r/(k+1)(k+2), dist(r_i,p)/(k+1)(k+2))$ set on the movement of robots is conservative, and is sufficient to avoid collisions between robots, and to satisfy Assumption 3.

In order to understand the working procedure of Algorithm 2, we give an example shown in Figure 5.1. By using Algorithm 2, $RankSequence$ can be gotten as $\{a, b, c, e, d\}$.

4.2.2 Correctness

From Algorithm 2, we obtain the following lemmas. In particular, Algorithm 2 gives a unique ranking to every robots in the system, and also ensures no collisions between robots.

Lemma 3 *Algorithm 2 gives a unique ranking to every correct robot in the system.*

PROOF. The proof is trivial. Since, robots agree on the direction and orientation of the

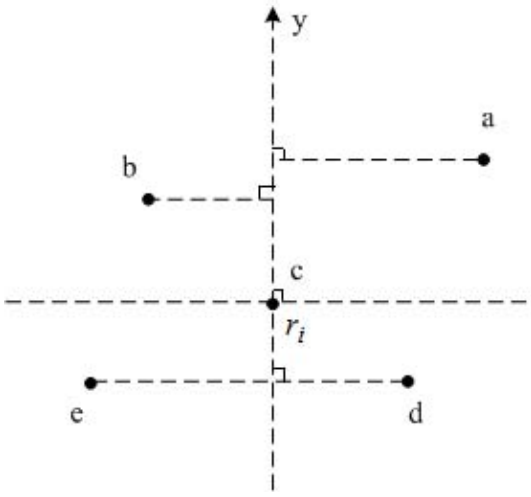


Figure 4.2: An example of Ranking_Correct_Robots: a, b, c, d, e are the locations of robots in system, e and d has the same y values.

$y-$axis, then, by Algorithm 2, all robots with different $y-$coordinates will have different ranks. In addition, for robots who have the same $y-$coordinate, the clockwise direction is used to determine the sequence. Since, robots agree on the clockwise direction, then two distinct robots having the same $y-$coordinate cannot see each other in the same direction. Thus, a unique ranking is given to each of these robots. □Lemma 3

Lemma 4 *By Algorithm 2, there is a finite time after which, all correct robots agree on the same initial sequence of ranking,* $RankSequence$.

PROOF. Assume without loss of generality that robot r_i is the first robot that was activated by the scheduler. That is, r_i has seen the initial configuration of the robots. Then, the proof consists of showing that all other robots (correct) compute the same sequence of ranking as r_i.

We first show that Algorithm 2 preserves the same sequence of ranking computed by r_i. Assume that robot r_j is activated after robot r_i has finished one full cycle. Recall that r_i has changed its position based on Assumption 3. Then, we will show that robot r_j will compute the same rank sequence as r_i no matter what the movement taken by r_i:

1. Robot r_i moves toward $Right(r_i)$: Since such a move does not change the $y-$coordinate of r_i, and r_j executes the same algorithm as r_i, then r_j will compute the same rank sequence as r_i.
2. Robot r_i moves toward the closest robot to $Right(r_i)$, say r_c by the distance $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_c)\ /(k+1)(k+2))$: Since such a distance is less than the distance $dist(r_i, r_c)/k$, then such a move does not change the order of r_i and r_c with respect to left and right, and also it preserves the same $y-$coordinate of r_i and r_c. Thus, by similar arguments as above, r_j will compute the same rank sequence as r_i.

The same proof applies to the other robots that are activated after r_i and r_j, by similar arguments. By Lemma 3, the rank sequence, $RankSequence$ computed by all correct robots is unique. In addition, by the assumption of the k-bounded scheduler, a robot is activated at least once during k activations, and its cycle is finite by Assumption 2. Consequently, after

k activations of the same robot in the system, every other correct robot is activated at least once, and have computed the same ranking sequence. Such computation is done in finite time and the lemma follows. $\square_{\text{Lemma 4}}$

Lemma 5 *The ranking algorithm (Algorithm 2) guarantees no collisions between the robots in the system.*

PROOF. The proof is straightforward. The only movement allowed by Algorithm 2 is to make robots move along the perpendicular of their $y-$axes, toward their right. Assume that robot r_i is one of the robots in the system. There are two cases to consider, depending on whether robot r_i has robots on $Right(r_i)$ or not. First, assume that robot r_i has a different y coordinate from the other robots in the system. Then, it is trivial that r_i will not collide with any of these robots because they do not belong to its line of movement. In addition, they will not arrive at its line of movement because they move in parallel to the y-axis of r_i, by Algorithm 2.

Now assume that robot r_i has the same y coordinate as another robot in the system, say r_j. Assume without loss of generality that r_j is the closest to r_i in $Right(r_i)$. By Algorithm 2, r_i is allowed to move at each activation cycle by at most $min(dist(r_i, r_j)/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$. Then, even in the worst case when r_i is activated each time during k activations, however r_j is not, we will not have the situation where r_i collides with r_j after k activations of r_i because r_i will not reach r_j since the distance $k \cdot dist(r_i, r_j)/(k+1)(k+2)$ is strictly less than $dist(r_i, r_j)$. $\square_{\text{Lemma 5}}$

4.3 Dynamic Fault-tolerant Flocking

In this section, we propose a dynamic fault tolerant flocking algorithm. Using this algorithm, a group of robots can dynamically generate an approximation of a regular polygon (Definition 15), and maintain it while moving. Our flocking algorithm relies on the existence of two devices, namely a *perfect failure detector* device and a *ranking* device, which were represented respectively in Algorithm 1, and Algorithm 2.

4.3.1 Algorithm Description

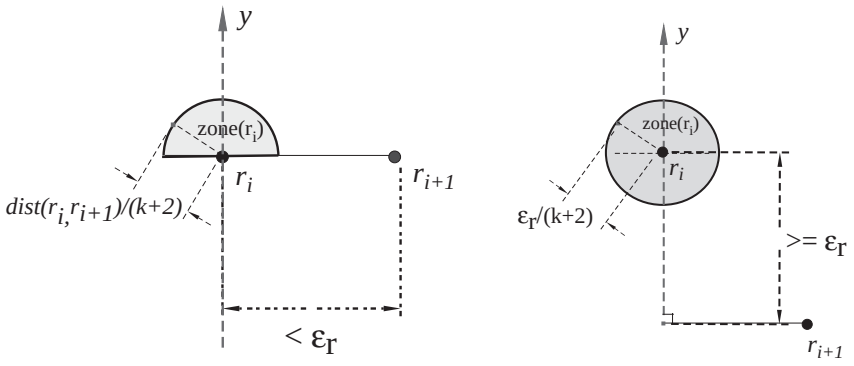
The flocking algorithm is depicted in Algorithm 4, and takes as *input* the desired length of the polygon edge d, and the *history* of robot r_i, which includes the following variables:

- $S_{PosPrevObser}$: the set of positions of robots in the system during the last observation of robot r_i.
- $HistoryMove$: the set of points on the plane visited by robot r_i during its last previous $k+1$ activations.
- nbr_{act}: a counter recording the last previous $k+1$ activations of robot r_i.

The overall idea of Algorithm 4 is as follows. First, when robot r_i gets activated, it executes the following steps:

(1) It takes a snapshot of the current positions $S_{PosCurrObser}$ of robots in the system.

(2) Robot r_i calls the failure detection module to get the set of correct robots, $S_{correct}$.



(a) r_i and r_{i+1} have the same y-coordinate, and $dist(r_i, r_{i+1}) < \epsilon_r$: $Zone(r_i)$ is the half circle with radius $dist(r_i, r_{i+1})/(k+1)(k+2)$.

(b) r_i and r_{i+1} do not have the same y-coordinate, and $dist(r_i, proj_{r_{i+1}}) \geq \epsilon_r$: $Zone(r_i)$ is the circle with radius $\epsilon_r/(k+1)(k+2)$.

Figure 4.3: Zone of movement of the leader.

(3) Robot r_i calls the ranking system, and gets a total ordering on the set of correct robots $S_{correct}$, called $RankSequence$.

(4) Depending on the rank of robot r_i in $RankSequence$, robot r_i executes the procedure described in Algorithm 5; Flocking_Leader(RankSequence, d, $\mathsf{nbr}_{\mathsf{act}}$, HistoryMove) if it has the first rank in $RankSequence$ (i.e., the leader). Otherwise, robot r_i is a follower, and it executes the procedure Flocking_Follower(RankSequence, d, $\mathsf{nbr}_{\mathsf{act}}$, HistoryMove), described in Algorithm 6.

(5) Robot r_i is a leader. First, r_i computes the points of the formation $P_1, ..., P_n$ as in Definition 15, with its location as the first point P_1 in the formation. The targets of the followers are the other points of the formation, and they are assigned to them based on their order in the $RankSequence$. After that, the leader will initiate the movement of the formation, while preserving the same rank sequence, keeping an approximation of the regular polygon, and also avoiding collisions with followers.

In order to prevent collisions between robots, the algorithm must guarantee that no two robots ever move to the same location. Therefore, the algorithm defines a movement zone for each robot, within which the robot must move. The zone of the leader, referred to as $Zone(r_i)$, is defined depending on the position of the next robot r_{i+1} in $RankSequence$. Let us denote by $proj_{r_{i+1}}$, the projection of robot r_{i+1} on the $y-$axis of r_i. The zone of the leader is defined as follows:

- r_i and r_{i+1} have the same y coordinate: $Zone(r_i)$ is the half circle with radius $min(dist(r_i, r_{i+1})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$, centered at r_i and above r_i (refer to Figure 4.3(a)).
- r_i and r_{i+1} do not have the same y coordinate: $Zone(r_i)$ is the circle, centered at r_i, and with radius $min(dist(r_i, proj_{r_{i+1}})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$ (refer to Figure 4.3(b)).

After determining its zone of movement $Zone(r_i)$, robot r_i needs to determine if there are crashed robots within $Zone(r_i)$. Let $S_{CrashInZone}$ be the set of positions of the crashed robots in $Zone(r_i)$. If $S_{CrashInZone}$ is equal to the empty set, then robot r_i can move to any desired target within $Zone(r_i)$, satisfying Assumption 3. Otherwise, robot r_i can move within $Zone(r_i)$ by excluding the points in $S_{CrashInZone}$, and satisfying Assumption 3.

(6) Robot r_i is a follower. First, r_i assigns the points of the formation $P_1, ..., P_n$ to the robots in $RankSequence$ based on their order in $RankSequence$. Subsequently, robot r_i determines its

target P_i based on the current position of the leader (P_l), and the polygon angle α given in the following equation: $\alpha = (n-2)\pi/n$, where n is the number of robots in the formation.

After that, robot r_i must follow the leader. while preserving the rank sequence of robots, keeping an approximation of the polygon, and also avoiding collisions with the other robots. In order to ensure no collisions between robots, the algorithm also defines a movement zone for each robot follower, within which the robot must make its movement. The zone of a follower, referred to as $Zone(r_i)$ is defined depending on the position of the previous robot r_{i-1} and the next robot r_{i+1} to r_i in $RankSequence$. Before we proceed, we denote by $proj_{r_{i-1}}$, the projection of robot r_{i-1} on the $y-$axis of robot r_i. Similarly, we denote by $proj_{r_{i+1}}$, the projection of robot r_{i+1} on the $y-$axis of r_i. The zone of movement of a robot follower is defined depending on the locations of r_{i-1} and r_{i+1} as follows:

- r_i, r_{i-1} and r_{i+1} have the same y coordinate, then $Zone(r_i)$ is the segment $\overline{r_i p}$, with p as the point at distance $min(dist(r_i, r_{i+1})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$ from r_i (see Figure 4.4(a)).
- r_i, r_{i-1} and r_{i+1} do not have the same y coordinate, then $Zone(r_i)$ is the circle centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (see Figure 4.4(b)).
- r_i and r_{i+1} have the same y coordinate, however r_{i-1} does not, then $Zone(r_i)$ is the half circle centered at r_i above it, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$.
- r_i and r_{i-1} have the same y coordinate, however r_{i+1} does not, then $Zone(r_i)$ is the half circle centered at r_i below it, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i-1})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (see Figure 4.4(c)).

As mentioned before, the bounded distance $min(\epsilon_r/(k+1)(k+2), dist(r_i, p)/(k+1)(k+2))$ set on the movement of robots is conservative, and is sufficient to avoid collisions between robots, and also to satisfy Assumption 3 (this will be proved later).

For the sake of clarity, we do not describe explicitly in Algorithm 6 the zone of movement of the last robot in the rank sequence. The computation of its zone of movement is similar to that of the other robot followers, with the only difference being that it does not have a next neighbor r_{i+1}. So, if robot r_i has the same $y-$coordinate as its previous neighbor r_{i-1}, then its zone of movement $Zone(r_i)$ is the half circle with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i-1})/(k+1)(k+2))$, centered at r_i and below r_i. Otherwise, its zone is the circle centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2))$.

After determining $Zone(r_i)$, robot r_i needs to determine if it can progress toward its target $\mathsf{Target}(r_i)$. Note that, $\mathsf{Target}(r_i)$ may not necessarily belong to $Zone(r_i)$. To do so, robot r_i computes the intersection of the segment $\overline{r_i \mathsf{Target}(r_i)}$ and $Zone(r_i)$, called $Intersect$. If $Intersect$ is equal to the position of r_i, then r_i will move toward its right as given by the procedure $\mathsf{Lateral\_Move\_Right}()$. Otherwise, r_i moves along the segment $Intersect$ as much as possible, while avoiding to reach the location of a crashed robot in $Intersect$, if any, and satisfying Assumption 3. In any case, if r_i is not able to move to any point in $Intersect$, except its current position, it moves to its right as in the procedure $\mathsf{Lateral\_Move\_Right}()$.

Note that the followers can move in any direction by adaptation of their target positions with respect to the new position of the leader. When the leader is idle, robot followers move within the distance $\epsilon_r/(k+1)(k+2)$ or smaller in order to keep an approximation of the formation with respect to the position of the leader, and preserve the rank sequence.

Algorithm 4 Dynamic Fault-tolerant Flocking (code executed by robot r_i)

Input:
Memory(r_i):$S_{PosPrevObser}$;
$HistoryMove$;nbr_{act};
d = the desired distance of the polygon edge;
When r_i **is activated**
r_i takes a snapshot of the positions $S_{PosCurrObser}$ of robots;
$S_{correct}$ = Failure_Detection($\mathsf{S_{PosPrevObser}}$, $\mathsf{S_{PosCurrObser}}$);
$RankSequence$ = Ranking_Correct_Robots($\mathsf{S_{correct}}$);
$leader$:= first robot in $RankSequence$;
if ($r_i = leader$) **then** {*leader*}
  Flocking_Leader(RankSequence, d, $\mathsf{nbr_{act}}$, HistoryMove);
else {*follower*}
  Flocking_Follower(RankSequence, d, $\mathsf{nbr_{act}}$, HistoryMove);
end if

Algorithm 5 Flocking Leader: Code executed by a robot leader r_i.

procedure Flocking_Leader($RankSequence$, d, nbr_{act}, $HistoryMove$)
  $n := |RankSequence|$;
  $\alpha := (n-2)\pi/n$;
  P := Formation($P_1, P_2, ..., P_n$) as in Definition 14;
  $P_1 :=$ current position of the leader;
  r_{i+1}:= next robot to r_i in $RankSequence$;
  $proj_{r_{i+1}}$:= the projection of r_{i+1} on $y-$axis of r_i;
  if ($proj_{r_{i+1}} = r_i$) **then** {r_i *has same* $y-$*coordinate as* r_{i+1}}
    $Zone(r_i)$:= half circle with radius $min(dist(r_i, r_{i+1})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$, centered at r_i and above r_i (refer to Fig. 4.3(a));
  else
    $Zone(r_i)$:= the circle centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (refer to Fig. 4.3(b));
  end if
  $S_{CrashInZone} :=$ the set of positions of crashed robots in $Zone(r_i)$;
  if ($S_{CrashInZone} \neq \emptyset$) **then**
    $leader$ moves to a desired point $\mathsf{Target}(r_i)$ within $Zone(r_i)$, excluding the points in $S_{CrashInZone}$, and the points in $HistoryMove$;
  else
    $leader$ moves to a desired point $\mathsf{Target}(r_i)$ within $Zone(r_i)$, excluding the points in $HistoryMove$;
  end if
  $CurrMove :=$ the set of points on the segment $\overline{r_i \mathsf{Target}(r_i)}$;
  if ($nbr_{act} \leq k+1$) **then**
    $HistoryMove := HistoryMove \cup CurrMove$;
  else
    $HistoryMove := CurrMove$;
    $nbr_{act} := 1$;
  end if
end

4.3.2 Correctness of the Algorithm

In this section, we prove the correctness of our flocking algorithm by first showing that correct robots agree on the same ranking of robots during the execution of Algorithm 4 (Theorem 4). Second, we prove that no two correct robots ever move to the same location, and that a correct robot never moves to a location occupied by a faulty robot (Theorem 5). Then, we show that all correct robots dynamically form an approximation of a regular polygon in finite time, and keep this formation while moving (Theorem 6). Finally, we prove that our algorithm tolerates permanent failures of robots (Theorem 7).

Lemma 6 *Algorithm 4 satisfies Assumption 3.*

PROOF. To prove the lemma, we first show that any robot r_i in the system is able to move to a destination that is different from its current location, and robot r_i never visits a point location that was within its line of movement for its last $k+1$ activations. Then, we show that a robot r_i never visits a location that was visited by another robot r_j during the last previous $k+1$ activations of r_j.

First, assume that robot r_i is the leader. By Algorithm 4, its zone of movement $Zone(r_i)$ is either a circle or a half circle on the plane (centered at r_i, and with radius greater than zero), excluding the points in its history of moves $HistoryMove$ for the last previous $k+1$ activations, and the positions of crashed robots. Since, $Zone(r_i)$ is composed of an infinite number of points, the positions of crashed robots are finite, and $HistoryMove$ is a strict subset of $Zone(r_i)$, then robot r_i can always compute and move to a new location that is different from the locations visited by r_i during its $k+1$ activations.

Now, assume that robot r_i is a follower. Three cases follow depending on the zone of movement of r_i. Let r_{i-1} and r_{i+1}, be respectively the previous, and next robots to r_i in $RankSequence$.

- Consider that r_i is the last in the $RankSequence$. Robot r_i can move to a free position of its right hand, excluding $HistoryMove$.
- Consider that $Zone(r_i)$ is the *segment* with length $min(\epsilon_r, dist(r_i, r_{i+1}))/(k+1)(k+2)$, excluding r_i. Since, such case occurs only when r_{i-1}, r_i, and r_{i+1} have the same y coordinate, and robot r_i is only allowed to move to $Right(r_i)$. Then, r_i can always move to a free position in $Right(r_i)$, which does not belong to $HistoryMove$, and which also excludes the positions of crashed robots since they are finite, and there exists an infinite number of points in $Zone(r_i)$.
- Consider that $Zone(r_i)$ is either a circle or a half circle, centered at r_i, and with a radius greater than zero, excluding its history of moves $HistoryMove$ for the last previous $k+1$ activations, and the positions of crashed robots. By similar arguments as above, we have $Zone(r_i)$ is composed of an infinite number of points, $HistoryMove$ is a strict subset of $Zone(r_i)$, and the positions of crashed robots are finite. Thus, robot r_i can always compute and move to a new location that is different from the locations visited by r_i during its last $k+1$ activations.

We now show that robot r_i never visits a location that was visited by another robot r_j during the last previous $k+1$ activations of r_j. Without loss of generality, we consider robot r_i and its next neighbor r_{i+1}. The same proof holds for r_i and its previous neighbor r_{i-1}. Observe that if r_i and r_{i+1} are moving away from each other, then neither robots move to a location that was occupied by the other one for its last $k+1$ activations.

Now assume that both robots r_i and r_{i+1} are moving to the same direction, then we will show that r_i never reaches r_{i+1} after $k+1$ activations of r_{i+1}. Assume the worst case where

r_{i+1} is activated once during each k activations of r_i. Then, after $k+1$ activations of r_{i+1}, r_i will move toward r_{i+1} by a distance of at most $min(dist(r_i, r_{i+1})(k+1)^2/(k+1)(k+2), \varepsilon_r(k+1)^2/(k+1)(k+2))$. which is strictly less than $dist(r_i, r_{i+1})$, hence r_i is unable to reach r_{i+1}, and move to a location that was occupied by r_{i+1} for the last $k+1$ activations of r_{i+1}.

Finally, we assume that both r_i and r_{i+1} are moving toward each other. In this case, we assume the worst case when both robots are always activated together. After $k+1$ activations of either r_i or r_{i+1}, each of them will travel toward the other one by at most the distance $dist(r_i, r_{i+1})(k+1)/(k+1)(k+2)$. Consequently, $2dist(r_i, r_{i+1})/(k+2)$ is always strictly less than $dist(r_i, r_{i+1})$ because $k \geq 1$. Hence, neither r_i or r_{i+1} moves to a location that was occupied by the other during its last $k+1$ activations, and the lemma holds. $\square_{\text{Lemma 6}}$

Corollary 1 *By Algorithm 4, there is no overlap between the zones of movement of any two correct robots in the system.*

Agreement on Ranking.

In this section, we show that correct robots agree initially on the same ranking sequence, and that the order of the sequence is preserved always for correct robots even in the presence of failure of robots.

Lemma 7 *By Algorithm 4, correct robots always agree on the same $RankSequence$ when there is no crash. Moreover, if some robot r_j crashes, there is a finite time after which, all correct robots exclude r_j from the ordered set $RankSequence$, and keep the same total order in $RankSequence$.*

PROOF. By Lemma 4, after the first k activations of any robot in the system, all correct robots agree on the same sequence of ranking, $RankSequence$. In the following, we first show that the $RankSequence$ is preserved during the execution of Algorithm 4 when there is no crash in the system. Second, we show that if some robot r_j has crashed, there is a finite time after which correct robots agree on the new sequence of ranking, excluding r_j.

(1) First, assume there is no crash in the system: we consider three consecutive robots r_a, r_b and r_c in $RankSequence$, such that $r_a < r_b < r_c$. We prove that the movement of r_b does not allow it to swap ranks with r_a or r_c in the three different cases that follow:

- r_a, r_b and r_c share the same y coordinate. In this case, r_b moves by $min(\epsilon_r/(k+1)(k+2), dist(r_b, r_c)/(k+1)(k+2))$ along the segment $\overline{r_b r_c}$. Such a move does not change the y coordinate of r_b, and it also does not change its rank with respect to r_a and r_c because it stays between r_a and r_c, and it never reaches either r_a nor r_b, by the restrictions on the algorithm.
- r_a, r_b and r_c do not share the same y coordinate. In this case, the movement of r_b is restricted within a circle $\mathcal{C}$, centered at r_b, and having a very small radius that does not allow r_b to reach the same y coordinate as either r_a nor r_c. In particular, the radius of $\mathcal{C}$ is equal to $min(\epsilon_r/(k+1)(k+2), dist(r_b, proj_{r_a})/(k+1)(k+2), dist(r_b, proj_{r_c})/(k+1)(k+2))$, which is less than $dist(r_b, proj_{r_a})/k$, and $dist(r_b, proj_{r_c})/k$, where $proj_{r_a}$ and $proj_{r_c}$ are respectively, the projections of robot r_a and r_c on the $y-$axis of r_b. Hence, the restriction on the movement of r_b does not allow it to have a y coordinate greater than or equal to that of r_a, and it also does not allow r_b to have a y coordinate less than or equal to that of r_c. Thus, r_b does not swap its rank with either r_a or r_b.

Algorithm 6 Flocking Follower: Code executed by a robot follower r_i.

procedure Flocking_Follower($RankSequence$,d,nbr_{act},$HistoryMove$)
  $n := |RankSequence|$;
  $\alpha := (n-2)\pi/n$;
  $P :=$ Formation($P_1, P_2, ..., P_n$) as in Definition 14;
  $P_1 :=$ current position of the leader;
  $\forall r_j \in RankSequence$, $\mathsf{Target}(r_j) = P_j \in Formation(P_1, P_2, ..., P_n)$;
  if ($\forall r_j \in RankSequence$, r_j is within ϵ_r of P_j) **then** {*Formation = True*}
    Lateral_Move_Right();
  else {*Flocking and formation generation*}
    r_{i-1}:= previous robot to r_i in $RankSequence$;
    $proj_{r_{i-1}}$:= the projection of r_{i-1} on $y-$axis of r_i;
    r_{i+1} := next robot to r_i in $RankSequence$;
    $proj_{r_{i+1}}$:= the projection of r_{i+1} on $y-$axis of r_i;
    if ($proj_{r_{i-1}} = r_i \wedge proj_{r_{i+1}} = r_i$) **then** {$r_i$ *has the same y coordinate as its neighbors*}
      $Zone(r_i)$:= segment with length $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$ starting at r_i to $Right(r_i)$ (see Fig. 4.4(a));
    else if ($proj_{r_{i-1}} \neq r_i) \wedge (proj_{r_{i+1}} \neq r_i$) **then**
      $Zone(r_i)$:= circle centered at r_i, with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (see Fig. 4.4(b));
    else if ($proj_{r_{i-1}} \neq r_i \wedge proj_{r_{i+1}} = r_i$) **then**
      $Zone(r_i)$:= half circle centered at r_i, with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$, and above r_i;
    else {r_i *has different y coordinate from next robot*}
      $Zone(r_i)$:= half circle centered at r_i, with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i-1})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$, and below r_i (see Fig. 4.4(c));
    end if
    $Intersect :=$ the intersection of the segment $\overline{r_i \mathsf{Target}(r_i)}$ with $Zone(r_i)$;
    if ($Intersect \neq r_i$) **then** {r_i *is able to progress to its target*}
      $S_{CrashInLine} :=$ the set of crashed robots that belongs to the segment $Intersect$;
      if ($S_{CrashInLine} = \emptyset$) **then**
        r_i moves linearly to the last point in $Intersect$, excluding the points in $HistoryMove$;
      else
        $r_c :=$ the closest crashed robot to r_i in $Intersect$;
        r_i moves linearly to the last point in the segment $\overline{r_i r_c}$, excluding the point r_c, and the points in $HistoryMove$;
      end if
    else
      Lateral_Move_Right();
    end if
  end if
  $CurrMove :=$ the set of points on the segment $\overline{r_i \mathsf{Target}(r_i)}$;
  if ($nbr_{act} \leq k+1$) **then**
    $HistoryMove := HistoryMove \cup CurrMove$;
  else
    $HistoryMove := CurrMove$;
    $nbr_{act} := 1$;
  end if
end

- Two consecutive robots have the same y coordinate, (say r_a and r_b), however r_c does not. This case is almost similar to the previous one. The movement of r_b is restricted within a half circle, centered at r_b, and below it, and with a very small radius that does not allow r_b to reach the same y coordinate as r_c, and also does not allow r_b to swap positions with r_c. In particular, the radius of that half circle is equal to $min(\epsilon_r/(k+1)(k+2), dist(r_a, r_b)/(k+1)(k+2), dist(r_b, proj_{r_c})/(k+1)(k+2))$, which is less than $dist(r_a, r_b)/k$, and also less than $dist(r_b, proj_{r_c})/k$, where $proj_{r_c}$ is the projection of robot r_c on the $y-$axis of r_b. Hence, the restriction on the movement of r_b does not allow it to change its rank with respect to r_a, and also it does not allow it to have a y coordinate less than or equal to that of r_c. Thus, r_b does not swap rank with either r_a or r_b.

Since, all robots execute the same algorithm, then the proof holds for any two consecutive robots in $RankSequence$. Note that, we do not distinguish between the algorithm executed by the leader, and the one executed by the follower, because the restrictions made on their movements are the same.

(2) Now consider the case where some robot r_j crashes: From what we proved above, we deduce that all robots agree and preserve the same sequence of ranking, $RankSequence$ in the case of no crash. In other words, by restrictions on the movements of robots, the total order in $RankSequence$ never changes. Assume now that a robot r_j crashes. By Lemma 15, we know that there is a finite time after which all correct robots detect the crash of r_j. Hence, there is a finite time after which correct robots exclude robot r_j from the ordered set $RankSequence$.

In conclusion, we find that the total order in $RankSequence$ is preserved for correct robots during the entire execution of Algorithm 4. $\square_{\text{Lemma 7}}$

The following Theorem is a direct consequence from Lemma 7.

Theorem 4 *By Algorithm 4, all robots agree on the total order of ranking during the entire execution of the algorithm.*

Collision-Freedom.

Lemma 8 *Under Algorithm 4, no two correct robots ever move to the same location. Also, no correct robot ever moves to a position occupied by a faulty robot.*

PROOF. To prove that no two correct robots ever move to the same location, we show that any robot r_i always moves to a location within its own zone $Zone(r_i)$, and the rest follows from the fact that the zones of two robots do not intersect (Corollary 1). By restriction on the algorithm, r_i must move to a location $\mathsf{Target}(r_i)$, which is within $Zone(r_i)$. Since, r_i belongs to $Zone(r_i)$, $Zone(r_i)$ is a convex form or a line segment, and the movement of r_i is linear, so all points between r_i and $\mathsf{Target}(r_i)$ must be in $Zone(r_i)$.

Now we prove that, no correct robot ever moves to a position occupied by a crashed robot. By Theorem 1, robot r_i can compute the positions of crashed robots in finite time. Moreover, by Lemma 6, robot r_i always has free destinations within its zone $Zone(r_i)$, which excludes crashed robots. Finally, Algorithm 4 restricts robots from moving to the locations that are occupied by crashed robots. Thus, robot r_i never moves to a location that is occupied by a crashed robot. $\square_{\text{Lemma 8}}$

The following theorem is a direct consequence from Lemma 8.

Theorem 5 *Algorithm 4 is collision free.*

Fault-tolerant Flocking.

Before we proceed, we state the following lemma, which gives a bound on the number of faulty robots under which a polygon can be formed.

Lemma 9 *Algorithm 4 allows correct robots to form an approximation of a regular polygon in finite time, and to maintain it in movement.*

PROOF. We first prove that correct robots form an approximation of a regular polygon in finite time. To do so, we show that each robot can reach within ϵ_r of its target in the formation $F(P_1, P_2, ..., P_n)$ in a finite number of steps. Assume that r_i is a correct robot in the system. If r_i is a leader, then P_1 is its current position, and by Algorithm 4, the target of r_i is a point within a circle or half circle, centered at r_i, and with radius less than or equal to ϵ_r satisfying Assumption 3, and excluding the positions of crashed robots. Since, there exists an infinite number of points within $Zone(r_i)$, and by Assumption 2, the cycle of a robot is finite, then r_i can reach its target within $Zone(r_i)$ in a finite number of steps.

Now, consider that r_i is a robot follower. We also show that r_i can reach within ϵ_r of its target P_i in a finite number of steps. We consider two cases:

- robot r_i can move freely toward its target P_i: every time r_i is activated, it can progress by at most $\epsilon_r/(k+1)(k+2)$. Since, the distance $dist(r_i, P_i)$ is finite, the bound k of the scheduler is also finite, and the cycle of a robot is finite by Assumption 2, then r_i can be within ϵ_r of P_i in a finite number of steps.
- robot r_i cannot move freely toward its target P_i: first, assume that r_i cannot progress toward its target P_i because of the restriction on the rank sequence $RankSequence$. Since, there exists at least one robot in $RankSequence$ that can move freely toward its target, and this is can be done in finite time as proved in the previous item, and also because the number of robots in $RankSequence$ is finite, and by Lemma 6, a robot can always move to a new location satisfying Assumption 3, then, eventually each robot r_i in $RankSequence$ can progress toward its target P_i and arrive within ϵ_r of it in a finite number of steps.

  Now, assume that r_i cannot progress toward its target P_i because it is blocked by some crashed robots. By Lemma 6, a robot can always move to a new location satisfying Assumption 3. Also, the number of crashed robots is finite, so eventually robot r_i can progress, and be within ϵ_r of its target in a finite number of steps, by similar arguments to those above.

We now show that correct robots maintain an approximation of the formation while moving. Since, all robots are restricted to move within one cycle by at most $\epsilon_r/(k+1)(k+2)$, then in every new k activations on the system, each correct robot r_i cannot go farther away than ϵ_r from its position during k activations. Consequently, r_i can always be within ϵ_r of its target P_i as in Definition 15, and the lemma follows. $\square_{\text{Lemma 9}}$

Theorem 6 *Algorithm 4 allows correct robots to dynamically form an approximation of a regular polygon, while avoiding collisions.*

PROOF. First, by Theorem 3, there is a finite time after which all correct robots agree on the same set of correct robots. Second, by Theorem 4, all correct robots agree on the total order of their ranking $RankSequence$. Third, By Theorem 5, there is no collision between any two robots in the system, including crashed and correct robots. Finally, by Lemma 9, all correct robots form an approximation of a regular polygon in finite time. $\square_{\text{Theorem 6}}$

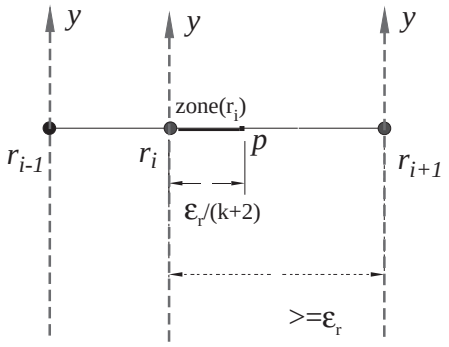
From Theorem 6, we infer the following theorem:

Theorem 7 *Algorithm 4 is a fault tolerant dynamic flocking algorithm that tolerates permanent crash failures of robots.*

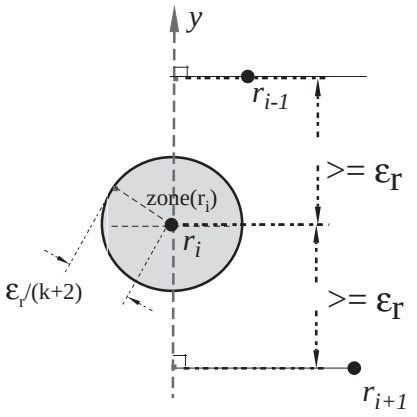
4.4 Discussion

In this chapter, we have proposed a fault-tolerant flocking algorithm that allows a group of asynchronous robots to self organize dynamically to form an approximation of a regular polygon, while maintain this formation in moving. The algorithm relies on the assumption that robots' activations follow a k-bounded asynchronous scheduler, and that robots have a limited memory of the past.

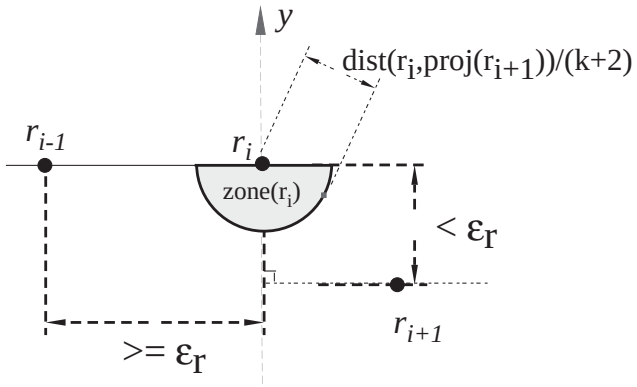
Our flocking algorithm allows correct robots to move in any direction, while keeping an approximation of the polygon. Unlike previous works (e.g., [GP04, CP07]), our algorithm can tolerate permanent crash failures of robots. The only drawback of our algorithm is the fact that it does not permit the rotation of the polygon by the robots, and this is due to the restrictions made on the algorithm in order to ensure the agreement on the ranking by robots.



(a) r_{i-1}, r_i, and r_{i+1} have the same y coordinate.



(b) r_{i-1}, r_i and r_{i+1} do not have the same y coordinate, and $dist(r_i, proj_{r_{i-1}}) \geq \epsilon_r$, and $dist(r_i, proj_{r_{i+1}}) \geq \epsilon_r$.



(c) r_{i-1} and r_i have the same y coordinate, however, r_{i+1} does not. Also, $dist(r_i, r_{i-1}) \geq \epsilon_r$, and $dist(r_i, proj_{r_{i+1}}) < \epsilon_r$

Figure 4.4: Zone of movement of a follower.

Chapter 5

Fault-tolerant Flocking of Mobile Robots with whole Formation Rotation

In the previous chapter, we proposed an asynchronous fault tolerant flocking algorithm. It makes robot formation move in lateral direction and forward/backward direction freely. However, as we analyzed, it could not make robot formation freely rotate. To lift such limitation, in this chapter we further explore a decentralized fault-tolerant flocking algorithm which can make robot formation freely rotation, yet in a weaker system model-SYm model [YSD09].

In detail, the proposed flocking algorithm uses the following two modules: a persistent rank assignment module, and a non-faulty robot selection module. Using rank assignment model, a unique rank is assigned for every robot and then the robots keep their ranks persistently. The failure detector module provides a method for robots to select the non-faulty robots based on a k-bounded scheduler. Finally, the correctness of the proposed algorithm are proved.

Before we introduce the algorithm, we first give the work environment of robots, i.e., *system model* as follows: 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 clockwise/counterclockwise. All robots share the same unit distance. The origin of the local coordinate system of a robot is fixed. The robots are not oblivious, that means, they has memory to remember their past information. Also, in our model, a robot can see all the other robots in the environment since the local view of robots make robot network disconnected easily.

5.1 Fault tolerant Flocking Algorithm

In this section, we give a decentralized fault tolerant flocking algorithm for robots. It can make sure all the correct robots form an approximate regular polygon and maintain it during moving. During the execution of flocking, two modules called failure detector and rank assignment are used. 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. First, we introduce the main contribution –flocking algorithm, where the rank assignment and failure detector are used as black boxes. Then, the two black boxes are described in the following two subsections.

5.1.1 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, and to reform a new regular polygon with the correct robots in the absence of crash of robots during flocking. More interesting point of our algorithm is that it can make robots rotate freely. The main idea of our algorithm is as follows:

1) Assign a unique persistent rank for each robot by rank assignment module;

2) Select the correct robots by failure detector module;

3) Based on the rank of robots, select a unique leader from the set of correct robots;

4) Based on positions of the leader and the other correct robots, a robot computes the target position and moves to satisfy the flocking requirements.

The specific algorithm is shown in Algorithm 7. Before we explain the detail of algorithm, the variables used in Algorithm 7 are shown as follows:

- r_i : the robot with rank number i;
- $S_{CurPosObser}$: the set of positions of robots at current activation;
- $S_{PrePosObser}$: the set of observed positions of robots at previous activation;
- $HistoryRankPos$: the rank and the set of all robot's positions during a robot's past k activations in a robot's memory, shown as (rank, the latest k positions);
- N_{act}: the number of activation of some robot;
- d: the expected edge length of the regular polygon, assumed $d \geq \xi$;
- $S_{correct}$: the set of positions of the correct robots;

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

- **Leader_Move** A global variable m is used for *leader* to control its moving speed per activation to wait for "lazy" follower robots. (Here, if the robots activates less often then the other robots, we called "lazy" robots.) The *leader* first observes the current robots' positions. If the positions satisfies Definition3, then the *leader* moves by not more than $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$ to any position p, where $p \notin HistoryRankPos$ [1]; otherwise, the leader will slow down by using the global variable m, and move by not more than $min(\frac{d}{2mk(k+1)}, \frac{\xi}{mnk})$ to any position p, where $p \notin HistoryRankPos$.
- **Follower_Move** First r_i computes its target position $Target(r_i)$ based on Function Formation and the codes in 6-8 in Algorithm 9. Then, it moves to a desired position p, which is not far away $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$ from the current position and $p \notin HistoryRankPos$.

A question may arise that: why is the speed of a robot's movement is not more than $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$? That is because: during a robot's k activations, the movement distance is not more than $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk}) * k = min(\frac{d}{2(k+1)}, \frac{\xi}{n})$. Thus, it can satisfy the requirement of Definition 3 and also can avoid collision among robots.

[1]The reason why the target position is different from the position in $HistoryRankPos$ is to satisfy the need of failure detector module.

Algorithm 7 Fault tolerant flocking (code executed by robot r_i)

Variables m: a global variable whose initial value is 1;
Input: $S_{CurPosObser}, N_{act}, HistoryRankPos, d, S_{PrePosObser}$

Upon activation {
  Call Persist_Rank($S_{CurPosObser}, N_{act}, HistoryRankPos, d$);
  $S_{correct}$:= Select_Correct_Robots($S_{PrePosObser}, S_{CurPosObser}$);
  $n = |S_{correct}|$;
  $leader$:= Robot with the smallest rank in $S_{correct}$;
if r_i is *leader* **then**
  leader_Move($d, \xi, HistoryRankPos, S_{correct}, m$);
else
  Follower_Move($d, \xi, HistoryRankPos, S_{correct}, leader$);
end if
}

Algorithm 8 Leader movement (code executed by the leader robot)

procedure leader_Move($d, \xi, HistoryRankPos, S_{correct}, m$)
  if $S_{correct}$ satisfies Definition 3 (see Section **??**) **then**
    Move by not more than $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$ to any position p, where $p \notin HistoryRankPos$;
    $m = 1$;
  else
    $m = m + 1$;
    Move by not more than $min(\frac{d}{2mk(k+1)}, \frac{\xi}{mnk})$ to any position p, where $p \notin HistoryRankPos$;
  end if
end

Algorithm 9 Follower movement (code executed by a follower robot r_i)

procedure Follower_Move($d, \xi, HistoryRankPos, S_{correct}, leader$)
  Variables: $S_{TargetPos}$: the set of target positions; $Target(r_i)$: the target position of r_i;
  Function: Formation($d, S_{correct}$, *leader*, $HistoryRankPos$):= the function to compute the target robot positions;

  $S_{TargetPos}$ = Formation($d, S_{correct}$, *leader*, $HistoryRankPos$);
    Sort the positions 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 j^{th} position of $S_{TargetPos}$ to the j^{th} robot of $S_{correct}$ ($1 \leq j \leq |S_{correct}|$);
  if $dist(r_i, Target(r_i)) > min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$ **then**
    Find position p on Segment $\overline{r_i Target(r_i)}$, which satisfies $dist(r_i, p) = min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$ (see Figure 5.1);
    $Target(r_i) := p$;
  end if
  if $Target(r_i) \in HistoryRankPos$ **then**
    Find a position p which satisfies $dist(p, Target(r_i)) < \frac{\xi}{nk}$ and $p \notin HistoryRankPos$;
    $Target(r_i) := p$;
  end if
  Move to $Target(r_i)$;
end

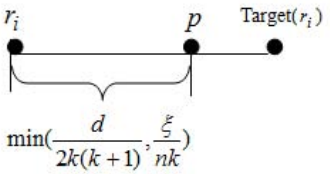


Figure 5.1: Finding a new target position p, where r_i is a robot's current position , $Target(r_i)$ is the target position.

5.1.2 Persistent Ranking for Robots

In this part, we provide a simple algorithm that gives a unique identification to each robot in the system, in order to select a unique leader in robot system. Our main idea is: 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 10, where procedure Assign_Rank is to assign rank 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 $S_{CurPosObser}$ by sensor. Then, it sorts the positions in $S_{CurPosObser}$ by decreasing order of y-coordinates, and puts the ordered positions into a variable named $RankSequence$; then it sorts the positions of robots with the same y-coordinate 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 $HistoryRankPos$.

- **Persist_Rank** First, by calling the failure detector module, the set of correct robots $S_{correct}$ can be gotten.

  referring to $HistoryRankPos$ and $S_{correct}$, a robot excludes the positions of crashed robots from the current position snapshot $S_{CurPosObser}$. Then, for every position p in $S_{CurPosObser}$, it can find its last recent position in $HistoryRankPos$ that is not far away from p by $min(\frac{d}{2(k+1)}, \frac{\xi}{n})$. This bound on distance is conservative, and it ensures no collision between robots. Thus, p's rank can be found by corresponding to p's last past position in $HistoryRankPos$.

Specially, during the first k activations, each activation a robot moves to its right hand along the perpendicular to y axis by $\frac{\xi}{nk}$. The moving speed is chosen as $\frac{\xi}{nk}$ 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 its rank is same in the memory of different robots.

Figure 5.2 and Figure 5.3 as examples illustrate how a robot assigns and tracks the rank of robots in the system, respectively. In Figure 5.2, 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 value. Based on procedure Assign_Rank in Algorithm 10, the ranks corresponds to the positions of four robots are: $(1, A), (2, B), (3, D)$, and $(4, C)$. These information are recorded in $HistoryRankPos$, i.e., $HistoryRankPos = \{(rank, position)\} = \{(1, A), (2, B), (3, D), (4, C)\}$. In Figure 5.3, all robots keep their ranks. The idea 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 $\acute{A}$'s searching disc. Then, by checking $HistoryRankPos$, it knows the information $(1, A)$. Therefore, a robot knows its rank is 1.

Algorithm 10 Persistent Rank $Persisting\_Rank$ (code executed by robot r_i)

Input: $S_{CurPosObser}, N_{act}, HistoryRankPos$

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

procedure Assign_Rank($S_{CurPosObser}$)
    $RankSequence$ = the positions in $S_{CurPosObser}$ 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 of each robot corresponds to its order in $RankSequence$;
    Save $RankSequence$ and the corresponding rank into $HistoryRankPos$;
end

procedure Persist_Rank($S_{CurPosObser}, S_{PrePosObser}$, d)
  $S_{correct} = SelectCorrectRobot(HistoryRankPos)$;
  $n = |S_{correct}|$;
  Referring to $S_{correct}$ and $HistoryRankPos$, exclude the positions of crashed robots from $S_{CurPosObser}$;
  for $\forall\, p \in S_{CurPosObser}$ **do**
    Find position q which satisfies $q \in S_{PrePosObser}$ and $dist(q,p) \leq min(\frac{d}{2(k+1)}, \frac{\xi}{n})$;
    Get p's rank that corresponds to q's;
    Save p's position into $HistoryRankPos$ and $S_{PrePosObser}$;
  end for
end

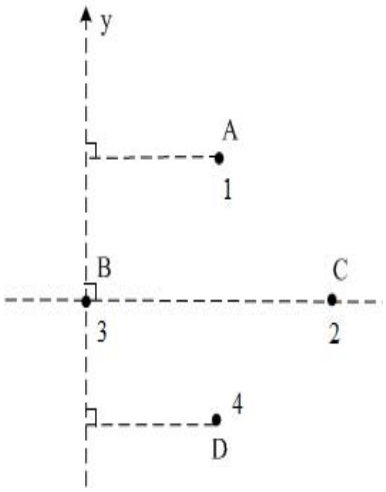


Figure 5.2: Assign_Rank during the first k activations: Initially, four robots are located at position A, B, C and D, where B and C has the same y coordinate. By Algorithm 1, their ranks are 1, 3, 2, and 4 which corresponds to positions A, B, C and D, respectively.

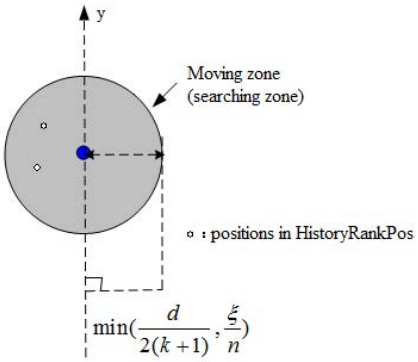


Figure 5.3: Robots keep their ranks in their searching zones (moving zone), whose radius is $min(\frac{d}{2(k+1)}, \frac{\xi}{n})$, excluding positions in $HistoryRankPos$.

5.1.3 Failure detector

In this section, we present a failure detector algorithm called $SelectCorrectRobot$ that provides a way to select the correct robots. 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 distinguish the other robots have crashed or not by comparing the position change of robots.

Each robot uses the same algorithm to select the non-faulty robots as shown in Algorithm 11. The variables used in Algorithm 2 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 $S_{CurPosObser}, S_{PrePosObser}, HistoryRankPos$. First, robot r_i compares the current position configuration $S_{CurPosObser}$ and the previous one $S_{PrePosObser}$. If there a position that in $S_{CurPosObser}$ and $S_{PrePosObser}$, then $Counter_i$ increases by 1; otherwise, $Counter_i$ becomes zero. When $Counter_i$ is larger than k, then the robot in that unchanged position will be regarded as a crashed robot based on k bounded scheduler and the movement rule in flocking. Finally, by corresponding the position and rank in $HistoryRankPos$, 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 $S_{correct}$.

Algorithm 11 Selection of Correct Robots

```
1: procedure Select_Correct_Robot (S_CurPosObser, S_PrePosObser, HistoryRankPos)
2:    S_temp := S_PrePosObser;
3:    S_correct := φ;
4:    for ∀p_i ∈ S_CurPosObser do
5:       if p_i ∈ S_PrePosObser then
6:          Counter_i = Counter_i + 1;
7:       else
8:          Counter_i = 0;
9:       end if
10:      if Counter_i > k then
11:         S_temp = S_temp − {p_i};
12:      end if
13:   end for
14:   for ∀p_i ∈ S_temp do
15:      S_correct = S_correct ⋃ {r_i | r_i is the rank of p_i};
16:   end for
17:   return (S_correct);
18: end
```

5.2 Correctness

5.2.1 Rank assignment

In the following, we will prove the correctness of Algorithm 10.

Lemma 10 *By Algorithm 10, 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, each activation, a robot moves to the right hand by not more than $\frac{\xi}{nk}$ along the perpendicular to the y-axis, which ensures that the y-values of the robots do not change during this period. Considering the extreme case: the robots on the right hand activate once during the period the left robots activate k times. At this time, the maximum distance that the left robots can move is $\frac{\xi}{nk} * k = \frac{\xi}{n} < d$. Therefore, these two robots do not swap positions. The same arguments apply 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.

$\square_{\text{Lemma 10}}$

A direct consequence from Lemma 1 can be deduced:

Lemma 11 *Algorithm 10 gives a unique rank to every robot in the system during the first k activations.*

Lemma 12 *By Algorithm 10 and Algorithm 7, the ranking is persistent during the entire execution of the algorithm.*

PROOF. Based on the rank persistent function, i.e., Function *PersistRank*, each robot matches its last position in the desired disc, which radius is $min(\frac{d}{2(k+1)}, \frac{\xi}{n})$ 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 is: only when some robots activate k times, a robot could activate 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 *PersistRank* 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 not more than $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$. Therefore, during its k activations, the maximum distance that the leader can move is not more than $min(\frac{d}{2km(k+1)}, \frac{\xi}{nmk}) * k = min(\frac{d}{2m(k+1)}, \frac{\xi}{nm}) \leq min(\frac{d}{2(k+1)}, \frac{\xi}{n})$ since $m \geq 1$.

Similarly for the followers, at each activation a follower moves by $min(d, min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})) \leq min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$. Therefore, during k activations, the maximum distance that a follower can move is also not more than $min(\frac{d}{2(k+1)}, \frac{\xi}{n})$.

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

In *PersistRank* function, a robot first finds the robots in the last activation who are within a distance of $min(\frac{d}{2(k+1)}, \frac{\xi}{n})$ 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. (2) When there is no robot crashed, every follower tries to move to its target position to form a desired regular polygon, i.e., tries to keep distance d with its neighbors. Between any two consecutive activations of a robot, the other robots can not be activated more than k times. That means, the maximal distance that a robot can move is not more than $min(\frac{d}{2(k+1)}, \frac{\xi}{n})$. Therefore, for any two correct robots, it is impossible to be closer than $min(\frac{d}{2(k+1)}, \frac{\xi}{n}) \leq min(\frac{d}{2}, \frac{\xi}{n}) \leq \frac{d}{2}$ since $n > 3$, $k > 2$, $\xi \leq d$, unless a robots has crashed.

$\square_{\text{Lemma 12}}$

From lemma 10 to lemma 12, we drive the following theorem.

Theorem 8 *Algorithm 1 gives a unique persistent rank for every robot in the system.*

5.2.2 Collision among robots

Lemma 13 *By Algorithm 10, there is no collision between any two robots in the system during their first k activations.*

PROOF. From the assumption, we know: The initial distance between any two robots is equal to d. During the first k activations, all robots just move to the right along y-axis by less than $\frac{\xi}{nk}$ each activation. Even if one robot is very active, the furthest distance that a robot can move is not more than $\frac{\xi}{nk} * k = \frac{\xi}{n} < d$. Therefore, there is no collision between any two robots during the first k activations by Algorithm 1, and the lemma holds. $\square_{\text{Lemma 13}}$

Lemma 14 *There is no collision among robots after the first k activations by Algorithm 7.*

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 among followers.
  - No collision between the leader and the followers: Initially the input of 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 $min(d_l, min(\frac{d}{2k(k+1)}, \frac{\xi}{nk}) * k \leq min(\frac{d}{2k(k+1)}, \frac{\xi}{nk}) * k = min(\frac{d}{2(k+1)}, \frac{\xi}{n}) < \frac{d}{2}$. Therefore, it is impossible for the leader and the followers to collide with each other.
  - No collision among followers: In the initial position configuration, the distance between neighbor followers is equal to d. Based on the target position computation method (see code in lines 12-14 in Follower Move function), each follower moves to its matched target position, and no two followers' movement paths cross each other. Therefore, for any two correct followers, there is no collision between them.
- 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 among 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 cannot move any more, and their positions are saved in $History$. By the algorithm, we know that the target positions of correct robots are different from the positions in $History$. Obviously, their target positions are different from those of the crashed robots. Therefore, it is impossible for the correct robots to collide with the crashed ones.

In the above two cases, there is no collision among robots, then we conclude that there is no collision after the first k activations based on our algorithm. $\square_{\text{Lemma 14}}$

5.2.3 Failure detector

The proposed non-faulty robot selection algorithm (Algorithm 11) satisfies the following properties of perfect failure detector [CT96]: *strong completeness*, *strong accuracy*. Furthermore, we get an additional property *eventual agreement*.

These properties are proved respectively, in Theorem 9, Theorem 10, and Theorem 11.

Theorem 9 ***(Strong completeness)****: eventually every robot that crashes is permanently suspected by every correct robot.*

Lemma 15 *If some robot r_i crashes at some time, then there is a time after which any correct robot detects the crash of robot r_i.*

PROOF. Let r_i be a crashed robot. Then, r_i will remain at its current position forever. Let r_f be a correct robot in the system. Robot r_f detects that r_i has crashed after its $(k+1)$ activations (activation of r_f). $\square_{\text{Lemma 15}}$

Theorem 10 ***(Strong accuracy)****: No robot is suspected before it crashes.*

Lemma 16 *By Algorithm 11, a faulty robot is never suspected by any other correct robot before it crashes.*

PROOF. This lemma is verified if Algorithm 11 can avoid the situation where robot r_i wrongly suspects robot r_i.

Take two robots r_i and r_j. Let r_j observe r_i. Robot r_i is activated at least once when r_j has been activated k times. From the *FaultTolerantFlocking* Algorithm, we know: when robot r_i is activated, it will move forward, and never go back to its and the other robots' past positions during its $(k+1)$ activations. We can determine that the position of r_i must change unless r_i has crashed. Thus, every faulty robot will not be suspected wrongly before it crashes by any other correct robots. □Lemma 16

From Theorem 9 and Theorem 10, we can conclude that Algorithm 11 satisfies the property of perfect failure detector [CT96] and also has the following property:

Theorem 11 ***(Eventual agreement)*** *Eventually all correct robots will have the same set of correct robots.*

5.2.4 Fault tolerant flocking

The following proof is for Algorithm 7.

Lemma 17 *Algorithm 7 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.

- **Induction basis**: Initially, a regular polygon with n robots is the input of flocking algorithm. When an activation number $N_{ack} \leq k$, every robot just adjusts its movement based on code in line 7 of Algorithm 7. It is not hard to see that during this period, $D(E, T_{P,E}) \leq \xi$.
- **Induction step**: Assume that before an activation N_{ack} ($N_{ack} \geq k$) of a robot, an approximate regular polygon is preserved. For any robot $r_i (0 < i \leq n)$, let $P_i^{N_{ack}}$ denote the current position of r_i and $T_i^{N_{ack}}$ denote the target position of r_i at activation N_{ack}. Then, $D(E, T_{P,E}) = \sum_{i=1}^{n} dist(P_i^{N_{ack}} - T_i^{N_{ack}}) \leq \xi$. For a robot r_i, we assume it is activated x_i times during this period. Then, it will move toward its target $T_i^{N_{ack}+x_i}$, which is at a distance of d_l from $T_i^{N_{ack}}$. That is, $dist(T_i^{N_{ack}} - T_i^{N_{ack}+x_i}) = d_l$.
  - **Leader.** During next k activations of a robot, the leader moves forward by $\dot{d}$, where $\dot{d} < min(\frac{d}{2(k+1)}, \frac{\xi}{n})$. During the period of two consecutive k activations, the leader moves forward by less than $\dot{d}$. Obviously, for a leader robot, $D(E, T_{P,E}) = 0$.
  - **Follower.** If r_i is a follower, then it will move based on the change of the position of the leader. If the leader don't move, the follower will find another position that is different from those in $HistoryRankPos$, but still keep the current formation. Once r_i finds the change of the leader's position, it will compute its new target position based on the leader's position. Then, it moves toward

its target $T_i^{N_{ack}+x_i}$, where x_i is the activation number of r_i after N_{ack}. Based on the code in lines 11-15 in Follower Movement algorithm, after r_i moves, the distance between its actual position and its target position is not more than $\frac{\xi}{nk} * k = \frac{\xi}{n}$. That is, $dist(P_i^{k+N_{ack}} - T_i^{N_{ack}+x_i}) \leq \frac{\xi}{n}$. Thus, we get $D(E, T_{P,E}) = \sum_{i=1}^{n} dist(P_i^{k+N_{ack}} - T_i^{N_{ack}+x_i}) \leq 0 + \sum_{i=1}^{n-1} \frac{\xi}{n} = \frac{\xi(n-1)}{n} < \xi$.

Here, because N_{ack} can be any natural number, a conclusion can be drawn that: all robots can keep an approximate regular polygon when no robots crash during flocking.

□Lemma 17

From Lemma 17, we derive the following lemma.

Lemma 18 *The flocking algorithm allows robots to make the formation move to any direction including rotation of the formation.*

Lemma 19 *When there are crashed robots during flocking, the remaining correct robots can be reformed into a regular polygon dynamically in finite time.*

PROOF. If the leader has crashed, then based on the code in line 6 of flocking module, a new leader is elected dynamically. Once a follower finds that the leader has crashed, a follower will follow a new leader. At the same time, if the leader finds the current correct robots can not form an approximate regular polygon, it will slow down based on the code in lines 8-9 in leader movement algorithm. For the correct followers, they will compute their target positions based on function *Target*. 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. □Lemma 19

5.3 Maneuverability and Bound Analysis

Based on the algorithm description and the correctness proof, we know the proposed algorithm lifts the limitation of formation rotation existed in the algorithm of the previous chapter. The reason why the proposed algorithm in this chapter can make formation rotation freely is that: in this algorithm, the robots are to keep their rank instead of keeping the relative positions among robots always.

To show the maneuverability of Algorithm 7 straightly, in this part we analyze the movement of the robots (formation) in detail. Also, we further explore the bound of robots movement to keep the formation, i.e., line speed per activation and angular speed per activation.

5.3.1 Maneuverability

From Algorithm 7, we know that: once the leader moves, the follower robots will follow the leader to move. 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 Figure 5.4: 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 formation (see Figure 5.5), formation move forward (see Figure 5.6), move in lateral direction and move in arbitrary direction. From Figure 5.5, we see: when the leader moves around a circle by clockwise, the followers compute their target position based on procedure

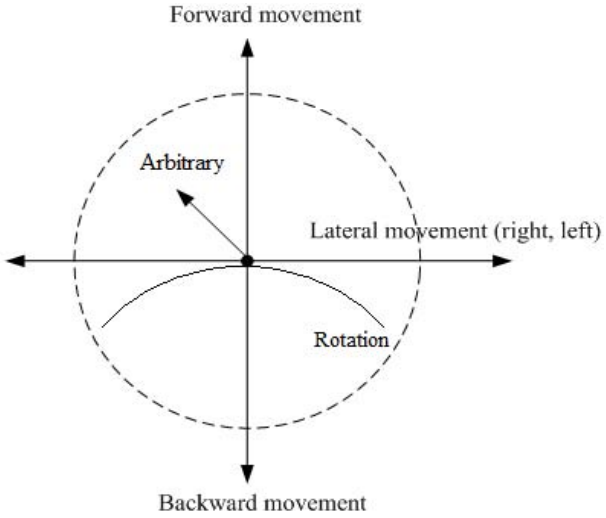


Figure 5.4: The possible leader movements: rotation, move forward, move backward, lateral movement (left, right) and random move.

Follower_Move. All followers rotate in the same way with the leader. At the same time, they can form an approximate regular polygon. For moving backward and forward, two cases are similar, so it is not necessary to give the formation backward movement. Same with lateral case, here we only show the formation changes when the leader moves to right. There is another intrusting movement–arbitrary move in the moving zone (searching zone). The arbitrary movement actually is a combination of line movement (formation move forward (backward) and lateral move) and rotation movement.

Remark: The difference between translation and rotation is that the formation of robots are moving along different routes. For the rotation, all robots move along a circle; but translation means all robots only move along y-axis and the perpendicular to y-axis.

5.3.2 Bound Analysis

In the following, we analyze the formation rotation of robots from the speed per activation and angular velocity per activation. From Algorithm 7, we can get the following properties: the maximum speed per activation of formation rotation is $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$ and the maximum angular velocity per activation is $min(\frac{\pi}{nk(k+1)}, \frac{2\pi\xi}{n^2kd})$, where n is the number of correct robots, k is from k bounded scheduler, d is the desired distance between neighbor robots, and ξ is from the Definition 3. That is because: when the distance per activation is equal to d, the angle that passes is $\frac{2\pi}{n}$ based on the definition of a regular polygon). When the distance that a robot passes per activation is $min(\frac{d}{2k(k+1)}, \frac{\xi}{nk})$, so we can get the angular velocity per activation by using $\frac{min(\frac{d}{2k(k+1)}, \frac{\xi}{nk}) * \frac{2\pi}{n}}{d} = min(\frac{\pi}{nk(k+1)}, \frac{2\pi\xi}{n^2kd})$. Thus, we get the following theorem:

Theorem 12 *When the robots flock, they satisfy the following conditions:*

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

From the analysis and the figures, we know the proposed flocking algorithm has good maneuverability. The limitation of formation movement in [DISC08] are lifted in this algorithm by keeping ranks of robots instead of keeping the relative positions among robots.

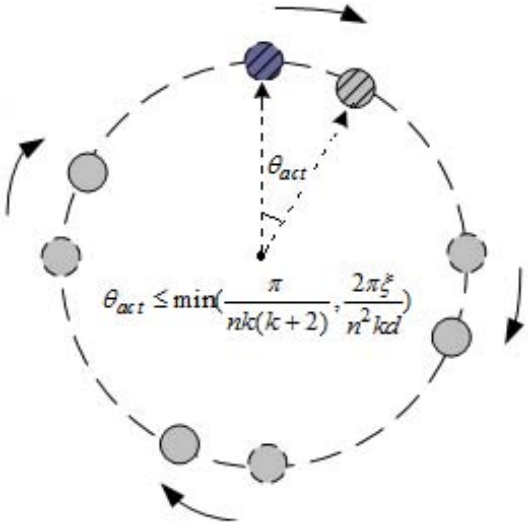


Figure 5.5: The formation rotates by clockwise: the circles with the broken line are the past positions of robots, and the circles with the solid line are the current (target) positions of robots.

5.4 Discussion

In this chapter, 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. Different from the work in previous chapter, the flocking algorithm in this chapter can make the shape rotate freely. Nevertheless, the maximal "speed"(expressed by distance per activation), and the rotation of the shape depend on the following parameters: the number of correct robots, the value of k in the bounded schedule, the desired edge length of the shape and the parameter in approximate shape ξ. In future work, we would like to investigate the speed (distance/activation) of the formation based on these different parameters.

One interesting question is: does the proposed flocking algorithm work correctly in CORDA if we don't consider the rotation? As we know, CORDA is totally asynchronous model; the robots can see the others that are moving. So, it makes the observation of robots are not accurate. Based on the inaccurate positions of the robots, the movement restriction should be more strict to make all robots keep the desired formation than that in this section. By comparing the movement restriction in these two models, you may find the difference.

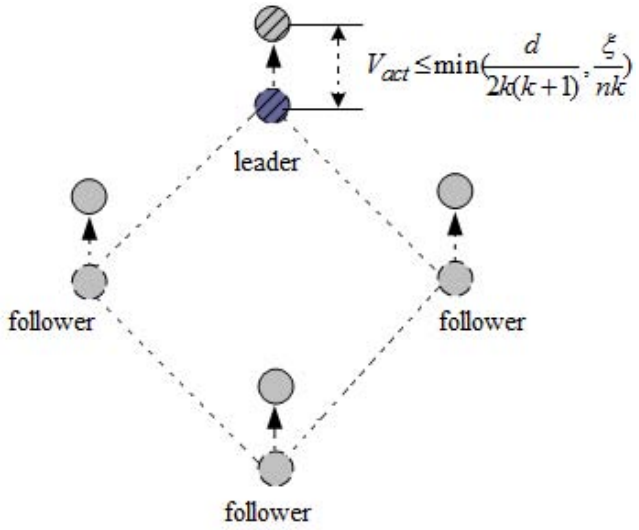


Figure 5.6: The formation moves forward: the circles with the broken line are the past positions of robots, and the circles with the solid line are the current (target) positions of robots.

Chapter 6

Flocking under Memory Corruption

In the previous two chapters (Chapter 4 and 5), we explored the fault tolerant flocking algorithms for a group of mobile robots in presence of faulty robots. In this chapter, we will discuss the same problem when robots' memory can become corrupted any time.

In the previous research, once the robots are faulty, they will stop moving and never recover. However, in practical applications, compared with the whole robot crash, some parts of a robot, like sensor, moving actuator, or memory etc., also have a high probability to fail. Thus, it becomes interesting to explore such kind of components failure or corruption. In particular, components such as memory, may be experienced bit flips due to external conditions. For example, memory may be influenced by some magnetic field, solar wind or some other kind of interferences. As a result, some data in memory may randomly change, e.g., the binary data "1" may become "0", or "0" may become "1". Thus, the data may be changed partly or totally. After the robots move away from the special place, the memory are not influenced any more.

To the best of our knowledge, few work that addressed on this issue in non-oblivious robots, especially for dynamic flocking. Therefore, in this chapter we discuss the memory corruption in the robots, and also address on the permanent faulty of the robots. The memory corruption is that the data in the memory may be changed partly or totally during unknown bounded period and after that the data is not changed by the environment.

The organization of this chapter is as follows: First, we analyze under which kind of system model (FSYNC, SYm and CORDA) the flocking algorithms in chapter 4 and 5 can work respectively; If the flocking algorithm cannot work in such model, we further discuss the reason why they cannot with memory corruption. If such algorithm cannot work in any system model, then we attempt to revise the flocking algorithm and make it work. In all, we try to find the weakest system model that a group of robot need to flock together with transient memory corruption.

6.1 System Model with Memory Corruption

In Chapter 2, we have presented the model of robots. Now we extend this model by allowing memory corruption of the robots. In our research, the robots are not oblivious, that is, they can remember their past position configuration in their memory. By this, they can distinguish if the other robots crash or not. In this work, we still consider the crash of robots are crashed and stop permanently. We don't consider the Byzantine behavior of robots, because the behavior of the robots are crazy.

The data in the memory may change randomly, for instance, past position of some robot changes to another location or nothing. Here, we assume that memory corruption only happened in "wait" state. In other words, during the period "look-compute-move", there is no corruption in memory. As a result, the observed position configuration each activation is correct. Only the data in memory may experience corrupt.

6.2 Problem Statement: Flocking in spite of crashes and memory corruption

From Chapter 2, we know the definition of position configuration. In this Chapter, we consider the memory corruption, as a result, the past position configuration will have the following two kinds of configuration: "clean configuration" and "dirty configuration".

Here, we assume there is a time t_{GST} called *Global Stabilized Time* (GST) after which the memory corruption stops forever. For instance, the robots move out of the magnetic field. After t_{GST}, the current position configuration observing by a robot is put into the memory and will not be changed by the outside environment. Recalling Chapter 2, robots may crash permanently.

Definition 17 (Clean configuration) *If all the data in the memory are correct, i.e., the data is not changed by outside environment, then we called such position configuration as clean configuration.*

Definition 18 (Dirty configuration) *If some of data in the memory are changed and different from the observed ones, then we called such position configuration as clean configuration.*

A distributed system that is self-stabilizing will end up in a correct state no matter what state it is initialized with, and no matter what execution steps it will take. This property guarantees that the system will end in a correct state after a finite number of execution steps. This is in contrast to typical fault-tolerance algorithms that guarantee that under all state transitions, the system will never deviate from a correct state. The ability to recover without external intervention is very desirable in robots system, since it would enable them to repair errors and return to normal operations on their own. The specific definition of self stabilization is as follows:

Definition 19 (Self-stabilization) *Starting from any arbitrary initial position configuration with n robots, any computation eventually reaches a legitimate configuration in finite activations after a finite time.*

Definition 20 (Legitimate configuration) *If a position configuration can satisfies the following properties, we called it legitimate configuration: (1) one unique leader (2) a consistent set of correct robots (3) all robots can re-form a desired formation in finite time.*

When memory is corrupted by the influence of outside environment temporarily, the robust flocking algorithm should satisfy the requirements of self-stabilizing into a legitimate position configuration in finite time.

Definition 21 (Self-stabilizing Flocking) *Let S a system of robots and P the flocking pattern. S verifies the flocking specification if and only if the robots satisfy the flocking pattern infinitely often.*

In detail, we need to find out if it is possible that a flocking algorithm self-stabilize to a legitimate state after t_{GST}; if it is possible, we further explore in what models; otherwise, we find out why it is not possible.

For any robot, in its memory, there are the history positions of all robots at least in the latest past k activations. The failure detector can use a robot's past positions to determine if it is crashed or not. For a robot r_i, we assume the set of all its past past positions in the memory be S^i_{pos}. After the memory get corrupted, the data in the memory has the following possibilities:

- Case 1: the values of data just exchange among them, but S^i_{pos} don't change;
- Case 2: the values of data becomes other values which are different from the data in S^i_{pos};
- Case 3: the values of data become same.

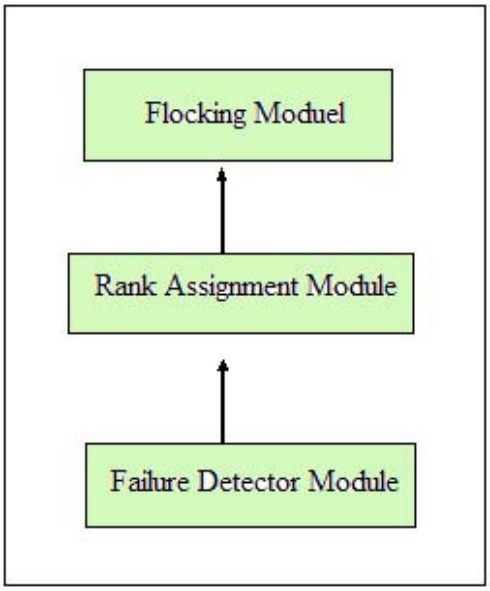


Figure 6.1: The modules in asynchronous flocking algorithm

- Case 4: part or total of the values are missing.

For Case 1 and Case 2, it is difficult to know if the memory has crashed, but the results in these two cases do not influence the output of failure detector. Therefore, we do not need to consider such corruption of memory.

For Case 3 and Case 4, it is very easy to detect the corruption of memory by comparing the data in the memory. Furthermore, in these two cases, the corrupted memory results in the wrong outputs of failure detectors. Consequently, the robots may not flock together.

6.3 $\mathcal{F}_T$ with Memory Corruption

6.3.1 Algorithm $\mathcal{F}_T$

Reminding the flocking algorithm $\mathcal{F}_T$ in Chapter 4, there are three modules, failure detector module, rank assignment module and flocking module shown in Figure 6.1.

The basic idea is: for every robot, first it calls *Failure detector* module to get the set of positions of correct robots, then it calls *Ranking algorithm* to re-assign rank for the correct robots based on y-value and clock/wiseclock direction. Finally, based on the positions of correct robots, it uses *Flocking algorithm* to achieve formation flock.

6.3.2 $\mathcal{F}_T^{MC}$ Self-stabilizes under the SYm Model

In the SYm model, we know the robots execute their activities of *observation*, *computation*, and *movement* in instantaneous (atomic) fashion and thus a robot observes other robots only when a cycle begins (i.e., when they are stationary.) Thus, once a robot activates, it can observe all the other robots correctly. Therefore, each activation the position configuration that a robot put into the memory is correct.

When the memory get corrupted, the data in Case 4, or Case 5 in section 6.2, the output of failure detector may be incorrect. Thus, a perfect failure detector becomes an unreliable one. (The failure detector may consider a correct robot as a fault one, or consider a fault robot as correct one.) Thus, the robots can not have the consistent position configuration about correct robots. Therefore, in flocking algorithm, more than one leader robot could coexist. Therefore, for $\mathcal{F}_T$ in Chapter 4, it is impossible

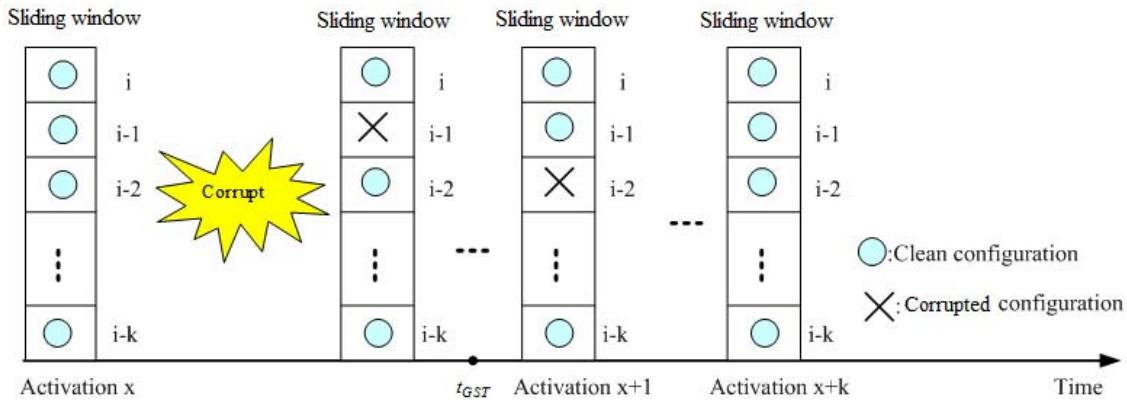


Figure 6.2: How the position configuration becomes clean for a robot? Where, n is the number of robots in the system.

for all the robots to achieve *eventual agreement* (See Theorem 3) and then to form a desired formation. Also, all robots can not flock together.

Even after t_{GST}, there are situations when the system never recovers. The data in memories of the robots will not be influenced any more. Each activation, a robot will get a new clean position configuration, and the dirty configuration will be removed from the memory gradually. Eventually all configurations in the memory will become clean. In other words, all the data are correct in the memory.

The failure detector can use these clean position configurations in the memory to select the set of positions of correct robots. Because the failure detector is perfect, all robots can get the same number of correct robots and the same set of positions of correct robots. After that, the rank assignment can re-assign rank for these correct robots. Using the rank of correct robots, the flocking module in Chapter 4, i.e., $\mathcal{F}_T$ can elect a unique leader and then flock together.

From the above analysis, we know the point is to make sure all the robots have the same position configurations and then have the consistent ranks; Thus, it makes possible to select a unique leader in flocking module.

Lemma 20 *$\mathcal{F}_T$ is self-stabilizing with respect to memory corruption and robot crash under the SYm model.*

PROOF.

For robot r_i, after time instant t_{GST}, when robot r_i activates once, one clean configuration is put into $HistoryMove$, and at the same time, a dirty configuration is pushed out of the slide window $HistoryMove$. Thus, for each robot, after at least m activations (m is the maximum length of the slide window $HistoryMove$), all the dirty configurations are cleaned. In our flocking algorithms of , we used k bounded scheduler, so the length of the slide window used can be equal to k.

Consider some arbitrary time interval $k \cdot n$ of consecutive activations. Let r_s denote the slowest robot in the sense that it is the robot that has the least number of activations during the interval. Let also r_f denote the fastest robot (i.e., with most activations).

Based on k-bounded scheduler, r_s will activate at least once when r_f activates k times. Thus, for all robots, after at most $k * n$ activations, all the position configurations in the memory become clean configurations. Here, we denote the time that all robots need to take to get stabilized be $t_{stabililzaed}$.

Then, the perfect failure detector module uses these clean configurations in the memory to select the set of positions of the correct robots. Since all the positions are correct, the output

of failure detector will always be correct using perfect failure detector. Thus, all the robots get the same set of the positions of the correct robots after $t_{GST} + t_{stabilized}$.

So, in rank assignment module, all robots can be re-assigned their ranks, and the ranks are consistent for the different robots. Because: (1) in SYm model, which is different from CORDA model, the moving of robots are instantaneous. That makes the observed positions of the robots are same for all the robots; (2) in $\mathcal{F}_T$ each activation the ranks of the robots are re-assigned based on y-value of robots and clockwise direction;

Thus, in flocking module, the unique leader can be chosen, as a result, the remaining robots can follow this leader to flock based on $\mathcal{F}_T$.

In all, under the SYm model, $\mathcal{F}_T$ can make robot system self-stabilizing to a legitimate configuration with respect to memory corruption and robot crash.

□Lemma 20

6.3.3 About $\mathcal{F}_T^{MC}$ in CORDA Model

From $\mathcal{F}_T$ in Chapter 4, we know: The robots need to control their movement and to make sure all robots have the same position configuration during their first k activations. That is, the relative position of robots can not be changed. After that, the relative position of robots are strictly controlled to make sure all robots have the same relative position configuration. Only by this way, flocking of a group of robots can be possible.

However, when the memory of robots are influenced, the output of failure detector may be not correct because failure detector seriously depends on the data in the memory. Also, for the different robots, they have different outputs. Thus, at this time, several leader robot may exist at the same time. Consequently, the relative positions of robots are broken and goes into a chaos state.

After t_{GST}, the memory will becomes clean, after each robot activates at least k times, where k is the value of k bounded scheduler. That means, that will need ($O(n\dot{k}^2)$) activations after all robots have been activated at least k times, and thus all the positions in memory are clean position configurations. Here, n is the number of robots in system.

Different from SYm model, CORDA model is totally asynchronous. A robot can observe the moving robot, which makes the observation of the positions of robots is not accurate. Thus, the relative positions of robots for different robots are different. Thus, the rank of robots are non-consistent. As a result, the leader is not unique. It becomes impossible too organize the robots to flock together. Therefore, $\mathcal{F}_T^{MC}$ is impossible to self-stabilize in CORDA model even after t_{GST}.

6.4 $\mathcal{F}_R$ with Memory Corruption

Differently from the modules in $\mathcal{F}_T(CORDA)$, the modules in $\mathcal{F}_{T+R}(SYm)$ are shown in Figure 6.3. First, all robots assign the consistent rank in the first k activations. After that, all robots persist their ranks by searching its past position based on the searching disc.

When the memory becomes corrupted, the past positions in the memory will become random, for example, Case 3 or 4 in Section 6.1. In Case 3, the robot may wrongly think a correct robot as crashed one; in Case 4, the robots may take mistake thinking a crashed robot as correct one. Thus, for different robots, they have different opinion and as a result they will move based on their own decision. Thus, the positions of the robots may become in disorder or they may collide each other.

Even after t_{GST}, using the searching method to find its past position still is unpractical, since there may exist more than one position in the searching disc. For example, after finite time, all the configurations in the memory become clean. But the current positions of robots still exist several positions in robot r_i searching area. Thus, r_i can not find which one is its past position and as a result it can not find

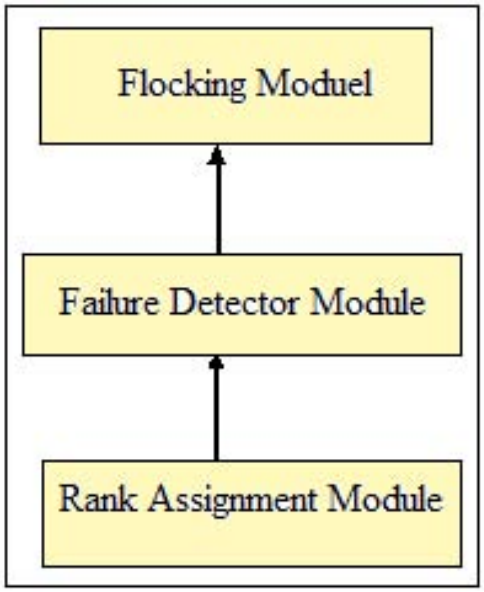


Figure 6.3: The modules of the flocking algorithm in SYM model.

its corresponding rank. Only in one case, the situation can be recovered that: in every robot's searching area, only one past position exists. Therefore, this dangerous situation are not always recoverable.

That is, $\exists r_i, r_j$, $dist(r_i, r_j) < R_{search}$, where R_{search} is the searching radius of a robot. From the method in Chapter 5, we know $R_{search} \leq min(\frac{d}{2(k+1)}, \frac{\xi}{n})$. In this case, the robots need to reset to an initial state and thus to satisfy the requirement that the relative positions of robots can not be changed. If they can not reset, the flocking algorithm becomes impossible under any system model, whatever CORDA, SYm or FSYNC.

Here, we attempt to revise the rank assignment algorithm shown in Algorithm 12. The procedure $Reset()$ can make sure that (1) the distance between any two robots be more than R_{search}; (2) the relative positions of robots are not changed. A question may arise that: why R_{search}? It is to make sure no collision among robots in their next activation. Obviously, the expected edge length d should be larger than R_{search}.

Algorithm 12 Persistent Rank $Persisting\_Rank$ for memory corruption(code executed by robot r_i)

1: **if** ($N_{act} \leq k$) **then**
2: First_AssignRank($S_{CurPosObser}$);
3: n= $|S_{CurPosObser}|$;
4: Move to its right hand by not more than $\frac{\xi}{nk}$ along the perpendicular to its y-axis;
5: **else**
6: Persist_Rank($S_{CurPosObser}, HistoryRankPos, d$);
7: **end if**

8: **if** ($\exists r_i, r_j, dist(r_i, r_j) < R_{search}$, r_i and r_j are neighbors) —— (formation don't satisfy an approximate formation) **then**
9: $Reset()$;
10: Assign_Rank($S_{CurPosObser}$);
11: **else**
12: Persist_Rank($S_{CurPosObser}, HistoryRankPos, d$);
13: **end if**

The requirement of Procedure $Reset$ is as follows: After resetting, the positions of the robots should satisfy: (1) the y-values of the robots are not changed; (2) the distance between any two robots is larger than R_{search}.

6.4.1 $\mathcal{F}_R^{MC}$ using Algorithm 12 works in FSYNC model

It is easy to prove that Algorithm 12 works under the FSYNC model. The function of procedure $Reset()$ is to ensure that the distance among robots is larger than R_{search}.

Lemma 21 *$\mathcal{F}_R$ using Algorithm 12 can self-stabilize in fully synchronous model when memory get corrupted.*

PROOF. In FSYNC model, all robots activates simultaneously. Thus, each activation all robots get the same position configuration.

Based on Algorithm $\mathcal{F}_R$, first a robot assigns their ranks using Algorithm 12. There are two procedures in Algorithm 12: Assign_Rank and Persist_Rank, which are same with the rank assignment module in $\mathcal{F}_R$.

If there are neighbor robots whose distance are smaller than R_{search}, or the formation of robots don't satisfy an approximate formation, then the robots will reset the positions of robots using the codes in line 9-10 in Algorithm 12. Then, a robot re-assign the ranks of robots. Now they all get the consistent ranks.

After t_{GST}, each robot needs at least k activations to make the position configurations in memory to be clean. Then, using the perfect failure detector, all robots get the same set of positions of correct robots. The following execution is same with that in $\mathcal{F}_R$.

In all, $\mathcal{F}_R$ using Algorithm 12 can self-stabilize in fully synchronous model even memory may get corrupted.

$\square_{\text{empty}}$

6.4.2 $\mathcal{F}_R^{MC}$ using Algorithm 12 cannot self-stabilize in the SYm model

When $\mathcal{F}_R$ uses Algorithm 12, we know that $Reset()$ is a very important procedure. But in SYm model, differently from FSYNC model, not all the robots activate at the same time. Thus, there is one case: when a fast robot executes $Reset()$, but a slow robot doesn't find any robot r_i, r_j who satisfy $dist(r_i, r_j) < R_{search}$, where r_i and r_j are the positions of two robots in the system. Therefore, the slow robot still executes *Persist_Rank* procedure to keep their ranks, while the fast robot has executed *Assign_Rank* to re-assign the ranks of the robots. Obviously, the ranks for the same robot may be different in eyes of these two robots. Thus, they may not get the same leader using the same flocking algorithm, and also can not flock together. Therefore, in the SYm model, $\mathcal{F}_R$ uses Algorithm 12 can not work when the memory of robots may corrupt.

One may ask: *Is there other modified algorithms that can work in SYm model?* For the question, we will leave it as an open question for our future work.

6.5 Discussion

In this chapter, we discuss the (im)possibility of self-stabilization of our fault tolerant flocking algorithms with memory corruption. In our fault tolerant flocking algorithms, the robots are non-oblivious. So, when the robots moves to some special environment, the memory of the robots may get corrupted temporarily.

From our analysis, we get the following conclusion that: for the transient failure occurred in robots, self-stabilization of robot flocking depends on the system model and design of algorithm .

Here, we only analyze the self-stabilization of our fault tolerant flocking algorithms. If our flocking algorithm can not work in some model, we analyze why it can not. However, we don't address if there is other algorithms that can work in such model. Because as we said before, it will open a new the design of flocking algorithm.

It would be interesting to apply into the system model with crash and recovery of memory if the appropriate change is made for the current flocking algorithms. Here, the crash and recovery of memory means that, in the sense, the memory is crashed and the data in the memory are totally lost; but after a finite time, it will get recovered and the new data can be saved into it.

In all, our analysis on memory corruption will inspire to consider the other transient failure of robot components and Byzantine failure model. Also, the self-stabilization of non-oblivious of robots also could apply to other robot applications, like gathering, or formation control.

Chapter 7

Adaptive Flocking Algorithm

In the previous chapters, we mainly focused on the fault tolerance of flocking, whatever robot crash or memory corruption. Also, we try to find as weak as possible capacity of a robot must have and the weakest model they could work. However, in previous chapters, there are no obstacles in the environment, so only the collision among robots is discussed. Differently from the previous work, here we mainly work on the collision avoidance between robots and between robots and obstacles in the environment.

The interesting is that: this proposed algorithm can achieve and maintain the desired distance from neighbors in absence of obstacles in the environments; in presence of obstacles, each robot uses the same algorithm to avoid the obstacles, also to avoid the other robots. Then, we made two simulations to evaluate the effectiveness of the proposed algorithm. The simulation results demonstrate that the proposed flocking algorithm can make a group of robots effectively adapt to a complex environment during flocking.

Specially, we consider the problem with the following *system model*: a system of anonymous mobile robots, which cannot be distinguished by appearance. Now there are assumptions that need to be made: All the robots can get information about their neighbors by local sensor capability, and they cannot communicate explicitly. Every robot is oblivious, that is, it can not remember its past information. The robots share the same coordinate system and unit distance. They execute the same algorithm, which takes the observed positions of the robots as input, and returns a destination for next step. The main assumption about the obstacles is that they are convex and compact sets.

7.1 Adaptive Flocking Algorithm

In this section, we introduce a novel and simple flocking algorithm, which can keep desired distance from neighbors when there are no obstacles, and also can avoid collisions between robots and (dynamic or static) obstacles when there are obstacles. Here, we assume the sense range of robots is larger than the desired distance d between neighbor robots, in other words, that a robot can use its sense ability, not communication ability, to achieve the desired distance from its neighbors. For simplicity, we assume all obstacles are closed discs, and the point of an obstacle that a robot can see and which is nearest to a robot is denoted by O_k, where k is a unique identity number of an obstacle. Let $\overrightarrow{F}$ denote the *flocking vector*, which is shared among robots.

In order to describe the proposed algorithm, the following notation is defined in Table 7.1:

Therefore, the set of neighbors for Robot i can be described as:

$$NB_i = \{R_j || q_i - q_j| < r, i, j = 1, 2, 3, ...\}, \tag{1}$$

where $|q_i - q_j|$ means the distance between q_i and q_j. Equation (1) means if robot R_j is in the sensor range of R_i, then R_j is a neighbor of R_i.

For Robot i, the movement vector from its neighbors $\overrightarrow{V_{nbs}^{i}}$ can be computed as follows:

Table 7.1: Notation of parameters

Notation	Description
R_i	the ID number of robot i;
N_i	the number of neighbors of Robot i;
$\overrightarrow{V_i}$	the movement vector of Robot i;
$\overrightarrow{V_{nbs}^i}$	the sum of movement vectors for all the neighbors of Robot i;
$\overrightarrow{V_{obstacles}^i}$	the sum of repulsive vectors from the obstacles of Robot i;
d	desired distance between robot neighbors;
O_k	the position in obstacle k that a robot can see, and which is nearest to a robot;
O_k	the position of obstacles;
l	safety margin of obstacles;
d_{min}^i	the minimum distance between robot R_i and nearby obstacles;
N_{iObs}	the set of obstacles of Robot i in its sensing range, i.e., $N_{iObs} = \{Obs_j \|\|\overrightarrow{q_i} - \overrightarrow{O_j}\| < r\}$, where $\overrightarrow{O_j}$ denotes the position of the obstacle Obs_j;
q_i	the position of Robot i;
$\theta(\overrightarrow{V_i}, \overrightarrow{V_j})$	the angle between the vectors $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$;
r	the interaction range for every robot, here $d < r$;
NB_i	the set of robot R_i's neighbors;

$$\overrightarrow{V_{nbs}^i} = \sum_{j \in NB_i} \frac{q_j - q_i}{|q_j - q_i|} \frac{(|q_j - q_i| - d)}{N_i + 1}, \tag{2}$$

Equation (2) describes a robot trying to achieve the desired distance from its neighbors. In this equation, $\frac{q_j - q_i}{|q_j - q_i|}$ shows the unit vector whose direction is from q_i to q_j, $\overrightarrow{R_i R_j}$. And $(|q_j - q_i| - d)$ is the distance deviation between the actual distance and the desired distance between two robots. Here, it is interesting to explain why the distance deviation is divided by 2 in Equation (2), i.e., $\frac{(|q_j - q_i| - d)}{2}$ as a robot's desired movement distance. That is because the algorithm is distributed, and every robot moves to the target using the same algorithm. For example, if the desired distance between two robots is 2m, and now the actual distance between them is 4m, in order to achieve the desired distance between them, a robot only needs to move towards the other by 1 m, i.e., $\frac{(|q_j - q_i| - d)}{N_i + 1} = \frac{4m - 2m}{1+1} = 1\ m$. When the number of a robot's neighbors is larger than one, the sum of movement vectors between the robot and its neighbors will be computed. Obviously, when the distance between two robots is larger than the desired distance d, the robots will move toward each other; otherwise, when the distance between them is smaller than d, they will move away from each other until they achieve the desired distance.

When there is no obstacle in the sensor range of a robot, a robot first computes the movement vector $\overrightarrow{V_i} = \overrightarrow{V_{nbs}^i} + \overrightarrow{F}$. Also, in order to avoid collision with obstacles or robots which are beyond the sensor range of a robot, the size of vector $\overrightarrow{V_i}$ can not be larger than the sensor range r, i.e., when $|\overrightarrow{V_i}| > r$, $\overrightarrow{V_i} = \overrightarrow{V_i}/|\overrightarrow{V_i}| * r$.

When there are obstacles in the environment, in order to avoid the obstacles, a robot will adjust the movement vector based on Equation (2). We assume obstacles have the safety margin l, if the distance between a robot and obstacles is larger than l, then there is no force on a robot; otherwise there will be a repulsive force on a robot in order to avoid collision between robots and obstacles.

$$\overrightarrow{V_{obs}^i} = \sum_{k \in N_{iObs}} \frac{O_k - q_i}{|O_k - q_i|} Sat\{l - |O_k - q_i|\}, \tag{3}$$

where

$$Sat\{x\} = max(0, x). \tag{4}$$

Furthermore, we can find that: the closer a robot is to an obstacle (i.e.,$|O_k - q_i|$ is smaller), the larger the repulsive force will become ($l - |O_k - q_i|$ becomes larger). Therefore, it is an effective method to protect the robot itself from colliding with an obstacle. In addition, when $|V^i_{obstacle}| < |\overrightarrow{V_i}|$, the coefficient α is computed by $|\overrightarrow{V_i}|/|V^i_{obstacle}|$ in order to make sure there is no collision between robots and obstacles; otherwise, when $|V^i_{obstacle}| > |\overrightarrow{V_i}|$, the value of α is 1. If there is no parameter α to balance the vectors of $\overrightarrow{V_i}$ and $V^i_{obstacle}$, the collision may happen when $|\overrightarrow{V_i}| > |V^i_{obstacle}|$ when the direction of $\overrightarrow{V_i}$ and $\frac{O_k - q_i}{|O_k - q_i|}|$ are opposite. Therefore, the parameter α is a very important factor, and it can make the algorithm more robust.

The specific algorithm for obstacle avoidance flocking is described in Algorithm 13. When the size of $\overrightarrow{V_i}$, i.e., $|\overrightarrow{V_i}|$ is larger than the sensor range of a robot, the vector $\overrightarrow{V_i}$ needs to be adjusted using Equation (5) in order to avoid any collision with anything that is beyond a robot's sensor range.

$$\overrightarrow{V_i} = \overrightarrow{V_i}/|\overrightarrow{V_i}| * r; \tag{5}$$

When there are obstacles and neighbors in a robot's sense range simultaneously, the movement for a robot is determined by three factors: the positions of its neighbors, the positions of obstacles, and the size of the flocking vector $\overrightarrow{F}$. First, a robot computes the movement vector $V^i_{obsacles}$ using Equation (4). Also, it computes the nearest distance d^i_{min} from the nearby obstacles. And then, the smaller one between $|\overrightarrow{V_i} + \overrightarrow{V^i_{obs}}|$ and d^i_{min} is chosen as the value of d_t. Finally, the target movement vector is computed by

$$\overrightarrow{V_i} = (\overrightarrow{V_i} + \overrightarrow{V^i_{obs}}) * d_t; \tag{6}$$

Thus, in order to avoid the obstacles, the distance that a robot moves is not more than d^i_{min}. That is why a robot chooses the smaller one between d^i_{min} and $|\overrightarrow{V_i} + \overrightarrow{V^i_{obs}}|$ as the movement distance. The direction of the final movement vector is same as that of the vector $\overrightarrow{V_i} + \overrightarrow{V^i_{obs}}$ and the distance that R_i can move is not more than d^i_{min}. Therefore, our contribution is that we not only consider collision avoidance among robots, but also consider collision avoidance between robots and (static or dynamic) obstacles at the same time.

Algorithm 13 Adaptive Flocking for every robot.

1: $\overrightarrow{V^i_{nbs}} = \sum_{j \in NB_i} \frac{q_i - q_j}{|q_i - q_j|} \frac{(|q_i - q_j| - d)}{N_i + 1}$. {*Compute the movement vector from neighbors*}
2: $\overrightarrow{V_i} = \overrightarrow{V^i_{nbs}} + \overrightarrow{F}$;
{*Compute the sum of two movement vectors, $\overrightarrow{V^i_{nbs}}$ and flocking vector $\overrightarrow{F}$*}
3: **if** $|\overrightarrow{V_i}| > r$ **then**
4: $\overrightarrow{V_i} = \overrightarrow{V_i}/|\overrightarrow{V_i}| * r$;
5: **end if**
6: **if** there exist obstacles **then**
7: $d^i_{min} = min(|q_i - O_1|, ...|q_i - O_k|)$, where $k \in N_{iObs}$;
{*Compute the minimum distance between robot R_i and the obstacles around it*}
8: $\overrightarrow{V^i_{obs}} = \sum_{k \in N_{iObs}} \frac{q_i - O_k}{|q_i - O_k|} Sat\{l - |q_i - O_k|\}$, where $Sat\{x\} = \begin{cases} 0, x \leq 0; \\ x, x > 0. \end{cases}$
9: $d_t = min(|\overrightarrow{V_i} + V^i_{obstacle}|, d^i_{min})$;
10: $\overrightarrow{V_i} = (\overrightarrow{V_i} + \overrightarrow{V^i_{obs}})/|\overrightarrow{V_i} + V^i_{obstaces}| * d_t$;
11: **end if**

7.2 Correctness Analysis

Lemma 22 *If there are no obstacles at the current time, Equation (2),*

$$\overrightarrow{V_{nbs}^{i}} = \sum_{j \in NB_i} \frac{q_i - q_j}{|q_j - q_i|} \frac{(|q_j - q_i| - d)}{N_i + 1},$$

ensures that robots achieve the desired distance d from their neighbors.

PROOF.

(1) $NB_i = 0$. A robot has no neighbor in its sense range, it just moves with movement vector $\overrightarrow{F}$.

(2) $NB_i = 1$. Robot R_i has one neighbor, and we assume that neighbor is robot R_j. The vector $\overrightarrow{V_{nbs}^{i}}$ for Robot i can be computed as follows:

$$\overrightarrow{V_{nbs}^{i}} = \frac{q_i - q_j}{|q_j - q_i|}(\frac{|q_j - q_i| - d}{2}).$$

Because the algorithm is distributed and robots are synchronous,the movement vector $V_{neighbors}^{j}$ for robot R_j is

$$V_{neighbors}^{j} = \frac{q_j - q_i}{|q_i - q_j|}(\frac{|q_i - q_j| - d}{2}).$$

If the current positions of Robot i and Robot j are described as $q_i(x_i, y_i)$ and $q_j(x_j, y_j)$, the current actual distance d_{act} between two robots can be described as $|q_i - q_j|$ when the robots can see each other. After Robot R_i moves with $\overrightarrow{V_{nbs}^{i}}$, the new position of Robot R_i will become

$$q_i' : (x_i + \frac{x_j - x_i}{d_{act}} \frac{d_{act} - d}{2}, y_i + \frac{y_j - y_i}{d_{act}} \frac{d_{act} - d}{2}). \quad (7)$$

The new position of Robot R_j is

$$q_j' : (x_j + \frac{x_i - x_j}{d_{act}} \frac{d_{act} - d}{2}, y_j + \frac{y_i - y_j}{d_{act}} \frac{d_{act} - d}{2}). \quad (8)$$

Therefore, the new distance between the new positions of Robot i and Robot j is:
$|q_i' - q_j'| =$

$$\sqrt{[(x_i - x_j) + \frac{(x_j - x_i)(d_{act} - d)}{d_{act}}]^2 + [(y_i - y_j) + \frac{(d_{act} - d)(y_j - y_i)}{d_{act}}]^2}$$

$= \sqrt{(x_i - x_j)^2 \frac{d^2}{d_{act}^2} + (y_i - y_j)^2 \frac{d^2}{d_{act}^2}}$
$= \frac{d}{d_{act}} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
$= \frac{d}{d_{act}} \cdot d_{act}$
$= d.$

Therefore, from the above analysis, we can see that two robots can achieve the desired distance based on Equation (3).

(3) $NB_i > 1$. The robot R_i has more than one neighbor in its sensor range. At this time, the movement of robot R_i for the next step will consider all its neighbors, and move with the sum of the vectors between neighbors. In other words, robot R_i will move to the average position of its neighbors and try to keep the desired distance with its neighbors. Eventually, the target of the desired distance from its neighbors will be achieved in finite time.

□Lemma 22

Lemma 23 *This flocking algorithm can enable robots to avoid collision with obstacles in the environment.*

PROOF.

When there are neighbors and obstacles in the interaction range of robot R_i, the final movement vector is determined by Equations (2), (3), (5) and the size of flocking vector $\overrightarrow{F}$.

First, if the size of $\overrightarrow{V_i}$, i.e., $|\overrightarrow{F} + \overrightarrow{V_{nbs}^i}|$ is larger than the sensor range r of a robot, the direction is not changed but $|\overrightarrow{V_i}|$ is adjusted to be equal $|r|$. Thus, the robot avoids the objects beyond its sensor range.

Second, the minimum distance d_{min}^i between the robot and the nearby obstacles is computed. And then, if there are many obstacles around R_i, Equation (3) is used to compute the sum of the repulsive force for R_i. The direction is along that of the vector $\overrightarrow{V_i} + \overrightarrow{V_{obs}^i}$ and the distance that R_i can move is not more than d_{min}^i. There is a critical case may happen if the robot R_i is close to its nearest obstacles but doesn't collide with it. Thus, R_i cannot collide with any obstacles.

□Lemma 23

7.3 Performance Illustration

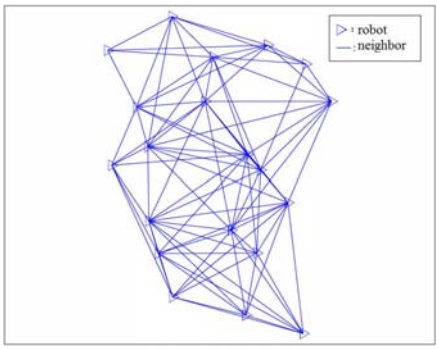
In this section, we would like to illustrate the algorithm execution by considering a few examples. During this illustration, we would like to know how a group of robots move together to avoid the collision between them, and between robots and obstacles. All the robots coordinate in distributed way.

There are two kinds of situations that need to illustrate: (1) when the obstacles are absent in the environment, how robots flock to avoid collision between robots? (2) when obstacles are present in the environments, how robots flock to avoid the collision between robots and between robots and obstacles.

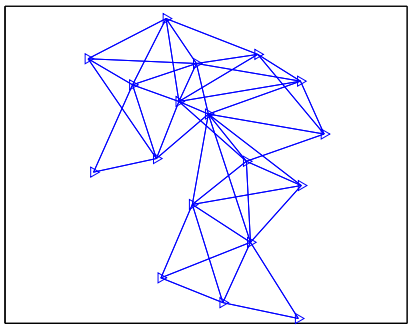
7.3.1 Simulation Setting

The specific simulation settings for the simulations are as follows: the number of robots is 20. The sensor range of every robot is 2.0m. Initially, the robots are randomly positioned in different places. The flocking movement vector $\overrightarrow{F}$ is [0.1; 0], which means, the direction of the whole flock is along the x-axis. The desired distance between robots is 1.5m.

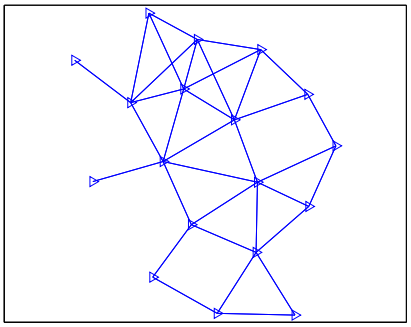
In the following, we run two simulations to evaluate the performance of the proposed algorithm: one is without obstacles and another one with obstacles in the environment. For the first case, we mainly consider the desired distance between robots; for the second case, we mainly evaluate two aspects of this algorithm: collision between robots and collision between robots and obstacles.



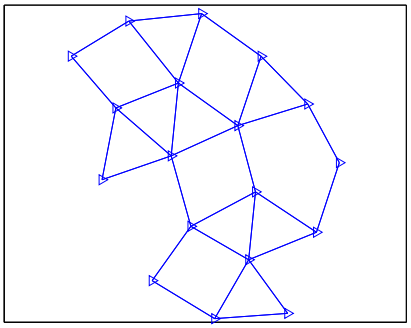
(a) Initial neighbor network of robots.



(b) Neighbor network at 0.01 s.



(c) Neighbor network at 0.03 s.



(d) Neighbor network at 0.05 s.

Figure 7.1: Dynamic change of the robot neighbor network without obstacles using the proposed algorithm: each of them keep the desired distance with their neighbors.

7.3.2 Simulation 1: without obstacles in the environment

In this simulation, there is no obstacle in the environment. The specific results are shown in Figure 7.1. In this figure, the triangles are robots, and the line between robots means robots are neighbors of each other, that is, two robots are neighbors if they can see each other. There are four sub-figures in Figure 7.1. The sub-figure 7.1(a) shows the initial positions and neighboring relationships of all robots. As we seen, the neighbor network is very dense. A robot may has many neighbors in its sensor range. Using the proposed algorithm, the distance between robots is adjusted and the neighborhood network becomes sparser after $0.01\ s$ (see sub-figure 7.1(b)). With time passing, at time slot $0.03\ s$, the distances between robots are mostly close to the desired distance shown in sub-figure 7.1(c). Finally, about at $0.05\ s$, all robots achieve the desired distance from their neighbors.

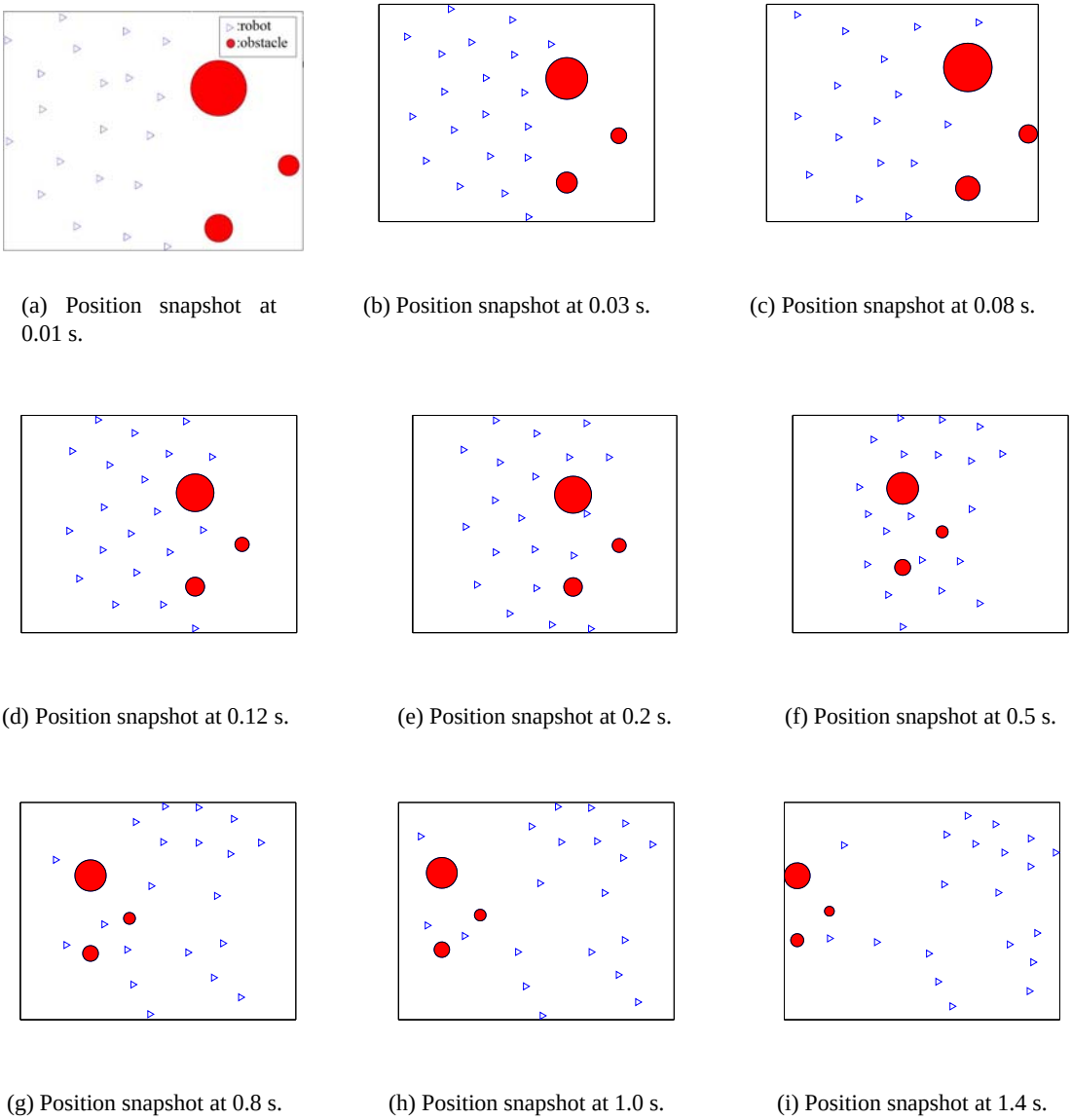
From the results shown in Figure 7.1, we can see: when there are no obstacles in the environment and each robot has at least one neighbor, all robots achieve the desired distance with their neighbors during flocking.

7.3.3 Simulation 2: with obstacles in the environment

Unlike Simulation 1, in this simulation there are 3 obstacles in the environment. Therefore, the movement of a robot is determined by three forces, from its neighbors, from obstacles (if there is an obstacle), and

from the flocking vector. The simulation results are shown in Figure 7.2. Here, every sub-figure in Figure 7.2 is a snapshot of 20 robots at a specific time. The sub-figure 7.2(a) is the initial position of robots. And at time 0.03t, all robots try to achieve the desired distance from their neighbors and move forward. With time passing, the robots adjust their movement vectors to avoid collision with neighbors and with obstacles. Finally, all robots avoid collision.

The detailed information about how a flock of robots move during flocking is shown in Figure 7.3. Specially, we analyze the following four typical track lines: Line 1, Line 2, Line 3 and Line 4. The robot that moved along Line 1 avoided collision with a big obstacle. The robot that moved along Line 2 simultaneously avoided two obstacles, the bigger one and the smaller one. The robot with the movement track of Line 3 adapted to avoid the second-biggest obstacle and the smallest one. The fourth case, the robot with the track of Line 4 avoided the second-biggest obstacles. In all, despite the fact that different robots may choose different routes to flock, they all can effectively avoid collision with obstacles.



(a) Position snapshot at 0.01 s.

(b) Position snapshot at 0.03 s.

(c) Position snapshot at 0.08 s.

(d) Position snapshot at 0.12 s.

(e) Position snapshot at 0.2 s.

(f) Position snapshot at 0.5 s.

(g) Position snapshot at 0.8 s.

(h) Position snapshot at 1.0 s.

(i) Position snapshot at 1.4 s.

Figure 7.2: Robots move together to avoid obstacles between robots and between robots and obstacles.

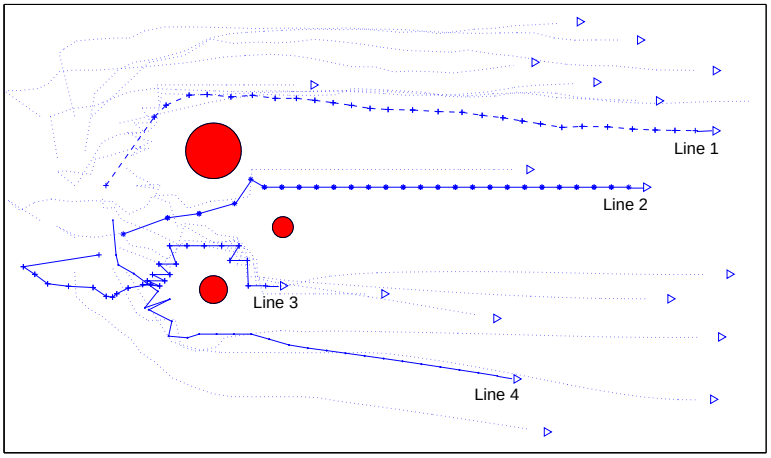


Figure 7.3: The movement tracks of all robots during flocking.

7.4 Discussion

This paper proposed a novel and simple flocking algorithm, which can make robots achieve and keep desired distance from neighbors when there are no obstacles, also can avoid collisions between robots and their neighbors and between robots and obstacles when there are obstacles in the environment. By algorithm correctness analysis and simulation results, we find this algorithm is effective and correct, and it would be very useful in practical applications, like rescue after earthquake, and space exploration etc.

Due to the different choice of routes and a robot's local sensor capability, the times that robots spent to pass by these obstacles are different. Eventually, a group of robots may be split into several groups. It is necessary to address this problem in future work, for example, to design a flexible flocking movement vector, or add some robots who have longer sensor capability into a flock of robots, to make all robots move together.

From the above analysis, we know due to the robot's limited sensor capability, the connection between robots becomes very weak. To find an appropriate distance between robot neighbors becomes important relatively. The distance between robot neighbors can not be too near to avoid collision, and also can not be too large to avoid losing connection. That is why in formation flocking part, we assume the robots has unlimited sensor ability. In future, we will further explore the fault tolerance of flocking of the robots with limited sensor ability.

Chapter 8

Conclusion

8.1 Research Assessment

In this research, we work on the flocking problems of a group of mobile robots. During flocking, all robots use the same algorithm to coordinate each other, for instance, formation maintenance, collision avoidance. Contrary to the existed work, our flocking algorithms could work in presence of robot failure; also in transient failure of some component, they could self-stabilize under appropriate system model.

The main originality is that we build bridge gap between several communities–distributed systems (fault tolerant algorithms), self-stabilization and mobile robotics. In fault tolerant distributed systems, all processes use the same algorithm to cooperate each other and they ensure the whole system reliable and dependable. That means, even in presence of process failure the remaining correct processes still can make sure the whole system work well. Self-Stabilization is the property of an autonomous process to obtain correct behavior no matter what initial state is given. Thus, self-stabilization automatically corrects following arbitrary transient faults that corrupt the state (so long as the program's code is still intact). In mobile robotics, the effective cooperation between robots is the focus. The similarity between distributed systems and mobile robotics is that they both address a group of processes (robots) which work in distributed way. The similarity of fault tolerant algorithms and self-stabilization is that they both can make sure the whole system dependable, but the difference is that self-stabilization focuses on transient failure but in distributed systems the failure can be transient or permanent.

In all, this research has led to four major contributions. The first major contribution is fault tolerant flocking in a k-bounded asynchronous robot system. All robots can form a regular polygon and keep it while moving, but the formation generated by the robots can not freely rotate. In the second important contribution, we lift such limitation yet in a weaker system model. The third contribution is the discussion about self-stabilization of robot system when memory may corrupt. The last major contribution is the distributed flocking algorithm to avoid collision among robots and obstacles.

Fault-tolerant flocking in a k-bounded asynchronous system This dissertation formally presents the definition of fault tolerant flocking, which builds a bridge between robots application (flocking) and dependable distributed system. After formally specifying flocking problems and developing an adequate computational model, we find the appropriate and as weak as possible conditions under which flocking can be achieved.

The main contribution of fault tolerant flocking in k-bounded robot system is to show that the geometric formation agreement during dynamic flocking. Specially, all robots can coordinate effectively each other to generate a desired formation even in presence of failure of robots.

Other contributions can be mentioned. First, we propose the algorithm modules, which is convenient for hardware design and using in practical applications. Second, we propose a method how to distinguish the crashed robots from the correct ones. Using this method, the robot system can break the deadlock that the correct robots wait for the crashed robots for ever and don't move any more. In all, they provide

a dependable platform for dynamic flocking of robots.

Fault-tolerant Flocking with whole Formation Rotation Despite the above flocking algorithm works well in asynchronous model, it does not allow the rotation of formation by the robots freely. To lift such movement limitation, we solve this problem yet in weaker (semi-synchronous) model.

Concretely, the main contribution is that the proposed flocking algorithm has good maneuverability. Also it can tolerate the permanent crash of the robots and can maintain the desired formation by the robots when there is no new crash.

Remark: In the proposed fault tolerant flocking algorithms, some modules like rank assignment and failure detector can be used in other robot coordination, such as gathering.

Flocking with memory corruption In practical applications, some parts of a robot, like sensor, moving actuator, or memory etc., is prone to crash due to the influence of complex environment. It is interesting that sometimes part of components, like memory of robots, may corrupt during some special area (e.g., magnetic field). After the robots move away from such special area, the memory is not influenced any more.

The main contribution is the discussion of the (im)possibility of self-stabilization of flocking algorithms with memory corruption. From the analysis result, we find: for the transient failure occurred in robots, the self-stabilization of robot flocking depends on the system model and algorithm design.

Decentralized Adaptive Flocking Algorithm There are many engineering points to solve flocking issue. In order to actually use the solutions, we present a distributed flocking algorithm to effectively avoid collision between robots, as well as the collision between robots and obstacles in presence of obstacles in the environment. More interesting is that in absence of obstacles, a robot can keep the desired distance with all its neighbors.

In this flocking algorithm, we mainly focus on the collision avoidance, not on fault tolerance. Putting it into this thesis is to compare the non-fault-tolerant flocking with fault-tolerant flocking.

8.2 Open Questions & Future Research Directions

"Research", as its names means, "re-search". It is a procedure to re-search interesting questions and the corresponding solutions. As mentioned in the literature, finding the answers to important questions usually leads to asking further questions. Now, our work also opens several new interesting questions and we present some of the related open questions and future directions.

Possibility of shape rotation in other models A shape rotation flocking presented in Chapter 5 considers a semi-synchronous model called SYm [SY99] with crash failure of robots, in which robots can crash but never recover. We know the SYm is an semi-synchronous model, in which robots execute their activations in atomic way.

An interesting question is whether it is possible to find shape rotation solution in more complex models: CORDA [GPRE01b] model where robots are totally asynchronous, systems where robots can crash and recover, or systems with Byzantine robots. In detail, Byzantine failure is more general but more complex than crash stop failure. In such case, when robots experience Byzantine failure, all robots may get crazy and move to wherever it wants. The other correct robots may get crash by colliding with such crashed robots. Therefore, it would be a big challenge to make all robots work coordinately with Byzantine failure.

The CORDA model is weaker than SYm, that means, if one flocking algorithm can in CORDA model, it implies that it also can work in SYm model. That makes the algorithm more general and have wider applications.

Flocking with other schedulers In our work, the k bounded scheduler is used in the fault tolerant flocking algorithms to make sure the fairness of the activations between robots in Chapter 4 and 5. The value of k is known for every robot. One question may arise that: What if the robots don't know the value of k ?

Therefore, in future, it is interesting to consider the fair version of the following schedulers in flocking application of robots: *unbounded* scheduler, *bounded* regular scheduler, *centralized* scheduler, *arbitrary* scheduler [DGMP06], etc.. Except *centralized* scheduler, the other schedulers are weak than k bounded scheduler. It would be interesting to discuss the (im)possibility using such scheduler and thus to find the weakest scheduler that robots coordination needs. It provides the way to find the weakest model that the robots need to coordinate each other.

Flocking with other failure detectors To design a fault tolerant algorithm, one important (core) question is to find a failure detect to distinguish the crashed processes and the correct ones. The fault tolerant robot research is no exception. In the presented fault tolerant flocking algorithms, a perfect failure detector is used by managing the moving of robots strictly.

As we know, the perfect failure detector is very strongest among eight failure detectors. We could call perfect failure detector be reliable failure detector, since it always provides the correct results. One interesting questions is: what if using the unreliable failure detector, like S failure detector or $\diamondsuit S$ failure detector [CT96].

Also, in our work, the robot needs very strict movement restriction to make sure the perfect failure detector work well. To loose such movement restriction and at the same time to make sure the effective coordination of robots, an adaptive failure detector would be a good choice and could bring more effective coordination.

Self-stabilization of robot system The property of self-stabilization is very important for dependable distributed system. In this dissertation, we analyzed the self-stabilization of flocking of a group of non-oblivious robots in detail in Chapter 6.

As we know, memory corruption is one kind of transient failures in robot system. In practical robot applications, like gathering, flocking, or formation control, there are many other kinds of transient failures, such as sensor is crashed or communication between robots is disconnected temporarily. Therefore, in future work, the deeper analysis of the self-stabilization of robot system, combined with the results obtained through our analysis of flocking with memory corruption, could provide more dependable and flexible robot systems.

Bibliography

[ABLS90] A. Bondavalli, L. Simoncini, “Failure classification with respect to detection”, in Proceedings.of IEEE Workshop on Future Trends of Distributed Computing Systems, pages: 47-53, 30 Sep-2 Oct, 1990.

[ALRL04] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing”, IEEE Transactions on Depenable and Secure Computing, vol. 1, No. 1, 2004.

[AOSY99] H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita, “A distributed memoryless point convergence algorithm for mobile robots with limited visibility,” IEEE Transactions on Robotics and Automation, vol.15, no.5, pp. 818-828, October 1999.

[AP04] N. Agmon and D.Peleg, “Fault tolerant gathering algorithms for autonomous mobile robots”, In Proceding of 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004), pages 1070-1078, New Orleans, LA, USA, January 2004.

[ASER03] A. Serrani, “Robust coordinated control of satellite formations subject to gravity perturbations, in Proceedings of the American Control Conference, June 2003, vol. 1, pp. 302C307.

[BH97] D. C. Brogan and J. K. Hodgins, “Group Behaviors for Systems with Significant Dynamics”, In Autonomous Robots Journal, vol. 4, 137-153, 1997.

[BLT02] B. R. Bellur, M. G. Lewis, and F. L. Templin. An Ad-hoc Network for Teams of Autonomous Vehicles. In Proc. of IEEE Symposium on Autonomous Intelligence Networks and Systems, 2002.

[CC98] J. Coble, D. Cook, “Fault tolerant coordination of robot teams,” available at: citeseer.ist.psu.edu/coble98fault.html.

[CFK97] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng, “Cooperative mobile robotics: antecedents and directions”, Autonomous Robots, 4(1): 7-23, March 1997.

[CFPS] M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro, “Solving the robots gathering problem”, in 30th Int. Colloq. on Automata, Languagues and Programming, pp. 1181-1196, 2003.

[CP07] D. Canepa and M. G. Potop-Butucaru, “Stabilizing flocking via leader election in robot networks”, in Proceeding of 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp.52-66, Paris, France, November 14-16, 2007.

[CT96] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems”, Journal of ACM, vol. 43, no. 2, March 1996, pp. 225-267.

[Defago02] X. Defago, “Agreement-relate problems: from semi-passive replication to totally ordered broadcast”, PhD thesis, Ecole Polytechnique Federale de Lausanne, Switzerland, August 2000. Number 2229.

[DGMP06] X. Défago, M. Gradinariu, S. Messika, and P. Raipin-Parvédy, “Fault-tolerant and self-organizing mobile robots gathering”, in Proceedings of 20th International Symposium on Distributed Computing (DISC 2006), vol. 4167, pp. 46–60, September 2006.

[DISC08] Samia Souissi, Yan Yang and Xavier Defago, “Fault-tolerant flocking in a k-bounded asynchronous system”, School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), IS-RR-2008-004, September 26, 2008.

[DKG07] M. J. Daigle, X. D. Koutsoukos, and G. Biswas, “Distributed diagnosis in formations of mobile robots,” IEEE Transactions on Robotics, vol. 23, no. 2, pp. 353-369, April 2007.

[DP07] Y. Dieudonné, and F. Petit, “A Scatter of Weak Robots”, Technical Report RR07-10, LARIA, CNRS, Amiens, France, 2007.

[FPSW99] P. Flocchini, G. Prencipe, N. Santoro, P. Widmayer, ”Hard Tasks for Weak Robots: The Role of Common Knowledge in Pattern Formation by Autonomous Mobile Robots”, in Proceedings of 10th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), 1999.

[FPSW01] P. Flocchini and G. Prencipe and N. Santoro and P. Widmayer, “Pattern Formation by Autonomous Robots Without Chirality”, In Proc. of 8th Intl. Colloquium on Structural Information and Communication Complexity (SIROCCO 2001), 147–162, Juin, 2001.

[GP01] V. Gervasi, and G. Prencipe, “Flocking by A Set of Autonomous Mobile Robots”, Technical Report, ”Dipartimento di Informatica, Universitá di Pisa, Italy, TR-01-24, 2001.

[GP04] V. Gervasi and G. Prencipe, “Coordination without communication: the case of the flocking problem”, Discrete Applied Mathematics, vol. 143, no. 1-3, pp. 203-223, Sep. 2004.

[GPRE00] G.Prencipe, “ A new distributed model to control and coordinate a set of autonomous mobile robots: the CORDA model”, Technical report, TR-00-10, August 17, 2000.

[GPRE01b] G. Prencipe, “Instantaneous actions vs. full asynchrononicity: Controlling and coordinating a set of autonomous mobile robots”,in Proc. 7th Italian Conf. on Theretical Computer Science, pages 185-190, October 2001.

[GPRE02] G. Prencipe, “Distributed cooridnation of a set of autonomous mobile robots”, PhD thesis, Universita Degli Studi Di Pisa, 2002.

[HJP03] G. T. Herbert, A. Jadbabaie and G. J. Pappas, “Stable flocking of mobile agents Part II: dynamic topology”, In IEEE Conference on Decision and Control, pp. 2016–2021, 2003.

[HT02] A. T. Hayes, P. Dormiani-Tabatabaei , “Self-organized flocking with agent failure: Off-line optimization and demonstration with real robots”, in Proceedings of IEEE International Conference on Robotics and Automation, vol.4, pp. 3900- 3905, USA, May 11-15, 2002.

[ISMY96a] I. Suzuki and M. Yamashita, “Agreement on a common x-y coordinate system by a group of mobiel robots”, in Proceedings of Dagstuhl Seminar on Modeling and Planning for Sensor-Based Intelligent Robots, September 1996.

[ISMY96b] I. Suzuki and M. Yamashita, “Distributed anonymous mobile robots- formation and agreement problems”, in Proceedings of 3rd Collq. on Structural Information and COmmunication Complexity, pp. 313-330, 1996.

[JORE08] O. J. O’Loan and M. R. Evans, “Alternating steady state in one-dimensional flocking”. Journal of Physics A: Mathematical and General. Retrieved on June 13, 2008.

[JSEB00] D. J. Stilwell and B. E. Bishop, “Platoons of underwater vehicles, IEEE Control Systems Magzine, vol. 20, pp. 45 C52, Dec. 2000.

[JWFE97] J. S. Jennings, G. Whelan, and W.F. Evans. Cooperative Search and Rescue with a Team of Mobile Robots. Proc. 8th International Conference on Advanced Robotics, 193–200, 1997.

[JY98] T. John and T. Yuhai, “Flocks, Herds, and Schools: A Quantitative Theory of Flocking”, Physical Review Journal. vol. 58(4), pp. 4828-4858, 1998.

[JYJJ91] J. Y. J. Joe, “A collision avoidance algorithm for the mobile robot and the robot manipulator in multi-robot system”, Ph.D. Thesis Texas Univ., Arlington, August 1991.

[KOVA03] K. Konolige, C. Ortiz, R. Vincent, A. Agno, M. Eriksen, B. Limketkai, M. Lewis, L. Briesemeister, E. Ruspini, D. Fox, J. Ko, B. Stewart, and L. Guibas, “CENTIBOTS: Large-Scale Robot Teams”, in Journal of Multi-Robot Systems: From Swarms to Intelligent Autonoma, 2003.

[KSIS96] K. Sugihara and I. Suzuki, “Distributed algorithms for formation of geometric patterns with many mobile robots”, Journal of Robotic systems, 13(3): 127-139, 1996.

[LALB02] D. L. Akin and M. L. Bowden, “Eva, robotic and cooperative assembly of large space structures”, in Proceeding of the IEEE Aerospace Conference, March 2002.

[LC06] G. Lee and N. Y. Chong, “Decentralized formation control for a team of anonymous mobile robots,” The 6th Asian control, July 18-21, 2006, Bali, Indonesia.

[LDC05] G. Lee, X. Defago, and N. Y. Chong, “A distributed algorithm for the coordination of dynamic barricades composed of autonomous mobile robots,” in the proceedings of the International Conference on Control, Automation and Systems (ICCAS 2005), June 2-5, 2005, in Kintex, Cyeong Gi, Korea.

[LHC06] G. Lee, Y. Hanada, and N. Y. Chong, “Decentralized formation control for small-scale mobile robot teams,” in Proceedings of the 2006 JSME Conference on Robotics and Mecha tronics, Waseda, Japan, May 26-28, 2006.

[MCGP02] M. Cieliebak, G. Prencipe, “Gathering autonomous mobile robots”, in Proceeding of 9th Int. Colloq. on Stuctural Information and Communication Complexity, pp. 57-72, June 2002.

[NADP04] N. Agmon, D. Peleg, “Fault-tolerant gathering algorithms for autonomous mobile robots”, in Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 1070 - 1078,2004.

[MLPO05] M. Lindhe and P. Ogren, “Flocking with Obstacle Avoidance: A New Distributed Coordination Algorithm Based on Voronoi Partitions”, in the Proceedings of the IEEE International Conference on Robotics and Automation Barcelona, Spain, April 2005.

[PREC01] G. Prencipe, “CORDA: Distributed Coordination of a Set of Autonomous Mobile Robots”, in Proceding of European Research Seminar on Advances in Distributed Systems, pp. 185-190, Bertinoro, Italy, May 2001.

[PTF01] L. Parker, C. Touzet, and F. Fernandez, “Techniques for learning in multi-robot teams”, in T. Balch and L. Parker, editors, Robot teams: From Diversity to Polymorphism, A. K. Peters, 2001.

[RCM04] P. Renaud, E. Cervera, and P. Martinet, “Towards a reliable vision-based mobile robot formation control”, in Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3176-3181, Sendai, Japan, Sep. 28 - Oct. 2, 2004.

[Rey87] C. W. Reynolds, “Flocks, Herds, and Schools: A distributed Behavioral Model”, Journal of Ccomputer Graphics, vol. 21 (1), 79-98, 1987.

[ROS06] R. Olfati-Saber, “Flocking for multi-agent dynamic systems: algorithm and theory,” IEEE Transactions on automatic control, vol. 51, no.3, March 2006.

[RCH98] A. Robertson, T. Corazzini, and J. P. How, “Formation sensing and control technologies for a separated spacecraft interferometer, in Proceedings of the American Control Conference,, June 1998, vol. 3, pp. 1574C1579.

[RSGHP02] P. E. Rybski, S. A. Stoeter, M. Gini, D. F. Hougen, and N. P. Papanikolopoulos, “Performance of a distributed robotic system using shared communications channels”, IEEE transactions on Robotics and Automation, pages 713-727, October 2002.

[SHP04] D. P. Scharf, F. Y. Hadaegh, and S. R. Ploen, “A survey of space formation flying guidance and control (part 2), in Proceedings of the American Control Conference, Boston, Massachusetts, June 2004.

[SCHR01] K. Schreiner, “NASA’s JPL Nanorover Outposts Project Develops Colony of Solar-powered Nanorovers”, In IEEE DS Online, 3(2), 2001.

[SM03] R. O. Saber and R. M. Murray, “Flocking with Obstacle Avoidance: Cooperations with Limited Communication in Mobile Networks,” in Proceedings of the 42th IEEE Conference on Decision and Control, pp. 2022 - 2028, Maui, Hawaii, USA, Dec. 2003.

[SY99] I. Suzuki and M. Yamashita, “Distributed anonymous mobile robot: Formation of geometric patterns,” SIAM Journal of Computing, vol. 28, no.4, pp. 1347-1363, 1999.

[SHPB01] P. S. Schenker, T. L. Huntsberger, P. Pirjanian, and E. T. Baumgartner, “Planetary rover developments supporting mars exploration, sample return and future human-robotic colonization”, in Proceedings of the 10th Conference on Advanced Robotics, pages 31-47, 2001.

[SYD08] S. Souissi, Y. Yang and X. Défago, “Fault-tolerant flocking in a k-bounded asynchronous system”, accepted by 12th International Conference On Principles Of Distributed Systems December, 2008, to appear.

[Gerard00] Gerard Tel, “Introduction to distributed algorithms”, Second Edition, Cambridge, 2000.

[VGGP01] V. Gervasi, G. Prencipe, “Need a Fleet? Use The Force!”, in FUN With Algorithms 2 (FUN 2001), pages 149-164, Elba, Italy, May 2001.

[YDCW07] R. Yared, X. Defago, J. I. Cartigny and M. Wiesmann, “Collision prevention platform for a dynamic group of asynchronous cooperative mobile robots”, Journal of Networks, 2(4):28C39, August 2007.

[YMF97] D. Yoshida, T. Masuzawa and H. Fujiwara, “Fault-tolerant distributed algorithms for autonomous mobile robots with crash faults,” Systems and Computers in Japan, vol. 28, no. 2, 1997.

[YSD08] Y. Yang, S. Souissi, and X. Défago, “ Fault-tolerant Flocking in a k-bounded Semi-synchronous System with Flock Rotation”, Submitted for publication.

[YB96] H. Yamaguchi and G.Beni, “Distributed Autonomous Formation Control of Mobile Robot Groups by Swarm-based Pattern Generation”, Proc. of the 2nd Int. Symp. on Distributed Autonomous Robotic Systems (DARS 96), pp. 141-155, 1996.

[YSD09] . Yang, S. Souissi, X. Défago, “Fault-tolerant flocking in a k-bounded semi-synchronous system with flock rotation”, accepted by the IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA-09),Bradford, United Kingdom, 2009.

[YXCD08] Y. Yang, N. Xiong, N. Y. Chong, X. Défago, “A Decentralized and Adaptive Flocking Algorithm for Autonomous Mobile Robots,” in the Proceeding of International Symposium on Advances in Grid and Pervasive Systems (GPC 2008), pp. 262-268, Kunming, China, May, 2008.

[ZXKHYP04] X. Zhang, R. Xu, C. Kwan, L. Haynes, Y. Yang, M. M. Polycarpou, “Fault tolerant formation flight control of UAVs”, International Journal of Vehicle Autonomous Systems, vol. 2, no.3-4, pp. 217 - 235, 2004.

Publications

***In direct relation with the research*:**

[1] Y. Yang, S. Souissi, X. Défago, “Fault-tolerant flocking in a k-bounded semi-synchronous system with flock rotation”, accepted by the IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA-09),Bradford, United Kingdom, 2009, to appear.

[2] S. Souissi, Y. Yang and X. Défago, “Fault-tolerant flocking in a k-bounded asynchronous system”, accepted by 12th International Conference On Principles Of Distributed Systems, December 2008, pp. 145-163.

[3] Y. Yang, N. Xiong, N. Y. Chong, X. Défago, “A Decentralized and Adaptive Flocking Algorithm for Autonomous Mobile Robots,” in the Proceeding of International Symposium on Advances in Grid and Pervasive Systems (GPC 2008), pp. 262-268, Kunming, China, May, 2008.

[4] N. Xiong, Y. Yang, X. Défago, “Comparative Analysis of QoS and Memory Usage of Adaptive Failure Detectors,” The 13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC’07), pp. 27-34, Melbourne, Victoria, Australia, December, 2007, pp. 27-34.

[5] X. Défago, N. Xiong, Y. Yang, and N. Hayashibara“Pragmatic Accrual Failure Detection with Kappa-FD”, Technical Report, Japan Advanced Institute of Science and Technology.

[6] S. Souissi, Y. Yang, X. Défago, “Fault-tolerant flocking in a k-bounded Asynchronous System”, School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), IS-RR-2008-004, September 26, 2008.

***On other earlier research*:**

[7] Y. Yang, L. Tan and N. Xiong, “PDAVQ: An adaptive virtual queue algorithm based on the proportional and differential control”, Journal of China Institute of Communications, vol. 26, no. 3, 2005, pp.39-44.

[8] Y. Yang, L. Tan, N. Xiong and X. Zhan, “A Novel TCP Algorithm in Multiple Bottleneck Network”, MINI-MICRO SYSTEMS, vol. 26, no. 2, 2005, pp. 186-191.

[9] L. Tan, Y. Yang, C. Lin and N. Xiong, “PID-RPR: A high performance bandwidth allocation approach for RPR networks”, IEICE Transactions on Communications, vol. E88-B, no.7, July 2005.

[10] L. Tan, Y. Yang, C. Lin, N. Xiong and M. Zukerman, “Scalable Parameter Tuning for AVQ”, IEEE Communications Letters, vol. 9, no.1, Jan. 2005. (Impact Factor: 1.196)

[11] L. Tan, Y. Yang, W. Zhang and M. Zukerman, “On Control Gain Selection in Dynamic-RED”, IEEE Communications Letters, vol. 9, no.1, Jan. 2005. (Impact Factor: 1.196)

[12] L. Tan, N. Xiong, Y. Yang and P. Yang, “A Consolidation Algorithm for Multicast Service Using Proportional Control and Neural Network Predictive Techniques”, Computer Communications, vol.29, no.1, pp.114-122, Dec.2005.

[13] N. Xiong, L. Tan and Y. Yang, “An Approach for Regulating the Transmission Rate in Multicast Congestion Control”, Journal of China Institute of Communications, vol.25, no.11, 2004.11, pp.142-150.

[14] L. Tan, N. Xiong, Y. Yang,, “A PGM-based Single-rate Multicast Congestion Control Scheme”,Journal of Software, Vol.15, No.10, 2004.10, pp.1538-1546.

[15] Y. He, N. Xiong, Y. Yang, “A TCP Congestion Control Algorithm in Multiple Bottleneck Networks”, Journal of Computer Research and Development, Vol. 42, No.12, 2005, pp.2070-2076.

[16] L. Tan, N. Xiong, and Y. Yang, “One class of rate control schemes for many-to-many multicast congestion control”, Journal of Postgraduates in Wuhan University (Natural Science Edition), Vol.21, No.3, pp.55-62, 2004.

[17] N. Xiong, Y. Yang, “A Note on Intrusion Detection in Telnet”, Journal of Central China Normal University, Vol.38, No.2, pp.160-164, February 2004.

[18] N. Xiong, L. Tan and Y. Yang, “A Novel Congestion Control Algorithm Using the BP Neural Network”, Computer Engineering, vol. 30, no. 24, pp. 35-36, 2004.

[19] Y. Yang, L. Tan and N. Xiong, “A resource-based admission control algorithm for grid computing systems”, in Proceedings of The Fourth International Conference on Computer and Information Technology (CIT 2004), Sept. 14-16, 2004, Wuhan, China.

[20] N. Xiong, X. Dfago, X. Jia, Y. Yang, Y. He, “Design and Analysis of a Self-tuning Proportional and Integral Controller for Active Queue Management Routers to Support TCP Flows”, IEEE INFOCOM 2006, Barcelona, Spain, April 23-29, 2006. (the 25th Annual Conference on Computer Communications, Impact Factor: 1.39)

[21] Y. He, N. Xiong, Y. Yang and C. Lin, “EBA: An efficient bandwidth allocation approach for RPR networks”, In the Proceedings of ACM SIGCOMM ASIA WORKSHOP 2005, ACM, April 2005, Beijing, China, pp. 175-181.

[22] N. Xiong, Y. Yang, Y. He, “On the Quality of Service of Failure Detectors Based on Control Theory”, The IEEE 20th International Conference on Advanced Information Networking and Applications (AINA2006), Vienna, Austria, pp. 75-80, April 18-20, 2006.

[23] N. Xiong, X. Dfago, Yan Yang, Y. He, and J. He, “On Control Gain Selection in PI-RED”, 2006 IEEE International Conference On Networking, Sensing and Control (ICNSC06), Ft. Lauderdale, Florida, USA, April 23-25, 2006, pp.516-522.

[24] N. Xiong, Y. He, J. Cao and Y. Yang,“On Designing a Novel PI Controller for AQM Routers Supporting TCP Flows”, in the Proc. of the Seventh Asia Pacific Web Conference (APWeb05), March, Shanghai, China, LNCS, Vol. 3399, pp. 991-1002, 2005.

[25] N. Xiong, Y. He, L. T. Yang, and Y. Yang, “A Self-tuning Reliable Dynamic Scheme for Multicast Flow Control”, The 3rd International Conference on Ubiquitous Intelligence and Computing (UIC-06), Wuhan, China, LNCS 4159/2006, book: Ubiquitous Intelligence and Computing, pp. 351-360, September, 2006.

[26] N. Xiong, Y. He, Y. Yang, L. T. Yang, C. Peng, “A Self-tuning Multicast Flow Control Scheme Based on Autonomic Technology”, The 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC’06), USA, pp. 219-226, September, 2006.

[27] N. Xiong, Y. He, Y. Yang, J. Cao and C. Lin, “An Efficient Flow Control Algorithm for Multi-rate Multicast Networks”, The 2004 IEEE International Workshop on IP Operations & Management (IPOM 2004), October, 2004, Beijing, China, pp.74-81.

[28] Y. He, N. Xiong and Y. Yang, “Data Transmission Rate Control in Computer Networks using Neural Predictive Networks”, The Proceeding of the 2004 International Symposium on Parallel Processing and Applications (ISPA 2004), LNCS, Vol. 3358, pp. 875-887.

[29] N. Xiong, X. Dfago, Y. He and Y. Yang, “A Resource-based Server Performance Control for Grid Computing Systems”, The IFIP International Conference on Network and Parallel Computing 2005 (NPC 2005), Nov. 30- Dec. 2, 2005, Beijing, China, pp. 56-64.

[30] Y. He, N. Xiong, X. Dfago, Y. Yang and J. He, “A Single-pass Online Data Mining Algorithm Combined with Control Theory with Limited Memory in Dynamic Data Streams”, The 4th International Conference on Grid and Cooperative Computing (GCC’05), November 30-December 3, 2005, Beijing, China, pp. 1119-1130.

[31] N. Xiong, Y. Yang, J. He, and Y. He, “On Designing QoS for Congestion Control Service Using Neural Network Predictive Techniques,” in the proceedings of IEEE International Conference On Granular Computing (IEEE-GrC 2006), pp. 299 - 304, Atlanta, USA, May 10-14, 2006.

[32] N. Xiong, L. T. Yang, Yan Yang, X. Dfago, Y. He, “A novel numerical algorithm based on self-tuning controller to support TCP flows”, Mathematics and Computers in Simulation 79(4): 1178-1188 (2008).